VM Composition with Meta-tracing

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

KING'S College LONDON

Software Development Team

June 3, 2016

Intro

Language Composition

“The ability to write a computer program in a mix of programming languages.”

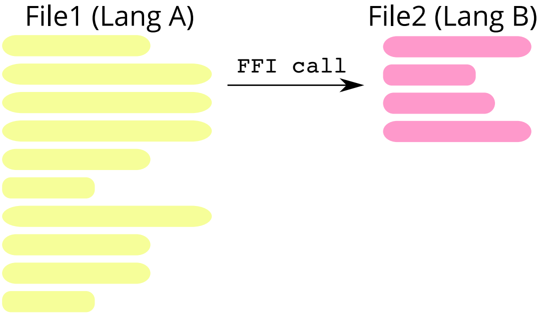
Why Compose Languages?

- Parts of a program are expressed best with different languages.

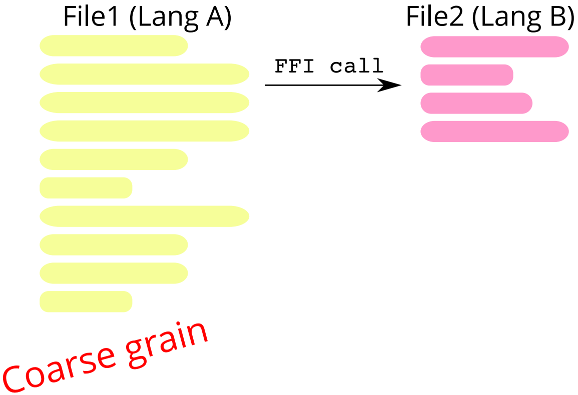
- Access to a broader set of libraries.

- Language migration.

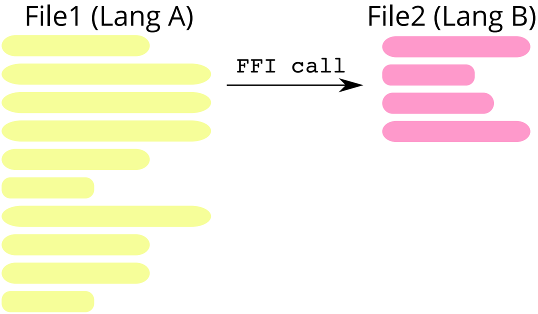
Most languages have a Foreign Function Interface



Most languages have a Foreign Function Interface



Most languages have a Foreign Function Interface

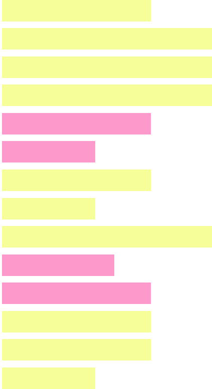


Coarse grain

Limited choice of langs

(second lang is nearly always C)

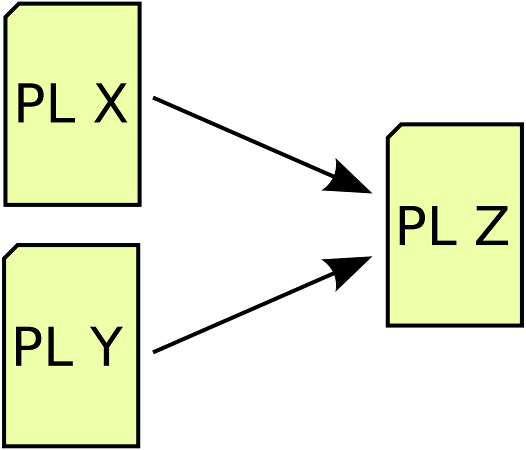
Can we do better?



- Mix languages in the same file
- Mix {methods, functions, expressions}
- Integrate scoping
- Arbitrary languages
- . . .

