Fine-grained language composition

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

Software Development Team
2016-05-17
Background
A perfect programming language
Solution
Background

Solution

A new programming language
Reality
Reality

Another imperfect programming language
What to expect from this talk
What to expect from this talk

A ∪ B
What to expect from this talk

Python ∪ 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

Python

PHP

PyHyp
Tooling challenges

- Syntax: Python, PHP
- Runtime: PyHyp

- Syntax: Python, PHP
- Runtime: PyHyp
Tooling challenges

Language boxes

PyHyp

Syntax
Runtime

Python
Syntax
Runtime

PHP
Syntax
Runtime

6/24 HTTP://SOFT-DEV.ORG/
Tooling challenges

- Syntax
  - Python
  - Runtime
- Language boxes
- Syntax
  - PyHyp
  - Runtime
- Composed meta-tracing VMs

- Syntax
  - PHP
  - Runtime
Syntax composition

**PL X**
<grammar>
expr ::= ...
term ::= ...  
  | ...  
  | ...  
func ::= ...

**PL Y**
<program>
for (j : js) {
doStuff();
}
.
.
.

7/24 HTTP://SOFT-DEV.ORG/
Syntax composition

Parser

<grammar>
expr ::= ...
term ::= ...
    | ...
    | ...
func ::= ...

<program>
for (j : js) {
    doStuff();
}

7/24 HTTP://SOFT-DEV.ORG/
Syntax composition

**PL X**
<grammar>
expr ::= ...
term ::= ...
    | ...
    | ...
func ::= ...

**PL Y**
<program>
for (j : js) {
    doStuff();
}
.
.
.

Parser
Parse Tree

Parser
Parse Tree

7/24 HTTP://SOFT-DEV.ORG/
Syntax composition

PL X
<grammar>
expr ::= ...
term ::= ...
    | ...
    | ...
func ::= ...

PL Y
<program>
for (j : js) {
doStuff();
}
.
.
.

LR
Parse Tree
Syntax composition

PL X
<grammar>
expr ::= ...
term ::= ...
term ::= ...
term ::= ...
func ::= ...

PL Y
<program>
for (j : js) {
    doStuff();
}
.
.
.

LR
Parse Tree
Undefined

7/24 HTTP://SOFT-DEV.ORG/
Syntax composition

**PL X**
<grammar>

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

**PL Y**
<program>

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

Generalised Parse Tree
Syntax composition

PL X
<grammar>
expr::= ...
term::= ...
    | ...
    | ...
func ::= ...

PL Y
<program>
for (j : js) {
    doStuff();
}
.
.
.
Generalised Parse Tree
Ambiguous

7/24 HTTP://SOFT-DEV.ORG/
Syntax composition

**PL X**
<grammar>
expr ::= ...
term ::= ...
    | ...
    | ...
func ::= ...

**PLY Y**
<program>
for (j : js) {
    doStuff();
}
.
.
.

PEG

Parse Tree
Syntax composition

PL X
<grammar>
expr::= ...
term::= ...
    | ...
    | ...
func ::= ...

PLY
<program>
for (j : js) {
    doStuff();
}
.
.
.
PEG
Parse Tree
Shadows

7/24 HTTP://SOFT-DEV.ORG/
The only choice?
The only choice?

SDE
The challenge

Challenge: SDE’s power + a text editor feel?
Runtime composition
## Composed Richards vs. other VMs

<table>
<thead>
<tr>
<th>Type</th>
<th>VM</th>
<th>Value</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono</td>
<td>CPython 2.7.7</td>
<td>9.475</td>
<td>± 0.0127</td>
</tr>
<tr>
<td></td>
<td>HHVM 3.4.0</td>
<td>4.264</td>
<td>± 0.0386</td>
</tr>
<tr>
<td></td>
<td>HippyVM</td>
<td>0.250</td>
<td>± 0.0008</td>
</tr>
<tr>
<td></td>
<td>PyPy 2.4.0</td>
<td>0.178</td>
<td>± 0.0006</td>
</tr>
<tr>
<td></td>
<td>Zend 5.5.13</td>
<td>9.070</td>
<td>± 0.0361</td>
</tr>
</tbody>
</table>
## Composed Richards vs. other VMs

<table>
<thead>
<tr>
<th>Type</th>
<th>VM</th>
<th>Value</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono</td>
<td>CPython 2.7.7</td>
<td>9.475</td>
<td>± 0.0127</td>
</tr>
<tr>
<td></td>
<td>HHVM 3.4.0</td>
<td>4.264</td>
<td>± 0.0386</td>
</tr>
<tr>
<td>Mono</td>
<td>HippyVM</td>
<td>0.250</td>
<td>± 0.0008</td>
</tr>
<tr>
<td>Mono</td>
<td>PyPy 2.4.0</td>
<td>0.178</td>
<td>± 0.0006</td>
</tr>
<tr>
<td>Mono</td>
<td>Zend 5.5.13</td>
<td>9.070</td>
<td>± 0.0361</td>
</tr>
<tr>
<td>Composed</td>
<td>PyHyp</td>
<td>0.335</td>
<td>± 0.0012</td>
</tr>
</tbody>
</table>
Datatype conversion

