Fine-grained language composition







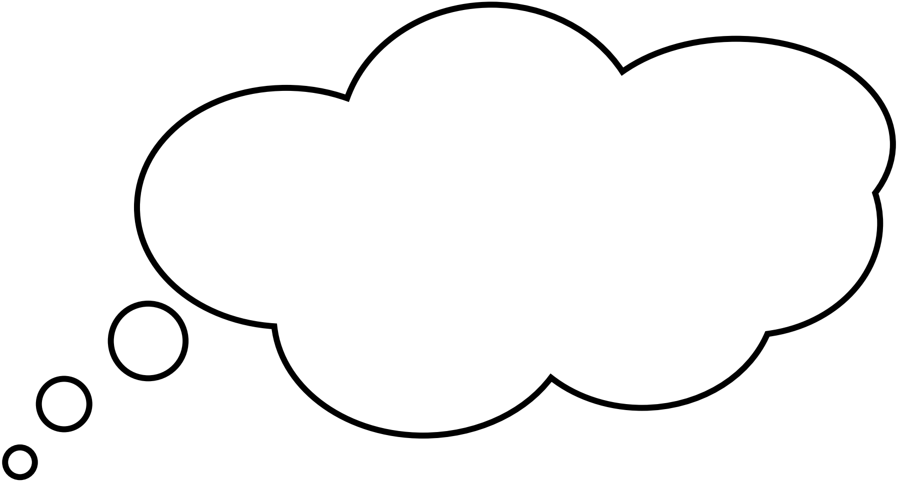
Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, Geoff French,
Sarah Mount, Laurence Tratt, Naveneetha Krishnan Vasudevan