Approach

Breaking it Down

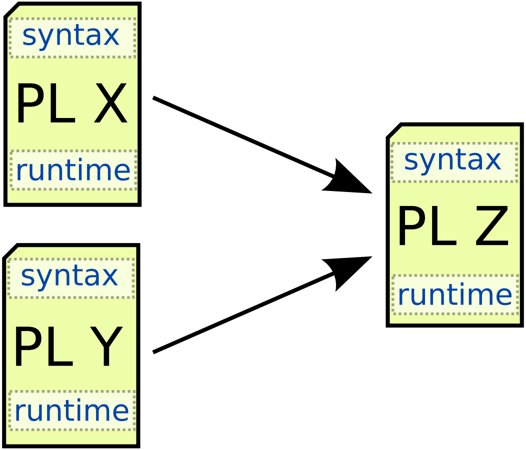


PL X

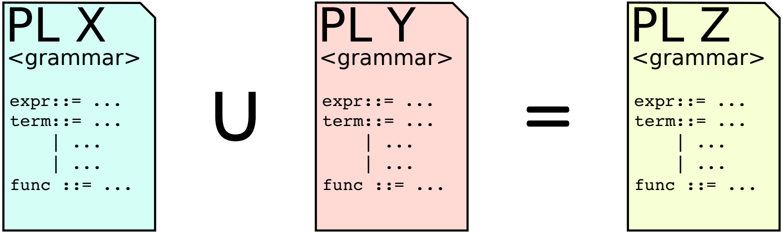
PL Y

PL Z

Breaking it Down



Composing Syntax



PL X

<grammar>

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

∪

PL Y

<grammar>

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

=

PL Z

<grammar>

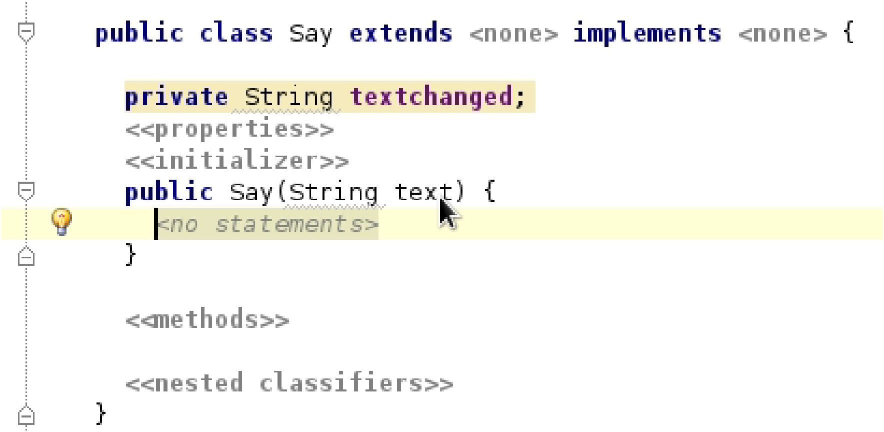
```
expr::= ...
term::= ...
    | ...
    | ...
func ::= ...
```

Easy?

Composing Syntax

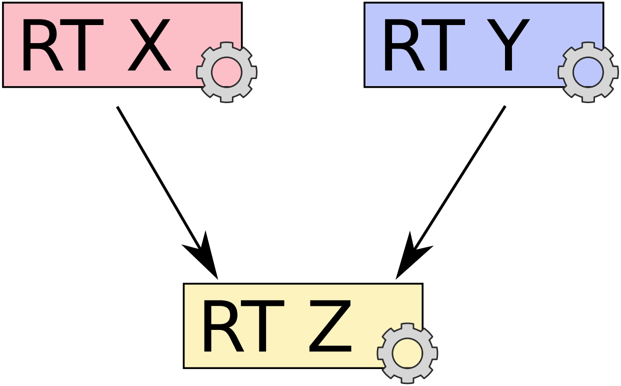
- LR → Possibly undefined.
- PEG → Shadows.
- GLR → Ambiguous.

Syntax Directed Editing



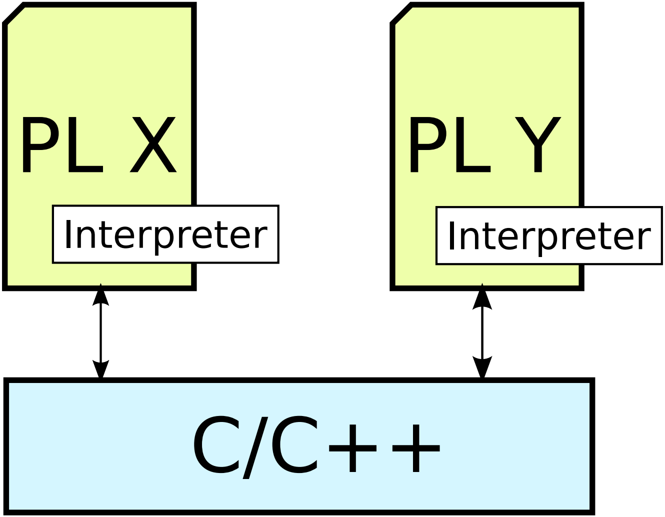
```
public class Say extends <none> implements <none> {

  private String textchanged;
  <<properties>>
  <<initializer>>
  public Say(String text) {
    <no statements>
  }

  <<methods>>

  <<nested classifiers>>
}
```