Data type conversion diagram:

- **PHPRoot**
  - **PHPObject**
  - **PHPInt**
  - **PHPFunc**
Datatype conversion

PHPRoot

PHPObject  PHPInt  PHPFunc

PyRoot

PyObject  PyInt  PyFunc
Datatype conversion: primitive types

PHP

Python
## Datatype conversion: primitive types

<table>
<thead>
<tr>
<th>PHP</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 : PHPInt</td>
<td></td>
</tr>
</tbody>
</table>

[15/24](http://soft-dev.org/)
Datatype conversion: primitive types

PHP

2 : PHPInt

Python

2 : PyInt
Datatype conversion: user types

PHP

Python
Datatype conversion: user types

PHP

Python

o : PHPObject
Datatype conversion: user types

Diagram:

- **PyRoot**
  - **PyObject**
  - **PyInt**
  - **PyFunc**
Datatype conversion: user types

```
PyRoot
|
- PyObject
- PyInt
- PyFunc

PyPHPAdapter
```
Datatype conversion: user types

```
PyRoot

PyObject    PyInt    PyFunc

PyPHPAdapter
  php_obj : PHPObject
```
Datatype conversion: user types

PHP

```
o : PHPObjecet
```

......

Python
Datatype conversion: user types

PHP

Python

o : PHPObjec

:PyPHPAdapter
Datatype conversion: user types

PHP

Python

\[ o : \text{PHPObject} \quad \text{PyPHPAdapter} \]

\[ \text{php_obj} \]
Datatype conversion: user types

PHP

\[ o : \text{PHPObject} \]

Python

\[ :\text{PyPHPAdapter} \]

\[ \text{php_obj} \]

\[ \text{Immutable field} \]
A good composition needs to reduce friction.
A good composition needs to reduce *friction*. Some examples:
- Lexical scoping (or lack thereof) in PHP and Python (semantic 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)
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)
Unipycation

PyPy
Hippy
Interpreters
Glue

Meta-tracing

PyHyp
Interpreter
Tracing JIT

18/24 HTTP://SOFT-DEV.ORG/
# Absolute timing comparison

<table>
<thead>
<tr>
<th>VM</th>
<th>Benchmark</th>
<th>Python</th>
<th>Prolog</th>
<th>Python → Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>±</td>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>CPython-SW</td>
<td>SmallFunc</td>
<td>0.125s</td>
<td>0.257s</td>
<td>28.893s</td>
</tr>
<tr>
<td></td>
<td>L1A0R</td>
<td>2.924s</td>
<td>7.352s</td>
<td>9.310s</td>
</tr>
<tr>
<td></td>
<td>L1A1R</td>
<td>4.184s</td>
<td>18.890s</td>
<td>20.865s</td>
</tr>
<tr>
<td></td>
<td>NdL1A1R</td>
<td>7.531s</td>
<td>18.643s</td>
<td>667.682s</td>
</tr>
<tr>
<td></td>
<td>TCons</td>
<td>264.415s</td>
<td>48.819s</td>
<td>2185.150s</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>9.374s</td>
<td>25.148s</td>
<td>2207.304s</td>
</tr>
<tr>
<td>Unipyrcation</td>
<td>SmallFunc</td>
<td>0.001s</td>
<td>0.006s</td>
<td>0.001s</td>
</tr>
<tr>
<td></td>
<td>L1A0R</td>
<td>0.085s</td>
<td>0.086s</td>
<td>0.087s</td>
</tr>
<tr>
<td></td>
<td>L1A1R</td>
<td>0.112s</td>
<td>0.114s</td>
<td>0.115s</td>
</tr>
<tr>
<td></td>
<td>NdL1A1R</td>
<td>0.500s</td>
<td>0.548s</td>
<td>2.674s</td>
</tr>
<tr>
<td></td>
<td>TCons</td>
<td>6.053s</td>
<td>2.444s</td>
<td>36.069s</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>0.845s</td>
<td>1.416s</td>
<td>5.056s</td>
</tr>
<tr>
<td>Jython-tuProlog</td>
<td>SmallFunc</td>
<td>0.088s</td>
<td>3.050s</td>
<td>52.294s</td>
</tr>
<tr>
<td></td>
<td>L1A0R</td>
<td>1.078s</td>
<td>206.590s</td>
<td>199.963s</td>
</tr>
<tr>
<td></td>
<td>L1A1R</td>
<td>2.145s</td>
<td>293.311s</td>
<td>294.781s</td>
</tr>
<tr>
<td></td>
<td>NdL1A1R</td>
<td>7.939s</td>
<td>1857.687s</td>
<td>1990.985s</td>
</tr>
<tr>
<td></td>
<td>TCons</td>
<td>543.347s</td>
<td>8014.477s</td>
<td>8202.362s</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>5.661s</td>
<td>6981.873s</td>
<td>5577.322s</td>
</tr>
</tbody>
</table>
## Relative timing comparison