KING'S College LONDON

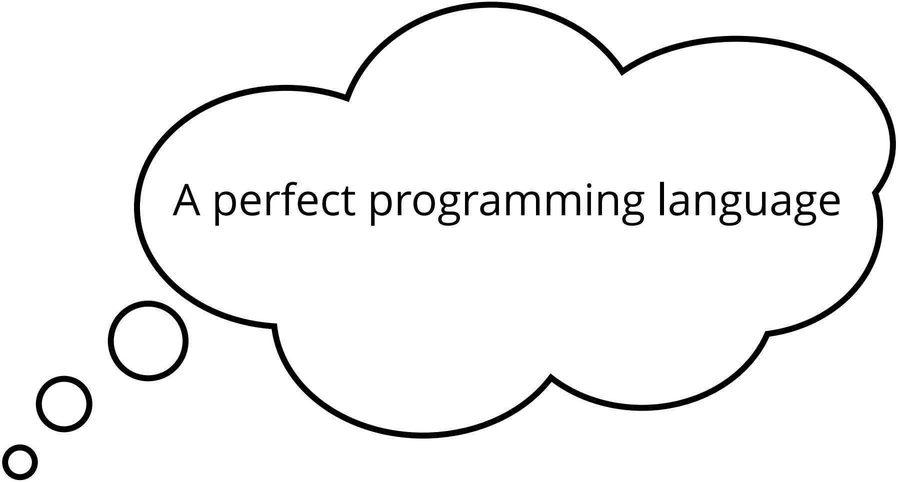
Software Development Team

2016-05-17

Background



Background



Solution

Solution

A new programming language

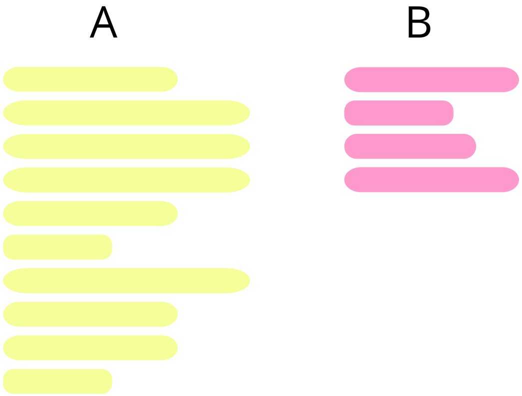
Background

Reality

Reality

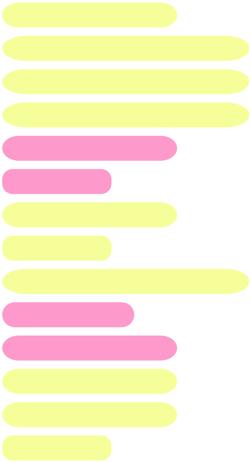
Another imperfect programming language

What to expect from this talk



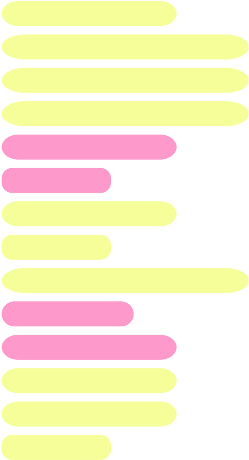
What to expect from this talk

$A \cup B$



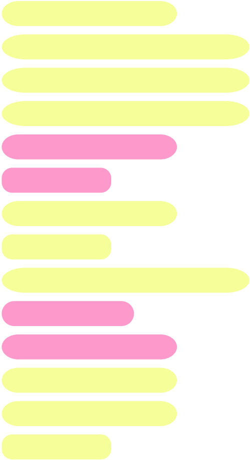
What to expect from this talk

Python $\cup$ Prolog



What to expect from this talk

Python ∪ PHP



Two levels of challenge

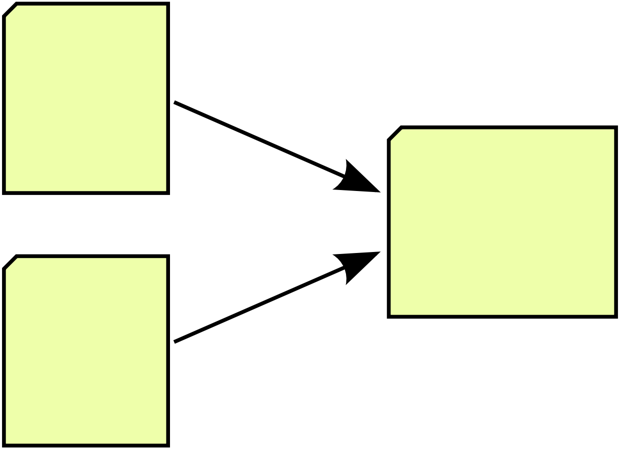
Tooling

Two levels of challenge

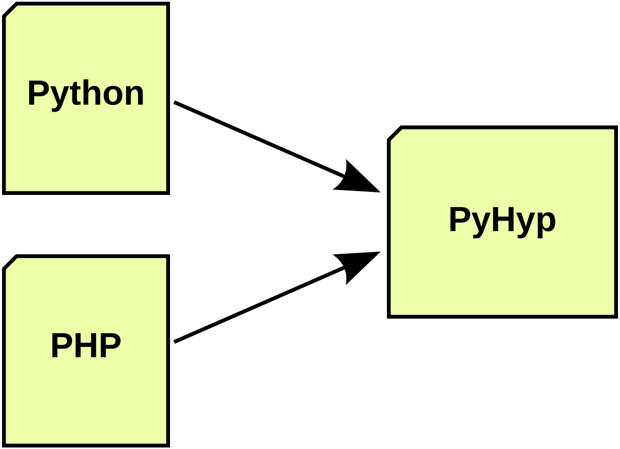
Tooling

Language friction

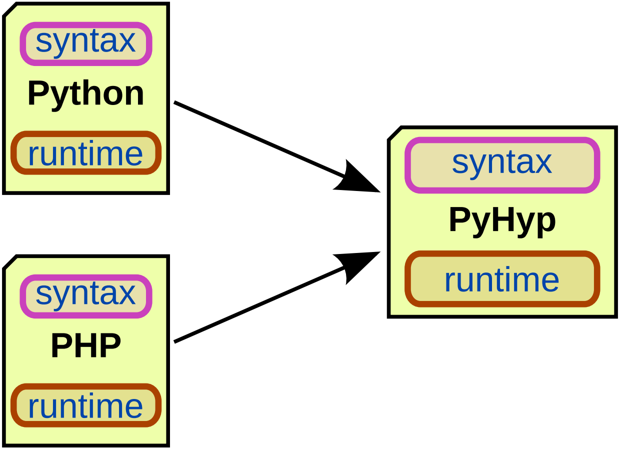
Tooling challenges



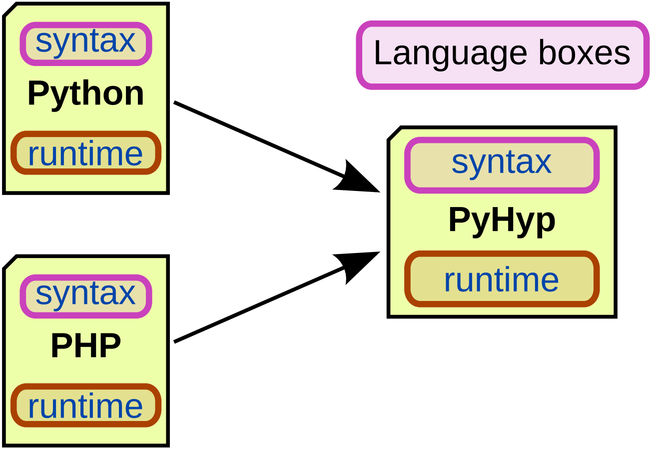
Tooling challenges



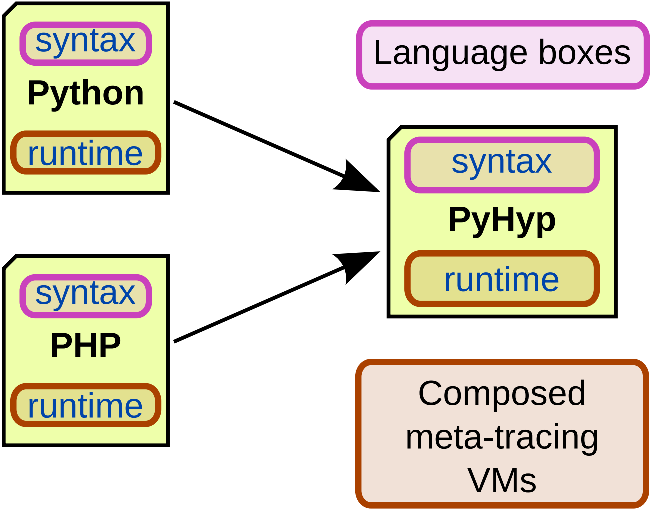
Tooling challenges



Tooling challenges



Tooling challenges



Syntax composition

PL X

<grammar>

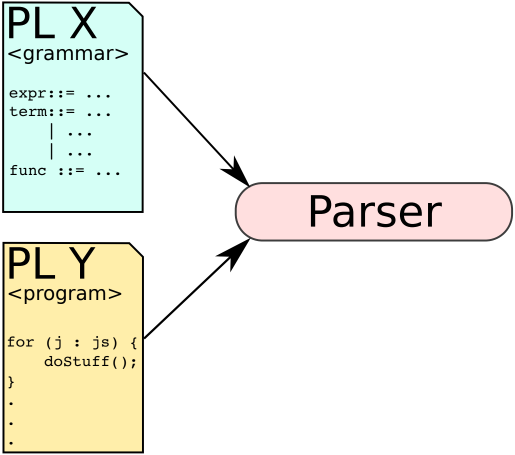
```
expr::= ...
term::= ...
    | ...
    | ...
func ::= ...
```

PL Y

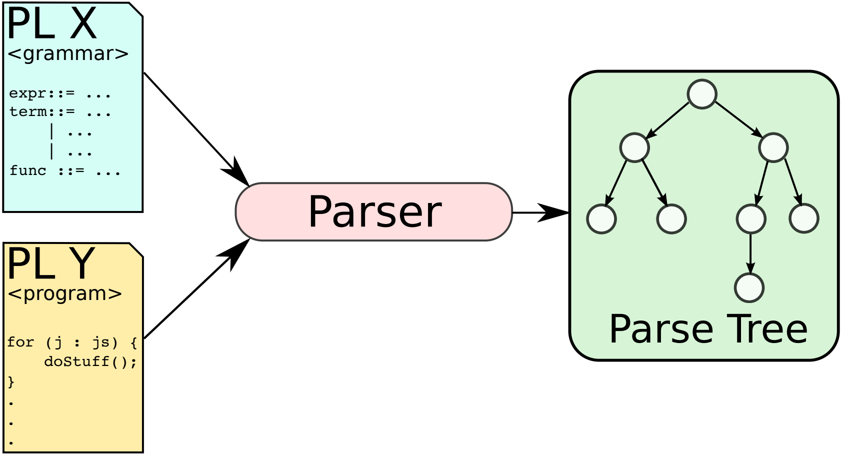
<program>

```
for (j : js) {
    doStuff();
}
.
.
.
```