Composing Runtimes

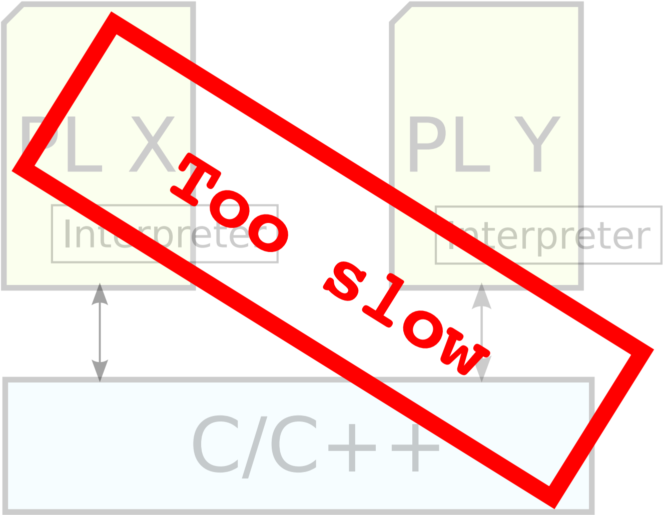


Easy?

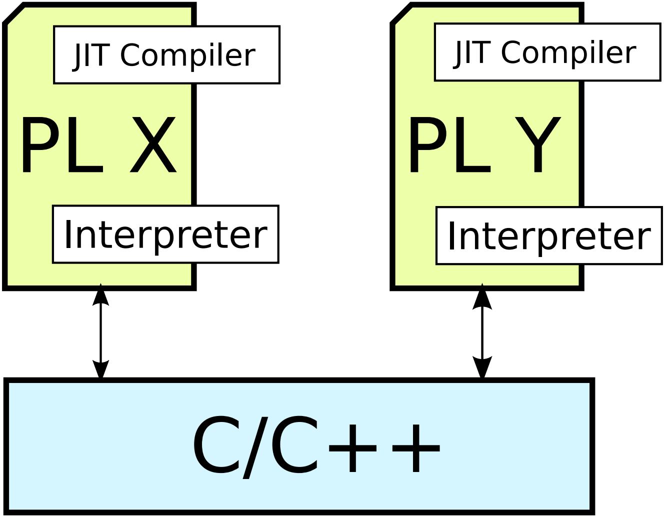
Runtime composition



Runtime composition



Runtime composition


JIT Compiler

PL X

Interpreter

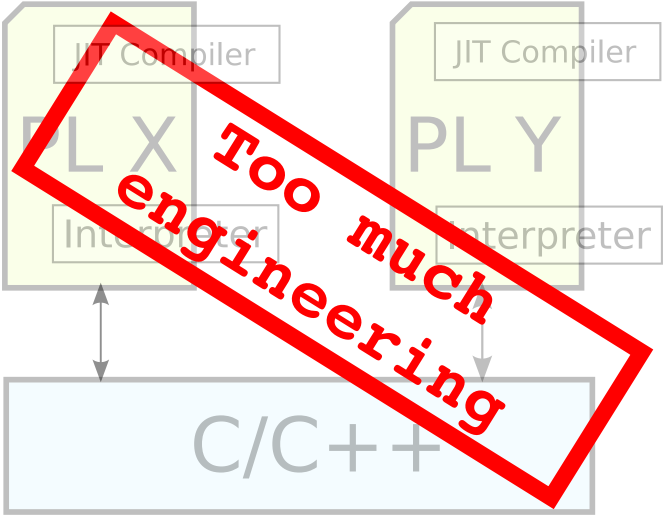
JIT Compiler

PL Y

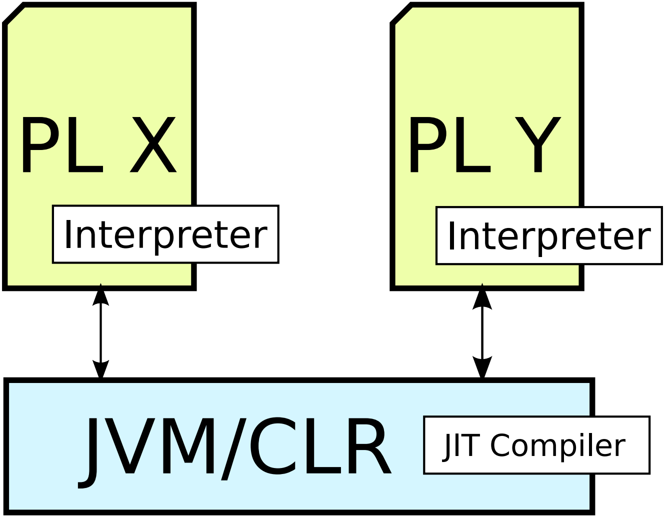
Interpreter

C/C++

Runtime composition



Runtime composition



PL X

Interpreter

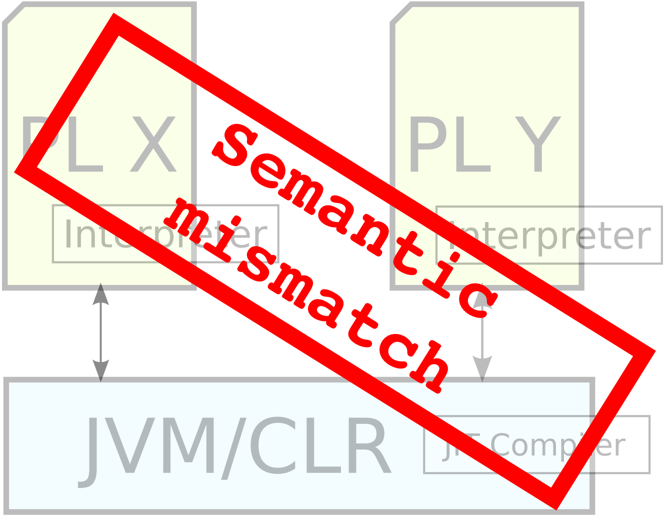
PL Y

Interpreter

JVM/CLR

JIT Compiler

Runtime composition



Our Approach

Summary:

We need a practical way of composing syntax and runtimes.

Our Approach

Summary:

We need a practical way of composing syntax and runtimes.

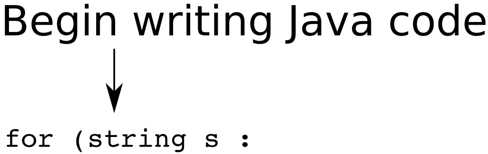
↓

Language Boxes + Meta-tracing

Language Boxes

- Borrows ideas from SDE.
- Palatable editing experience.
- Simple and practical way to compose grammars.

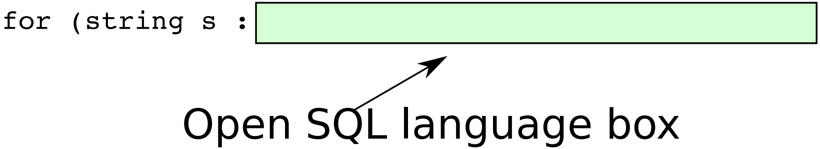
Language Boxes: E.g. Java + SQL



Begin writing Java code

```
for (string s :
```

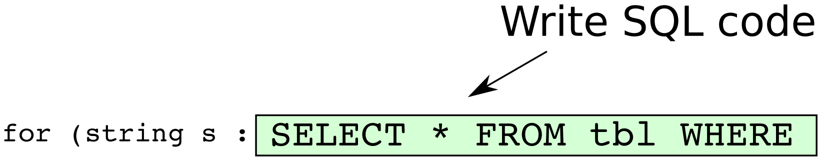
Language Boxes: E.g. Java + SQL



```
for (string s :
```