<table>
<thead>
<tr>
<th>VM</th>
<th>Benchmark</th>
<th>Python $\rightarrow$ Prolog</th>
<th>Prolog $\rightarrow$ Python</th>
<th>Unipytcation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPython-SWI</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>231.770×</td>
<td>$\pm$ 13.136</td>
<td>112.567× $\pm$ 1.242</td>
<td>27821.079× $\pm$ 2331.665</td>
</tr>
<tr>
<td>L1A0R</td>
<td>3.184×</td>
<td>$\pm$ 0.300</td>
<td>1.266× $\pm$ 0.014</td>
<td>107.591× $\pm$ 0.995</td>
</tr>
<tr>
<td>L1A1R</td>
<td>4.987×</td>
<td>$\pm$ 0.049</td>
<td>1.105× $\pm$ 0.007</td>
<td>181.899× $\pm$ 0.590</td>
</tr>
<tr>
<td>NdL1A1R</td>
<td>88.654×</td>
<td>$\pm$ 1.368</td>
<td>35.814× $\pm$ 0.554</td>
<td>249.737× $\pm$ 2.922</td>
</tr>
<tr>
<td>TCons</td>
<td>8.264×</td>
<td>$\pm$ 0.101</td>
<td>44.760× $\pm$ 0.453</td>
<td>60.583× $\pm$ 0.637</td>
</tr>
<tr>
<td>Lists</td>
<td>235.459×</td>
<td>$\pm$ 2.314</td>
<td>87.772× $\pm$ 1.017</td>
<td>436.609× $\pm$ 4.415</td>
</tr>
<tr>
<td><strong>Unipytcation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>1.295×</td>
<td>$\pm$ 0.105</td>
<td>0.182× $\pm$ 0.054</td>
<td>1.000×</td>
</tr>
<tr>
<td>L1A0R</td>
<td>1.020×</td>
<td>$\pm$ 0.002</td>
<td>1.012× $\pm$ 0.002</td>
<td>1.000×</td>
</tr>
<tr>
<td>L1A1R</td>
<td>1.025×</td>
<td>$\pm$ 0.002</td>
<td>1.002× $\pm$ 0.003</td>
<td>1.000×</td>
</tr>
<tr>
<td>NdL1A1R</td>
<td>5.349×</td>
<td>$\pm$ 0.045</td>
<td>4.879× $\pm$ 0.924</td>
<td>1.000×</td>
</tr>
<tr>
<td>TCons</td>
<td>5.959×</td>
<td>$\pm$ 0.282</td>
<td>14.756× $\pm$ 0.092</td>
<td>1.000×</td>
</tr>
<tr>
<td>Lists</td>
<td>5.982×</td>
<td>$\pm$ 0.045</td>
<td>3.569× $\pm$ 0.026</td>
<td>1.000×</td>
</tr>
<tr>
<td><strong>Jython-tuProlog</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>592.904×</td>
<td>$\pm$ 19.517</td>
<td>17.143× $\pm$ 0.338</td>
<td>50354.204× $\pm$ 4341.413</td>
</tr>
<tr>
<td>L1A0R</td>
<td>185.460×</td>
<td>$\pm$ 2.818</td>
<td>0.968× $\pm$ 0.021</td>
<td>2310.844× $\pm$ 28.093</td>
</tr>
<tr>
<td>L1A1R</td>
<td>137.427×</td>
<td>$\pm$ 14.537</td>
<td>1.005× $\pm$ 0.028</td>
<td>2569.873× $\pm$ 52.847</td>
</tr>
<tr>
<td>NdL1A1R</td>
<td>250.776×</td>
<td>$\pm$ 14.666</td>
<td>1.072× $\pm$ 0.009</td>
<td>744.699× $\pm$ 6.726</td>
</tr>
<tr>
<td>TCons</td>
<td>15.096×</td>
<td>$\pm$ 0.106</td>
<td>1.023× $\pm$ 0.004</td>
<td>227.409× $\pm$ 1.592</td>
</tr>
<tr>
<td>Lists</td>
<td>985.149×</td>
<td>$\pm$ 8.674</td>
<td>0.799× $\pm$ 0.003</td>
<td>1103.206× $\pm$ 8.338</td>
</tr>
</tbody>
</table>
What can we use this for?
First-class languages
What can we use this for?