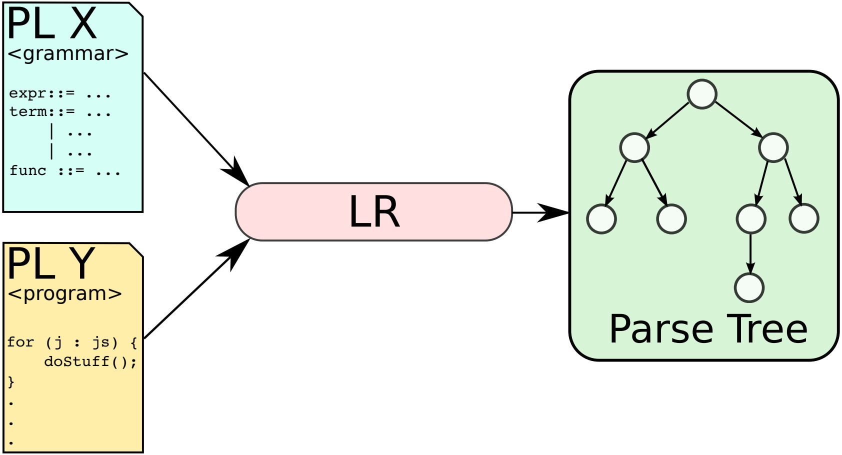
Syntax composition



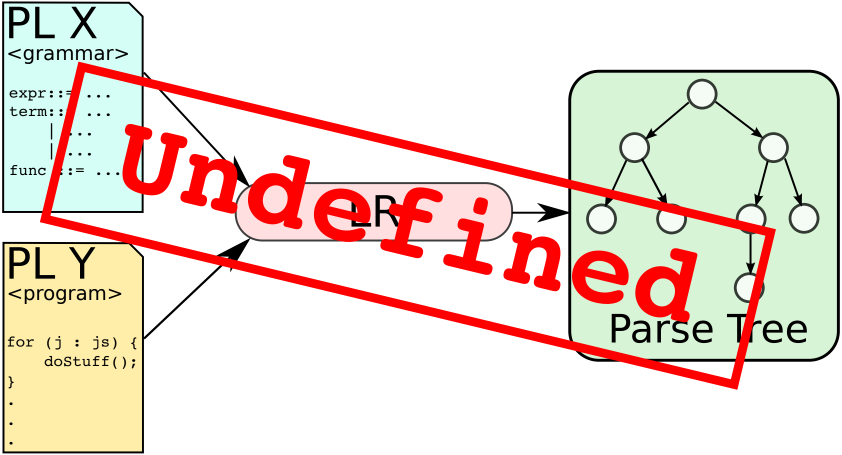
Syntax composition



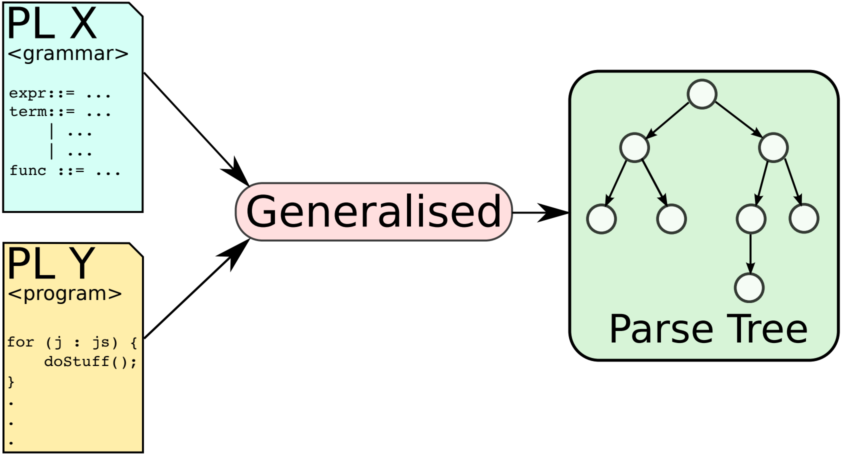
Syntax composition



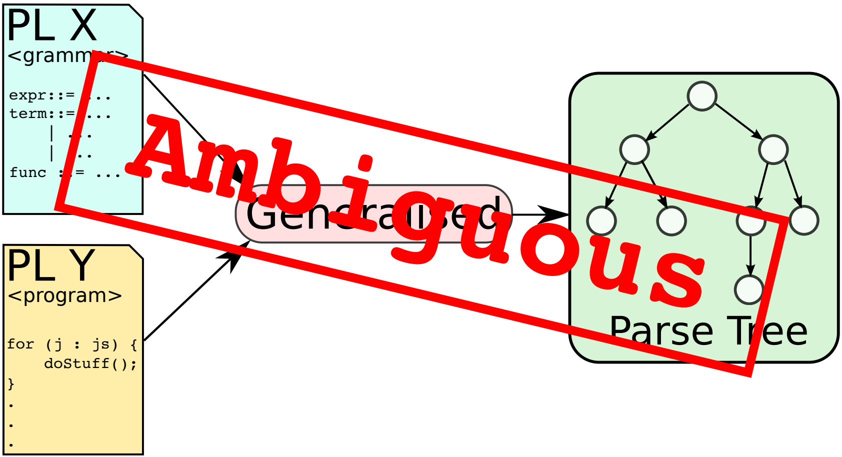
Syntax composition



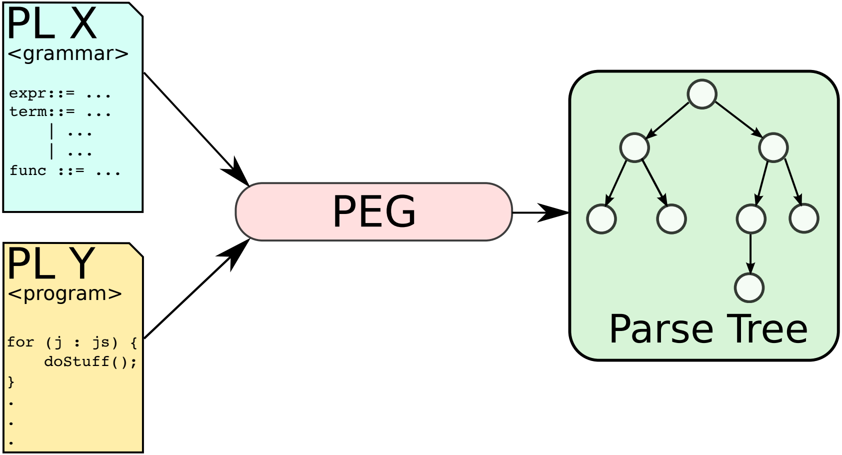
Syntax composition



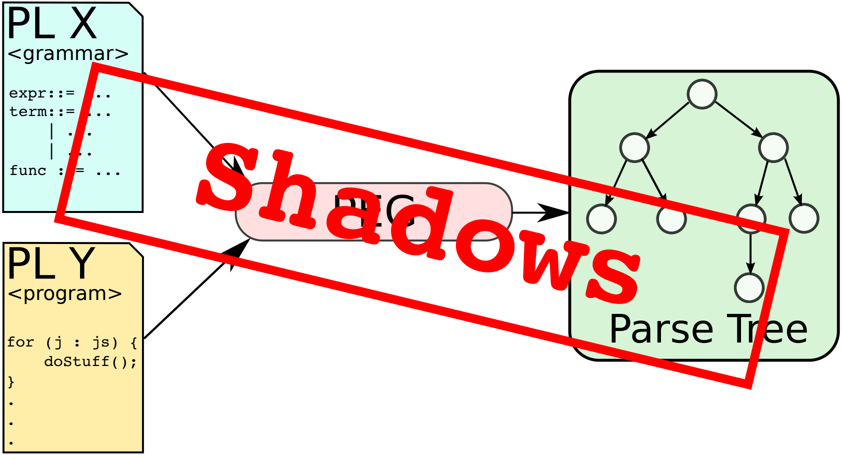
Syntax composition



Syntax composition



Syntax composition



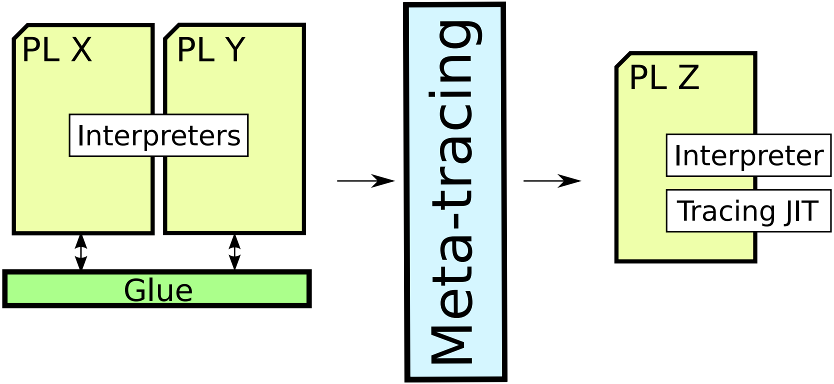
The only choice?

SDE

Challenge:
SDE's power $+$
a text editor feel?

Eco demo

Runtime composition



Performance demo

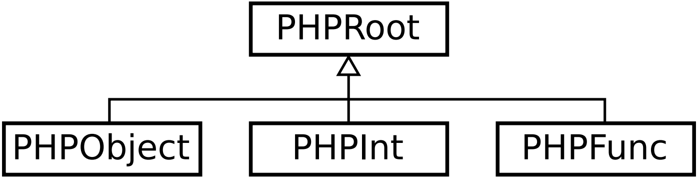
Composed Richards vs. other VMs

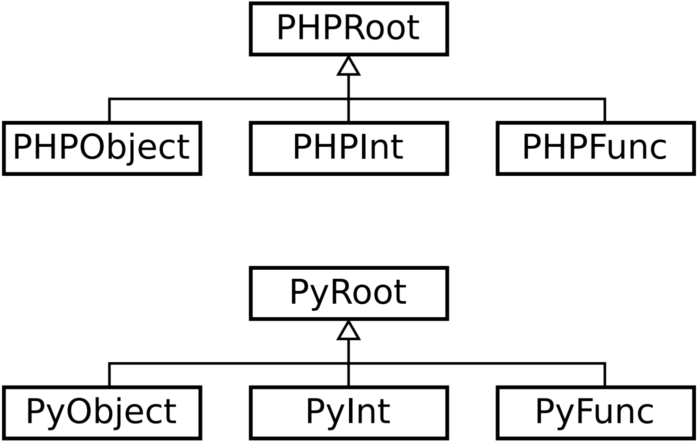
Type	VM	
Mono	CPython 2.7.7	9.475 $\pm$ 0.0127
	HHVM 3.4.0	4.264 $\pm$ 0.0386
	HippyVM	0.250 $\pm$ 0.0008
	PyPy 2.4.0	0.178 $\pm$ 0.0006
	Zend 5.5.13	9.070 $\pm$ 0.0361

Composed Richards vs. other VMs

Type	VM	
Mono	CPython 2.7.7	9.475 $\pm$ 0.0127
	HHVM 3.4.0	4.264 $\pm$ 0.0386
	HippyVM	0.250 $\pm$ 0.0008
	PyPy 2.4.0	0.178 $\pm$ 0.0006
	Zend 5.5.13	9.070 $\pm$ 0.0361
Composed	PyHyp	0.335 $\pm$ 0.0012