Open SQL language box

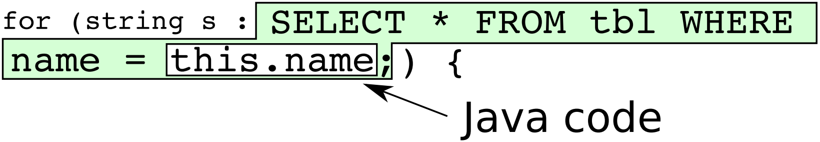
Language Boxes: E.g. Java + SQL



Write SQL code

```
for (string s : SELECT * FROM tbl WHERE
```

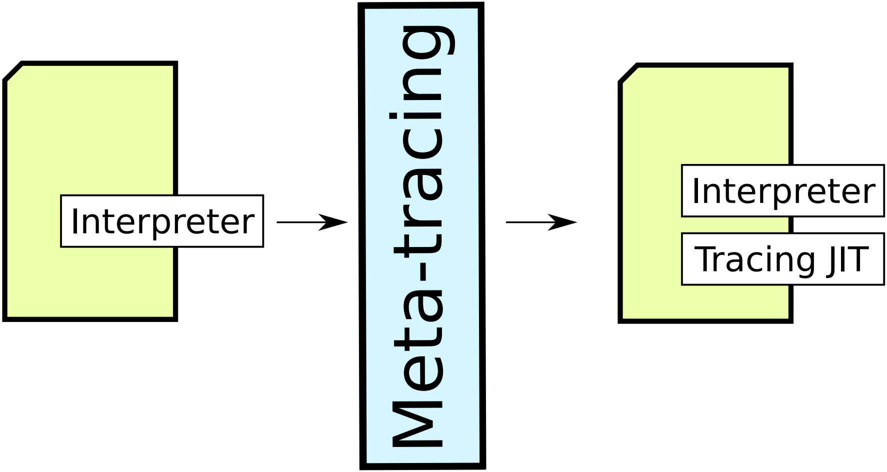
Language Boxes: E.g. Java + SQL



```
for (string s : SELECT * FROM tbl WHERE
name = this.name;) {
```

Java code

Meta-tracing



Interpreters are Loops

- Tell a meta-tracer about the interpreter loop.
- Generate a tracing JIT.
- Trace the interpreter itself, not the user program.

Adding a JIT with Meta-tracing

```
...
pc := 0
while 1:

    instr := load_next_instruction(pc)
    if instr == POP:
        stack.pop()
        pc += 1
    elif instr == BRANCH:
        off = load_branch_jump(pc)
        pc += off
    elif ...:
        ...
```

Adding a JIT with Meta-tracing

```
...
pc := 0
while 1:
    jit_merge_point(pc)
    instr := load_next_instruction(pc)
    if instr == POP:
        stack.pop()
        pc += 1
    elif instr == BRANCH:
        off = load_branch_jump(pc)
        pc += off
    elif ...:
        ...
```

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

```
elif instr == INSTR_IF:
  result = stack.pop()
  if result == True:
    program_counter += 1
  else:
    program_counter +=
      read_jump_if_instruction()
elif instr == INSTR_ADD:
  lhs = stack.pop()
  rhs = stack.pop()
  if isinstance(lhs, int)
   and isinstance(rhs, int):
    stack.push(lhs + rhs)
  else: ...
  program_counter += 1
```

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

User program (lang *FL*)

```
assume x == 6
if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3
```

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

Initial trace

```
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
...
```

Initial trace in full

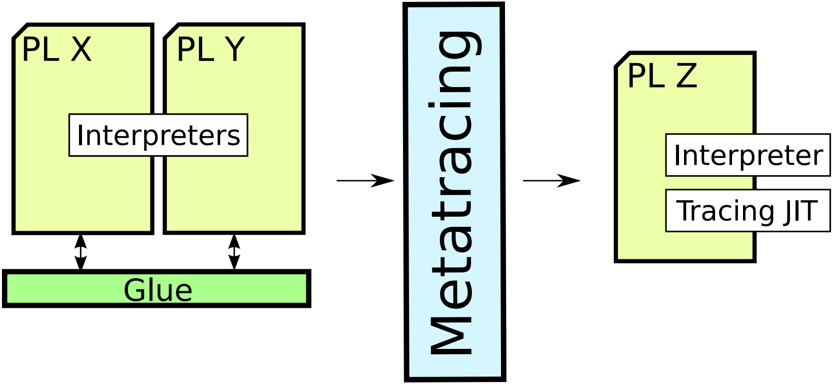
```
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
v14 = add(v11, 2)
v15 = load_instruction(v14)
guard_eq(v15, INSTR_VAR_GET)
v16 = dict_get(v2, "x")
list_append(v1, v16)
v17 = add(v14, 1)
v18 = load_instruction(v17)
guard_eq(v18, INSTR_INT)
list_append(v1, 2)
v19 = add(v17, 1)
v20 = load_instruction(v19)
guard_eq(v20, INSTR_ADD)
v21 = list_pop(v1)
v22 = list_pop(v1)
guard_type(v21, int)
guard_type(v22, int)
v23 = add(v22, v21)
list_append(v1, v23)
v24 = add(v19, 1)
v25 = load_instruction(v24)
guard_eq(v25, INSTR_VAR_SET)
v26 = list_pop(v1)
dict_set(v2, "x", v26)
v27 = add(v24, 1)
v28 = load_instruction(v27)
guard_eq(v28, INSTR_VAR_GET)
v29 = dict_get(v2, "x")
list_append(v1, v29)
v30 = add(v27, 1)
v31 = load_instruction(v30)
guard_eq(v31, INSTR_INT)
list_append(v1, 3)
v32 = add(v30, 1)
v33 = load_instruction(v32)
guard_eq(v33, INSTR_ADD)
v34 = list_pop(v1)
v35 = list_pop(v1)
guard_type(v34, int)
guard_type(v35, int)
v36 = add(v35, v34)
list_append(v1, v36)
v37 = add(v32, 1)
v38 = load_instruction(v37)
guard_eq(v38, INSTR_VAR_SET)
v39 = list_pop(v1)
dict_set(v2, "x", v39)
v40 = add(v37, 1)
```