First-class languages

Language migration
Thanks to our funders

- EPSRC: COOLER and Lecture.
- Oracle: various.
End of part one...

PHP

```
1: <html>
2: <head>
3: import sqlite3
4: c = sqlite3.connect('/home/ltratt/scratch/demo/test.db')
5: </head>
6: <body>
7: <h1>hello</h1>
8: for n, in SELECT name FROM people:
9:   print n
10: </body>
11: </html>
```

Python

```
RPython
Interpreter
Tracing JIT

Glue
PyHyp
Meta-tracing
```

**Immutable field**

PyPy
Hippy
Interpreters
Glue

PyPy
Interpreters
Tracing JIT
VM Warmup Blows Hot and Cold

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

Software Development Team
2016-05-17
What is ‘warmup’?
What is ‘warmup’?

Idealised VM Warmup

iteration time

in-process iteration
What is ‘warmup’?

Idealised VM Warmup

Profiling Interpreter
What is ‘warmup’?

Idealised VM Warmup

- Compilation
- Profiling Interpreter

iteration time

in-process iteration
What is ‘warmup’?

Idealised VM Warmup

- Compilation
- Profiling Interpreter
- Peak Performance
What is ‘warmup’?

Idealised VM Warmup

iteration time

← warmup →
in-process iteration
More Realistic VM Warmup
More Realistic VM Warmup

iteration time

in-process iteration

some noise
More Realistic VM Warmup

- compilation tiers
- some noise
More Realistic VM Warmup

- Some noise
- Compilation tiers
- GC spikes

Iteration time

In-process iteration
Why care about warmup?
Why care about warmup?

We report: peak performance
We report: peak performance

Users see: warmup + peak performance
The Warmup Experiment
Hypothesis: Small, deterministic programs exhibit classical warmup behaviour
Method 1: Which benchmarks?
The language benchmark games are perfect for us (unusually)
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?
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

BENCHMARKRUNNER: TRIESTO CONTROL AS MANY CONFOUNDING VARIABLES AS POSSIBLE E.G.:
• MINIMISE
• TESTS
• EDHEAP AND STACK LIMITS
• DROP PRIVILEGES TO A 'CLEAN' USER ACCOUNT
• AUTOMATICALLY REBOOT THE SYSTEM PRIOR TO EACH PROC. EXEC
• CHECK dmesg FOR CHANGES AFTER EACH PROC. EXEC
• CHECK SYSTEM IS AT (ROUGHLY) SAME TEMPERATURE FOR EACH PROC. EXEC

10/27 HTTP://SOFT-DEV.ORG/
Method 5: Krun

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

Fasta, V8, Linux2/i7-4790, Process execution #1

In-process iteration

Time(s)

In-process iteration
Classical Warmup

Spectral Norm, PyPy, Linux1/i7-4790K, Process execution #7

In-process iteration vs. Time(s) graph showing performance metrics.
(Different machines)
Slowdown
Slowdown

Fannkuch Redux, LuaJIT, OpenBSD/i7-4790, Process execution #10

In-process iteration

Time(s)
Cycles

Fannkuch Redux, Hotspot, OpenBSD/i7-4790, Process execution #4

In-process iteration

Time(s)

0.358
0.366
0.374
0.382
0.389
0.397
0.405

0 200 400 600 800 1000 1200 1400 1600 1800 2000

0.359
0.372
0.386

250 300 350 400 450 500 550 600

15/27 HTTP://SOFT-DEV.ORG/
Never-ending Phase Changes

Fasta, LuaJIT, OpenBSD/i7-4790, Process execution #5

In-process iteration vs. Time(s)
Inconsistent Process-executions

(Note: same machine)
Inconsistent Process-executions

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

# ClassicalWarmup Occurs For Only:
- 50% of process executions
- 25% of (6-, benchmark) pairs
- Benchmark warms up classically on all 6-S

How Can we Measure Anything?

http://soft-dev.org/
Classical warmup occurs for only:
50% of process executions
Classical warmup occurs for only:
50% of process executions
25% of (VM, benchmark) pairs
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

outliers outside 5σ of rolling average
Outlier Detection

Spectral Norm, PyPy, Linux1/i7-4790K, Process execution #1

Spectral Norm, PyPy, Linux1/i7-4790K, Process execution #2

Recurring outliers
Change-point Analysis

fasta:V8:default–javascript , run: 5
Change-point Analysis

fannkuch_redux:Hotspot:default-java, run: 1

(Time, data_setts(x))
Change-point Analysis

binarytrees:PyPy:default–python, run: 1
Future Work

- Obvious (control more variables, more benchmarks, etc.)
- Why we see what we see? E.g., is that spike at $x = 78$ actually {G#, compilation, ...}
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 \}?
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.
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
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)
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!).