Datatype conversion




PHPRoot
PHPObject
PHPInt
PHPFunc
PyRoot
PyObject
PyInt
PyFunc

Datatype conversion: primitive types

PHP	Python

Datatype conversion: primitive types

PHP	Python
$\boxed{\text{2 : PHPInt}}$	

Datatype conversion: primitive types

PHP	Python
2 : PHPInt	2 : PyInt

Datatype conversion: user types

PHP	Python

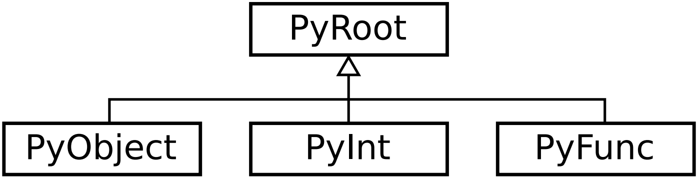
Datatype conversion: user types

PHP

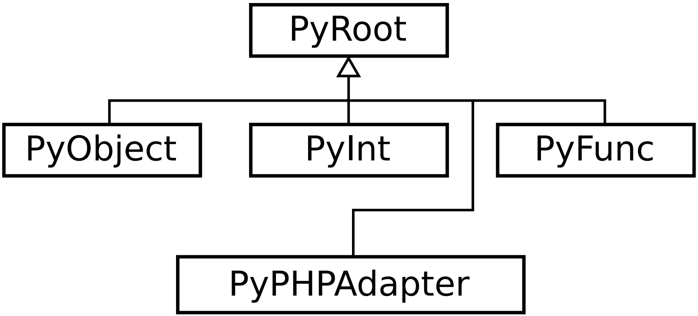
Python

o : PHPObject

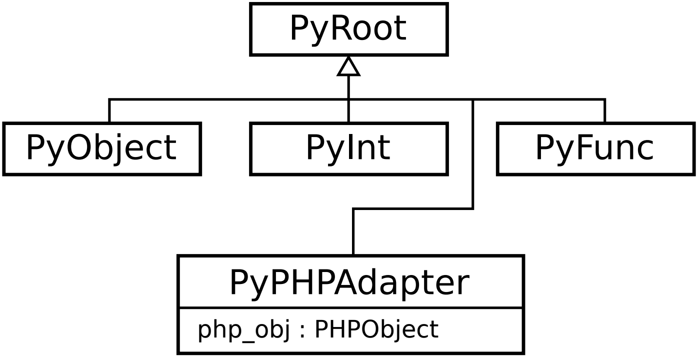
Datatype conversion: user types



Datatype conversion: user types



Datatype conversion: user types



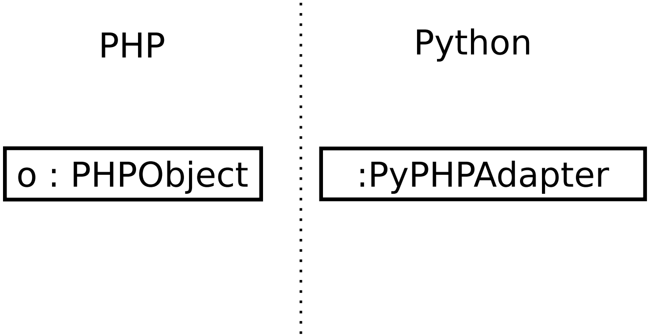
Datatype conversion: user types

PHP

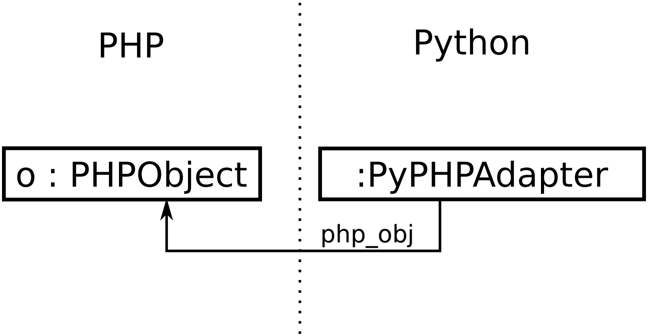
Python

o : PHPObject

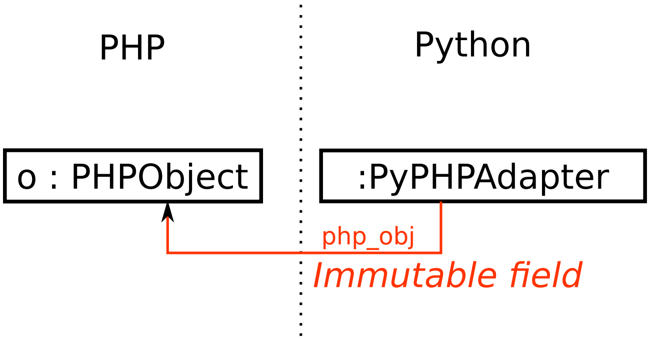
Datatype conversion: user types



Datatype conversion: user types



Datatype conversion: user types



Friction

A good composition needs to reduce *friction*.

Friction

A good composition needs to reduce *friction*. Some examples:

- Lexical scoping (or lack thereof) in PHP and Python (semantic friction)

Friction

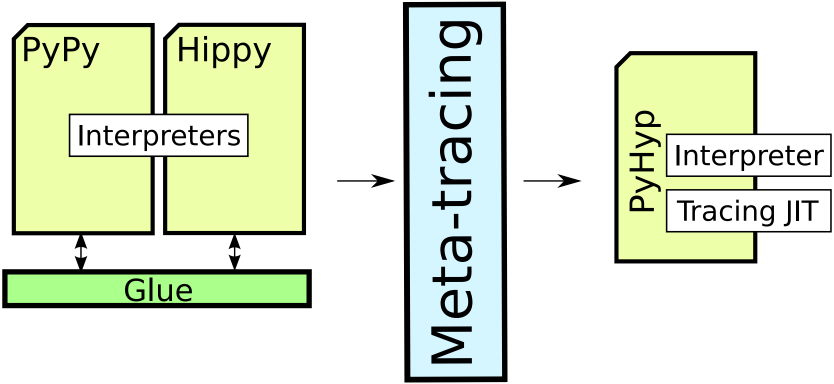
A good composition needs to reduce *friction*. Some examples:

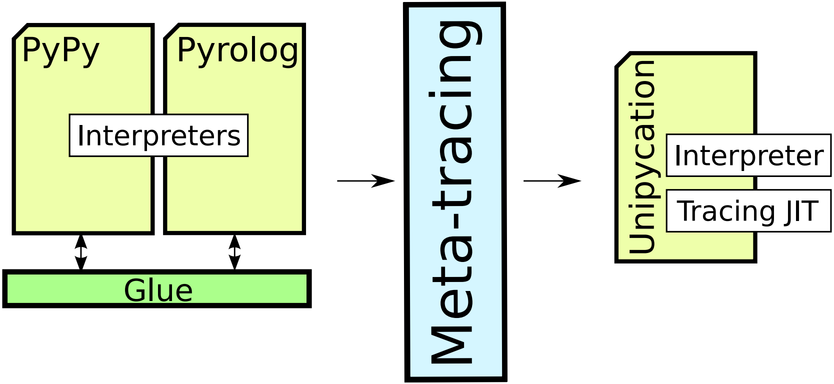
- Lexical scoping (or lack thereof) in PHP and Python (semantic friction)
- PHP datatypes are immutable except for references and objects; Python's are largely mutable (semantic and performance friction)

Friction

A good composition needs to reduce *friction*. Some examples:

- Lexical scoping (or lack thereof) in PHP and Python (semantic friction)
- PHP datatypes are immutable except for references and objects; Python’s are largely mutable (semantic and performance friction)
- PHP has only dictionaries; Python has lists and dictionaries (semantic friction)


PyPy
Hippy
Interpreters
Glue
Meta-tracing
PyHyp
Interpreter
Tracing JIT


PyPy
Pyrolog
Interpreters
Glue
Meta-tracing
Unipycation
Interpreter
Tracing JIT

Unipycation demo

Absolute timing comparison

VM	Benchmark	*Python*		*Prolog*		*Python* $\rightarrow$ *Prolog*	
CPython-SWI	SmallFunc	0.125s	$\pm$0.007	0.257s	$\pm$0.002	28.893s	$\pm$0.227
	L1A0R	2.924s	$\pm$0.284	7.352s	$\pm$0.048	9.310s	$\pm$0.084
	L1A1R	4.184s	$\pm$0.038	18.890s	$\pm$0.111	20.865s	$\pm$0.067
	NdL1A1R	7.531s	$\pm$0.080	18.643s	$\pm$0.197	667.682s	$\pm$6.895
	TCons	264.415s	$\pm$2.250	48.819s	$\pm$0.252	2185.150s	$\pm$18.225
	Lists	9.374s	$\pm$0.059	25.148s	$\pm$0.221	2207.304s	$\pm$16.073
Unipycation	SmallFunc	0.001s	$\pm$0.000	0.006s	$\pm$0.001	0.001s	$\pm$0.000
	L1A0R	0.085s	$\pm$0.000	0.086s	$\pm$0.000	0.087s	$\pm$0.000
	L1A1R	0.112s	$\pm$0.000	0.114s	$\pm$0.000	0.115s	$\pm$0.000
	NdL1A1R	0.500s	$\pm$0.003	0.548s	$\pm$0.085	2.674s	$\pm$0.012
	TCons	6.053s	$\pm$0.288	2.444s	$\pm$0.003	36.069s	$\pm$0.225
	Lists	0.845s	$\pm$0.002	1.416s	$\pm$0.003	5.056s	$\pm$0.035
Jython-tuProlog	SmallFunc	0.088s	$\pm$0.003	3.050s	$\pm$0.053	52.294s	$\pm$0.475
	L1A0R	1.078s	$\pm$0.009	206.590s	$\pm$3.846	199.963s	$\pm$2.476
	L1A1R	2.145s	$\pm$0.232	293.311s	$\pm$5.691	294.781s	$\pm$6.193
	NdL1A1R	7.939s	$\pm$0.457	1857.687s	$\pm$5.169	1990.985s	$\pm$15.071
	TCons	543.347s	$\pm$3.289	8014.477s	$\pm$17.710	8202.362s	$\pm$24.904
	Lists	5.661s	$\pm$0.046	6981.873s	$\pm$18.795	5577.322s	$\pm$15.754

Relative timing comparison

VM	Benchmark	$\frac{Python \to Prolog}{Python}$		$\frac{Python \to Prolog}{Prolog}$		$\frac{Python \to Prolog}{\text{Unipycation}}$	
CPython-SWI	SmallFunc	231.770×	±13.136	112.567×	±1.242	27821.079×	±2331.665
	L1A0R	3.184×	±0.300	1.266×	±0.014	107.591×	±0.995
	L1A1R	4.987×	±0.049	1.105×	±0.007	181.899×	±0.590
	NdL1A1R	88.654×	±1.368	35.814×	±0.554	249.737×	±2.922
	TCons	8.264×	±0.101	44.760×	±0.453	60.583×	±0.637
	Lists	235.459×	±2.314	87.772×	±1.017	436.609×	±4.415
Unipycation	SmallFunc	1.295×	±0.105	0.182×	±0.054	1.000×	
	L1A0R	1.020×	±0.002	1.012×	±0.002	1.000×	
	L1A1R	1.025×	±0.002	1.002×	±0.003	1.000×	
	NdL1A1R	5.349×	±0.045	4.879×	±0.924	1.000×	
	TCons	5.959×	±0.282	14.756×	±0.092	1.000×	
	Lists	5.982×	±0.045	3.569×	±0.026	1.000×	
Jython-tuProlog	SmallFunc	592.904×	±19.517	17.143×	±0.338	50354.204×	±4341.413
	L1A0R	185.460×	±2.818	0.968×	±0.021	2310.844×	±28.093
	L1A1R	137.427×	±14.537	1.005×	±0.028	2569.873×	±52.847
	NdL1A1R	250.776×	±14.666	1.072×	±0.009	744.699×	±6.726
	TCons	15.096×	±0.106	1.023×	±0.004	227.409×	±1.592
	Lists	985.149×	±8.674	0.799×	±0.003	1103.206×	±8.338

What can we use this for?

What can we use this for?

First-class languages

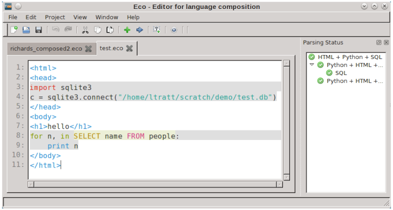
First-class languages

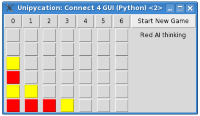
Language migration

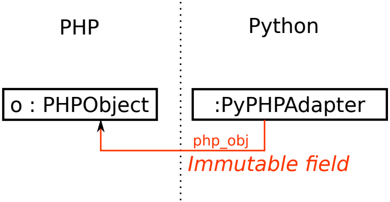
Thanks to our funders

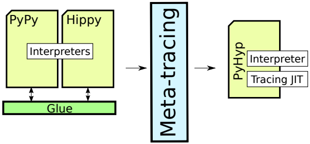
- EPSRC: *COOLER* and *Lecture*.
- Oracle: various.

End of part one...









VM Warmup Blows Hot and Cold





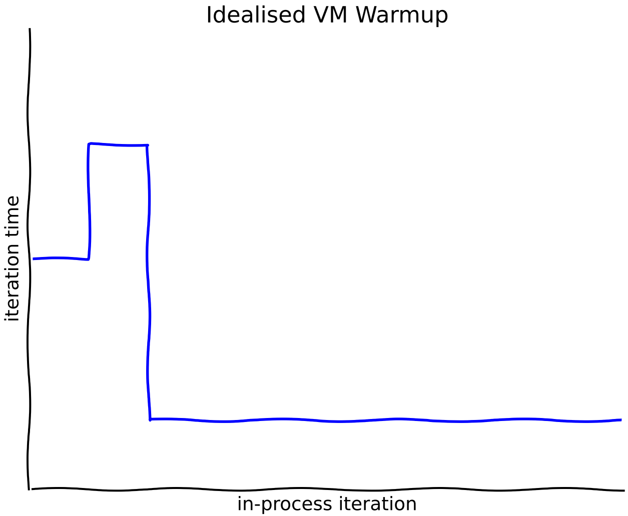

Edd Barrett, Carl Friedrich Bolz, Rebecca Killick (Lancaster),
Vincent Knight (Cardiff), Sarah Mount, Laurence Tratt

KING'S College LONDON

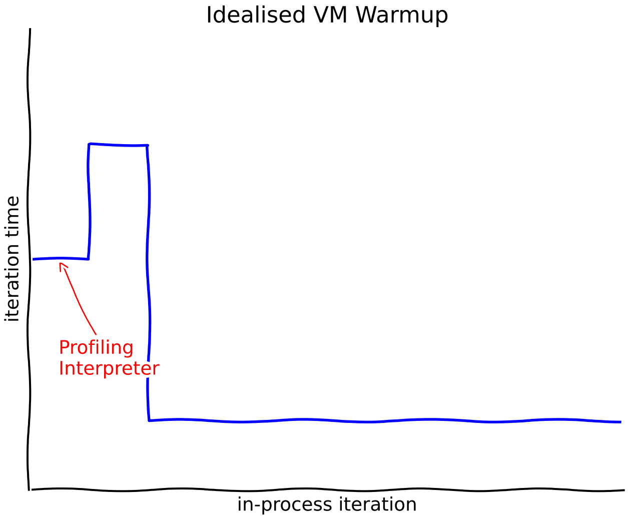
Software Development Team
2016-05-17

What is ‘warmup’?

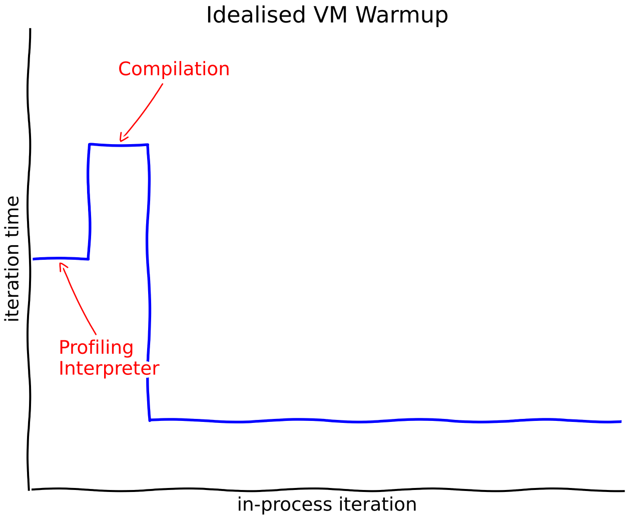
What is ‘warmup’?



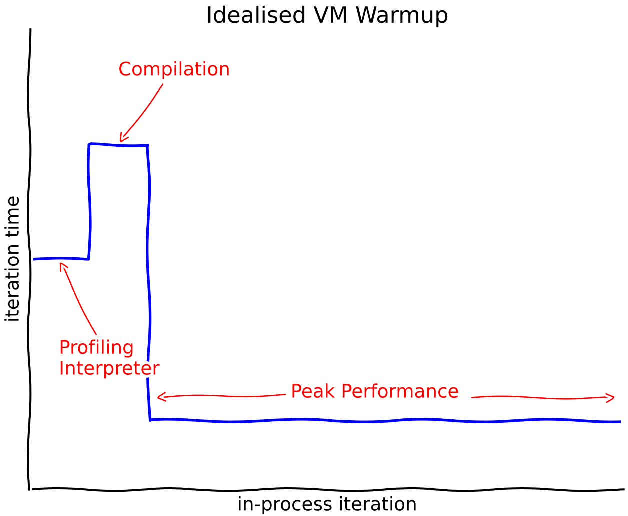
What is ‘warmup’?



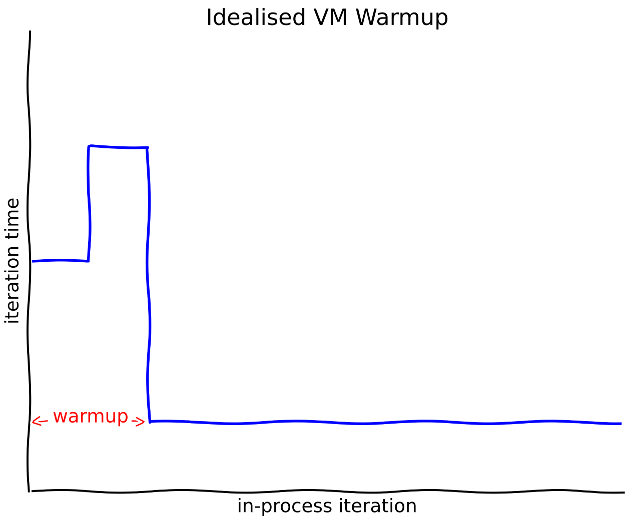
What is 'warmup'?

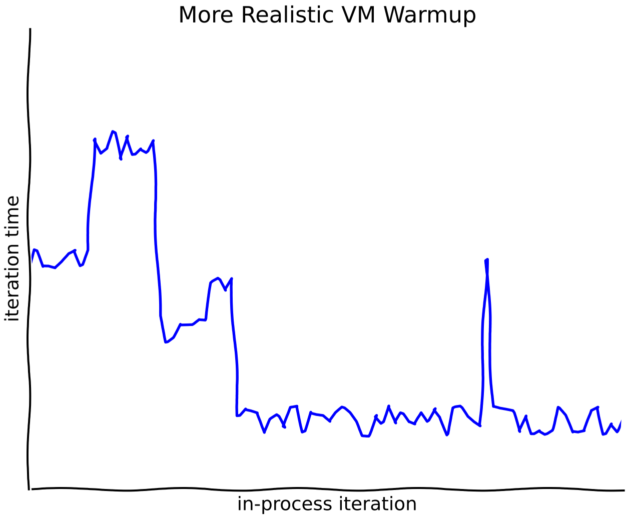


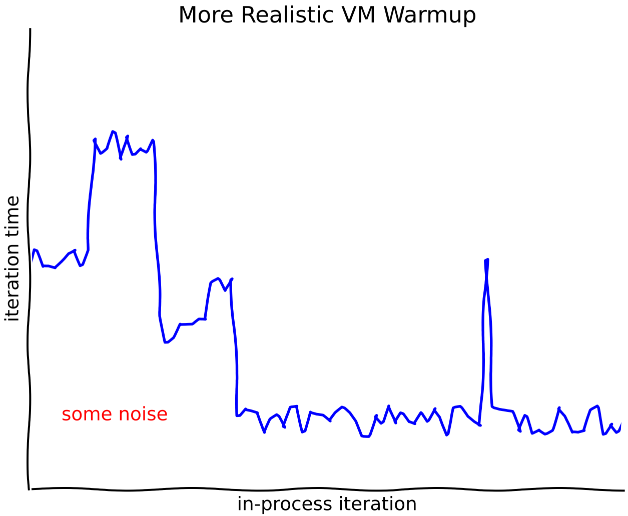
What is 'warmup'?

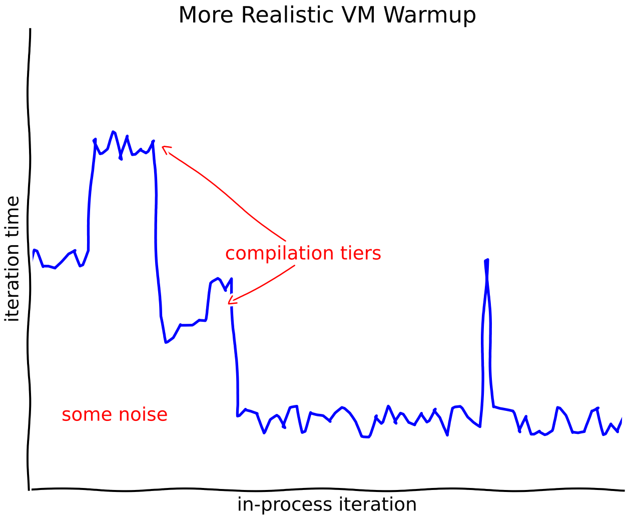


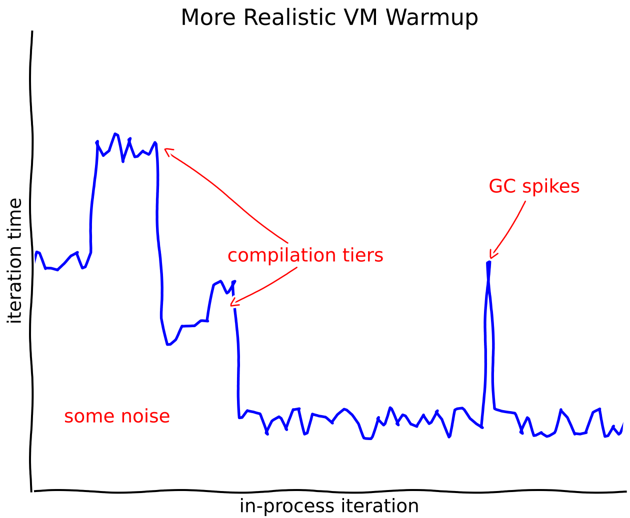
What is ‘warmup’?




More Realistic VM Warmup
iteration time
in-process iteration


More Realistic VM Warmup
iteration time
some noise
in-process iteration


More Realistic VM Warmup
iteration time
compilation tiers
some noise
in-process iteration


More Realistic VM Warmup
iteration time
in-process iteration
compilation tiers
GC spikes
some noise

Why care about warmup?

We report: peak performance

Why care about warmup?

We report: peak performance

Users see: warmup + peak performance

The Warmup Experiment

Hypothesis: Small, deterministic programs exhibit classical warmup behaviour

Method 1: Which benchmarks?

Method 1: Which benchmarks?

The language benchmark games are perfect for us (unusually)

Method 1: Which benchmarks?

The language benchmark games are perfect for us (unusually)

We removed any CFG non-determinism

Method 1: Which benchmarks?

The language benchmark games are perfect for us (unusually)

We removed any CFG non-determinism

We added checksums to all benchmarks

Method 2: How long to run?

Method 2: How long to run?

10 *process executions*

Method 2: How long to run?

10 *process executions*

Within each process execution,
2000 *in-process iterations*

Method 3: VMs

- Graal-0.13
- HHVM-3.12.0
- JRuby/Truffle (git #f82ac771)
- Hotspot-8u72b15
- LuaJit-2.0.4
- PyPy-4.0.1
- V8-4.9.385.21
- GCC-4.9.3

Note: same GCC (4.9.3) used for all compilation

Method 4: Machines

- Linux-Debian8/i4790K, 24GiB RAM
- Linux-Debian8/i4790, 32GiB RAM
- OpenBSD-5.8/i4790, 32GiB RAM

Method 4: Machines

- Linux-Debian8/i4790K, 24GiB RAM
- Linux-Debian8/i4790, 32GiB RAM
- OpenBSD-5.8/i4790, 32GiB RAM

- Turbo boost and hyper-threading disabled
- SSH blocked from non-local machines
- Daemons disabled (cron, smtpd)

Method 5: Krun

Method 5: Krun

Benchmark runner: tries to control as many confounding variables as possible

Method 5: Krun

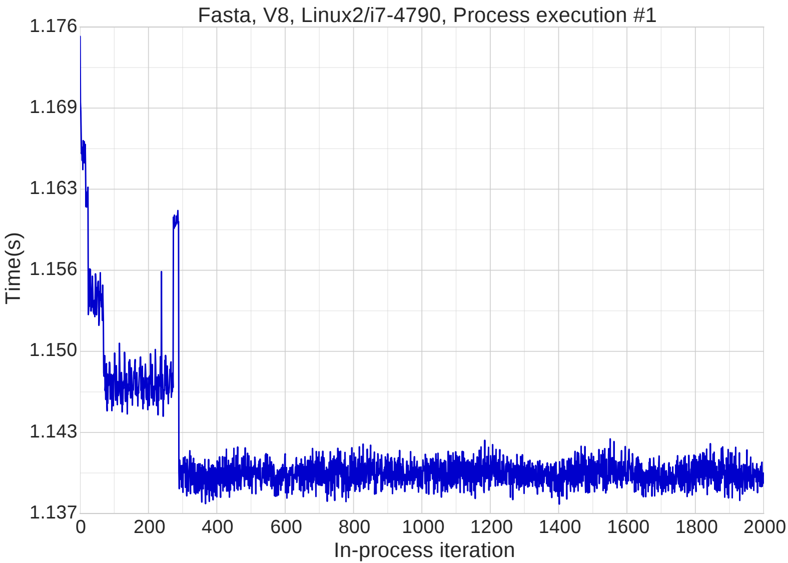
Benchmark runner: tries to control as many confounding variables as possible e.g.:

- Minimises I/O
- Sets fixed heap and stack ulimits
- Drops privileges to a ‘clean’ user account
- Automatically reboots the system prior to each proc. exec
- Checks `dmesg` for changes after each proc. exec
- Checks system is at (roughly) same temperature for each proc. exec

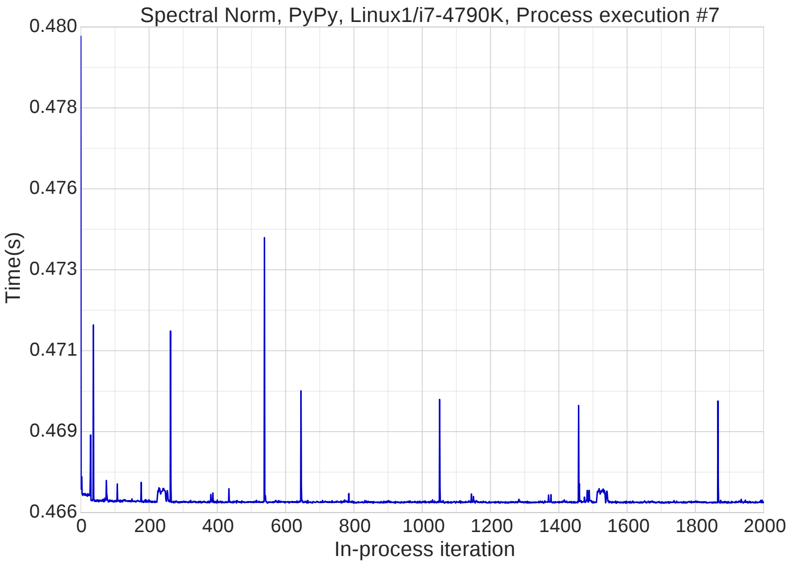
Draft results

Classical Warmup

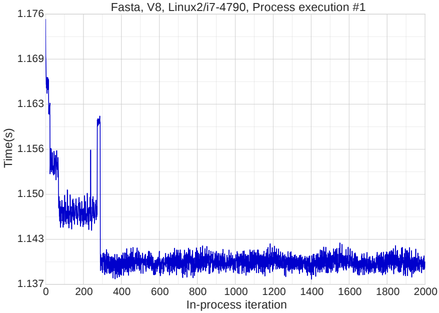
Classical Warmup



Classical Warmup



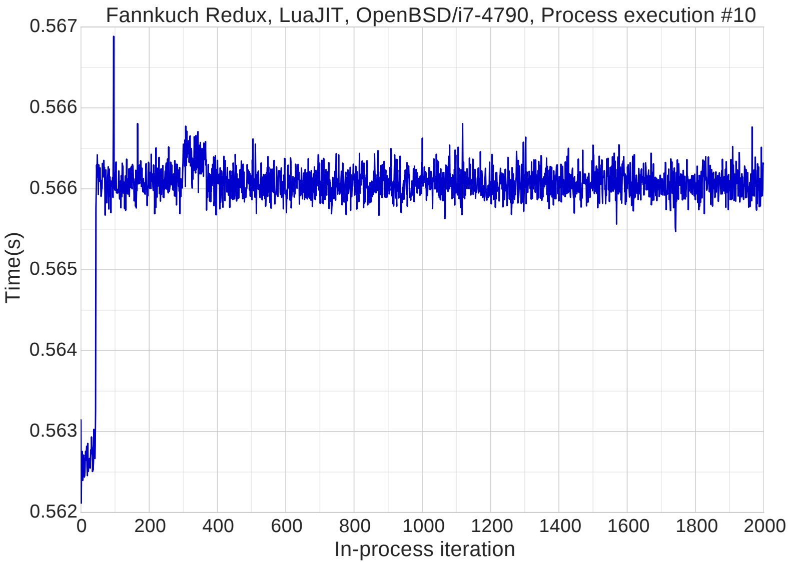
Classical Warmup



(Different machines)

Slowdown

Slowdown

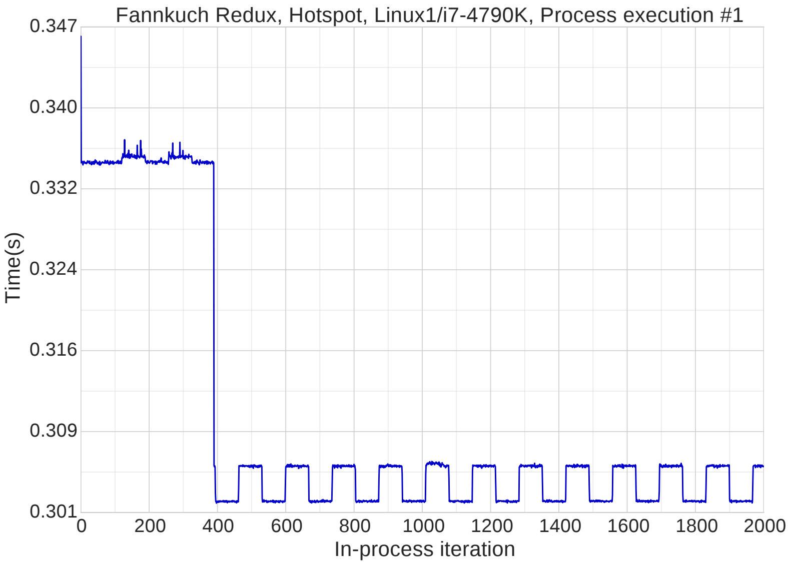


Fannkuch Redux, LuaJIT, OpenBSD/i7-4790, Process execution #10
Time(s)
0.567
0.566
0.566
0.565
0.564
0.563
0.562
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration

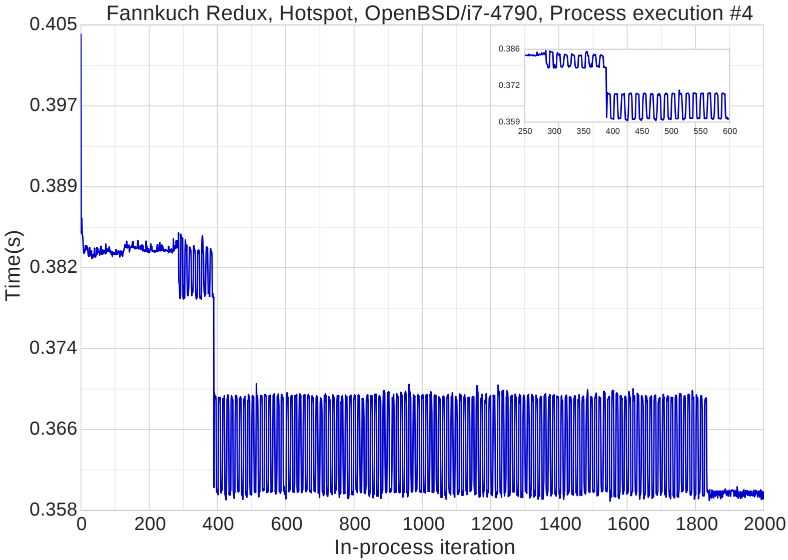
Slowdown

Richards, Hotspot, Linux2/i7-4790, Process execution #2
0.298
0.293
0.287
0.282
0.276
0.271
0.266
Time(s)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration

Cycles

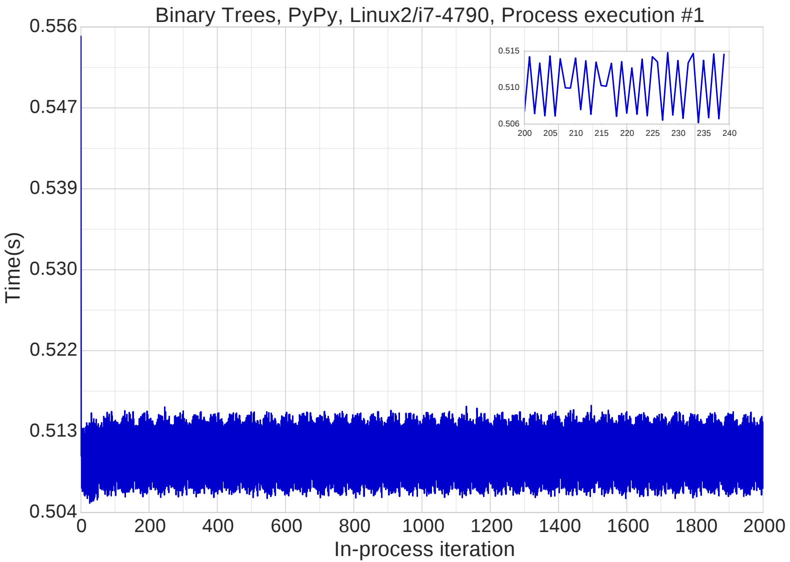

Fannkuch Redux, Hotspot, Linux1/i7-4790K, Process execution #1
Time(s)
0.347
0.340
0.332
0.324
0.316
0.309
0.301
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration

Cycles



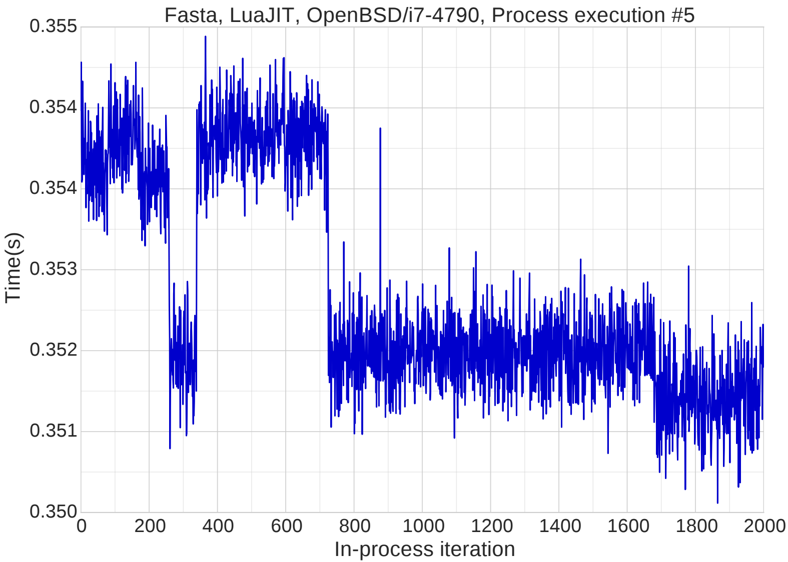
Fannkuch Redux, Hotspot, OpenBSD/i7-4790, Process execution #4
Time(s)
0.405
0.397
0.389
0.382
0.374
0.366
0.358
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration
0.386
0.372
0.359
250
300
350
400
450
500
550
600

Cycles

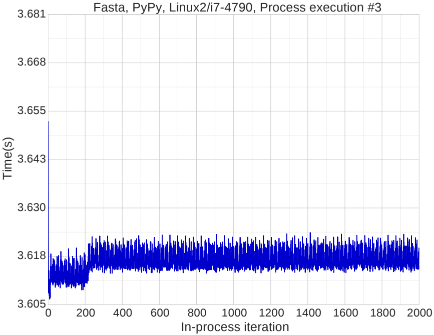


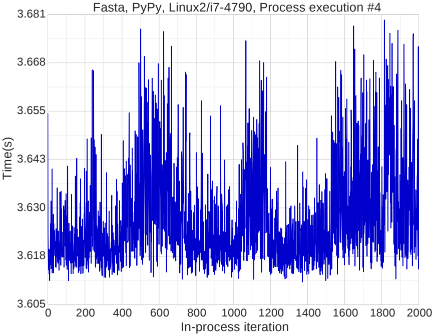
Binary Trees, PyPy, Linux2/i7-4790, Process execution #1
Time(s)
0.556
0.547
0.539
0.530
0.522
0.513
0.504
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration
0.515
0.510
0.506
200
205
210
215
220
225
230
235
240

Never-ending Phase Changes



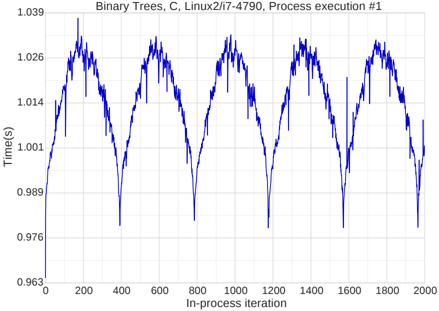
Inconsistent Process-executions





(Note: same machine)

Inconsistent Process-executions



(Note: different machines. Bouncing ball pattern Linux-specific)

Summary

Classical warmup occurs for only:
50% of process executions

Summary

Classical warmup occurs for only:
50% of process executions
25% of (VM, benchmark) pairs

Summary

Classical warmup occurs for only:
50% of process executions
25% of (VM, benchmark) pairs

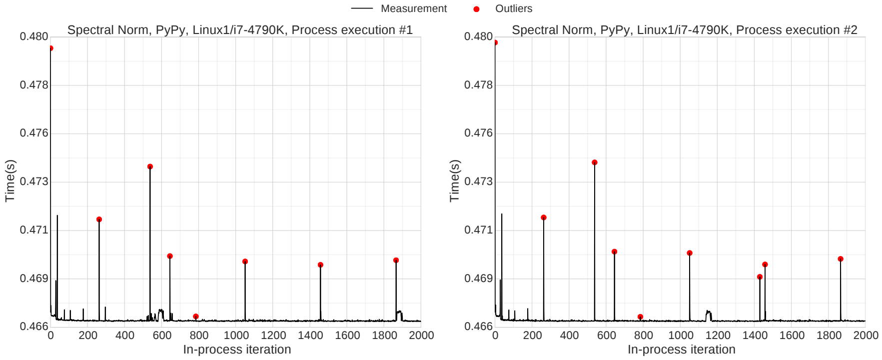
No benchmark warms up
classically on all VMs

Classical warmup occurs for only:
50% of process executions
25% of (VM, benchmark) pairs

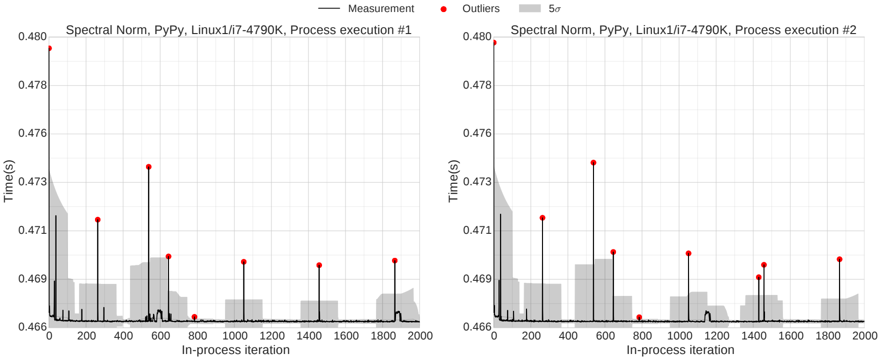
No benchmark warms up
classically on all VMs

How can we measure anything?

Outlier Detection

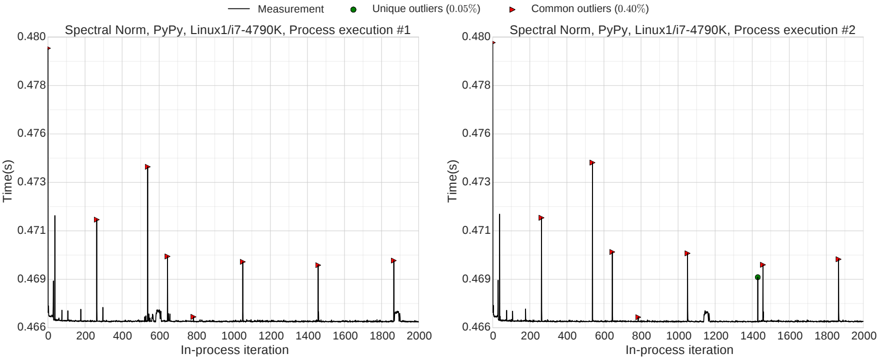


Outlier Detection



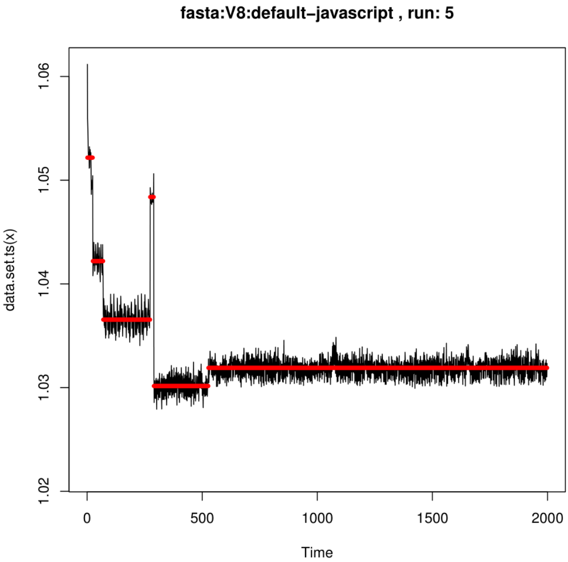
outliers outside 5σ of rolling average

Outlier Detection

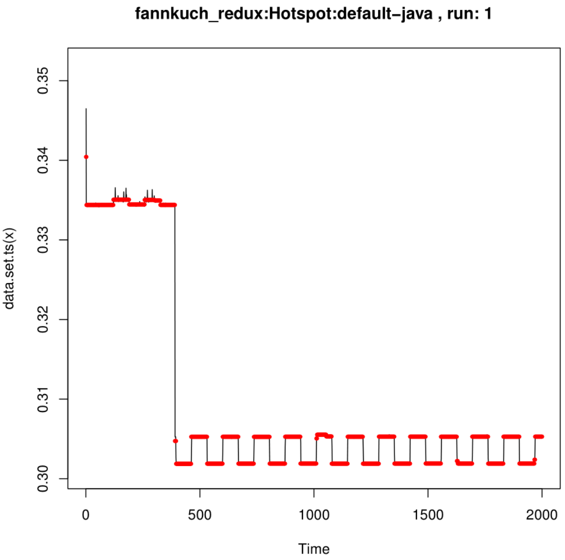


Recurring outliers

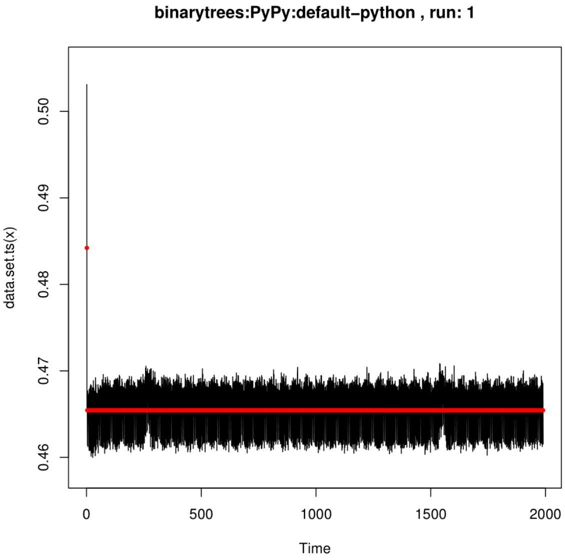
Change-point Analysis



Change-point Analysis



Change-point Analysis



Future Work

Future Work

The ‘obvious’ (control more variables; more benchmarks; more VMs; etc.)

The ‘obvious’ (control more variables; more benchmarks; more VMs; etc.)

Can we work out *why* we see what we see? e.g. is that spike at $x = 78$ actually $\{$GC, compilation, $\ldots\}$

Full Results

https://archive.org/download/softdev_warmup_experiment_artefacts/v0.2/

- all_graphs.pdf All plots in one huge PDF.
- warmup_results*.json.bz2 Raw results.

References

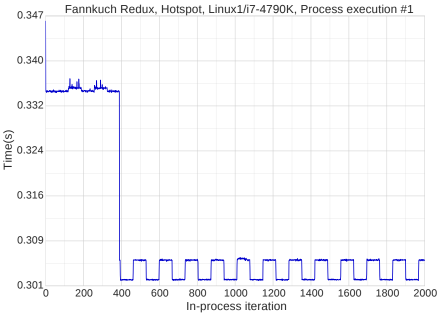
JIT Warmup Blows Hot and Cold
E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount and L. Tratt.

Rigorous Benchmarking in Reasonable Time
T. Kalibera and R. Jones

Specialising Dynamic Techniques for Implementing the Ruby Programming Language
C. Seaton (Chapter 4)

Quantifying performance changes with effect size confidence intervals
T. Kalibera and R. Jones

Thanks for listening



Krun Controls

Linux controls. Krun checks:

- Intel P-state support is disabled in the kernel.
- The `performance` governor is used.
- The kernel is “tickless” (`NO_HZ_FULL_ALL`).
- The `perf` sample rate is lowest possible (1Hz).
- ASLR is disabled.

(Note: Linux ignores ulimits)

Krun Controls

OpenBSD controls. Krun checks:

- Malloc flags ensure malloc does the minimum possible work.
- `apm -H`

On OpenBSD:

- We can't disable ASLR
- We can't disable ticks.
- We can't disable P-states in software.
- There is no kernel profiler (good for us!).