Optimisation

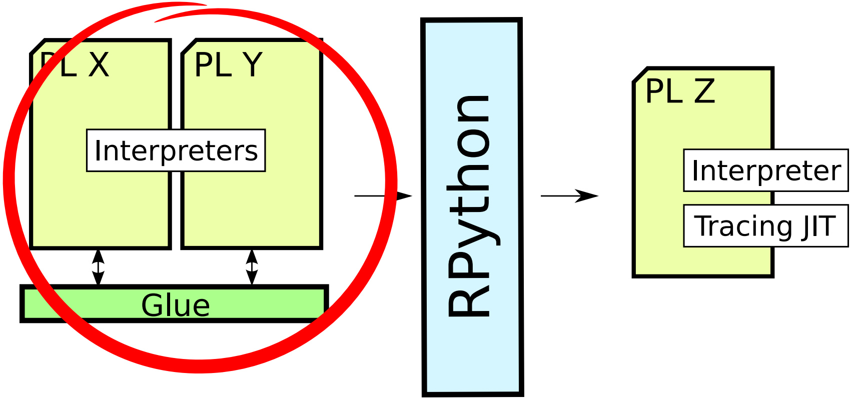
Optimised Trace

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 5)
dict_set(v2, "x", v23)
```

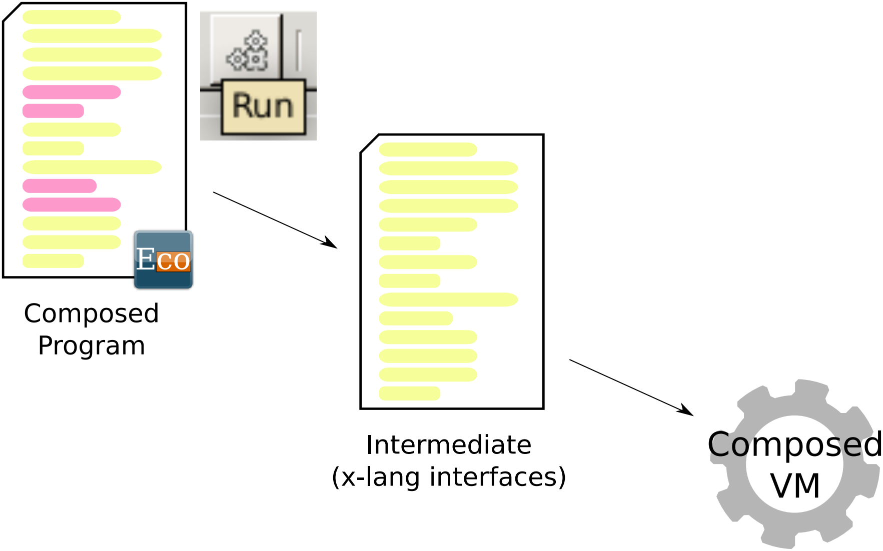
How Does this Apply to VM Composition?



How Does this Apply to VM Composition?



Putting it All Together



Composed Program

Intermediate (x-lang interfaces)

Composed VM

Summarising our Approach

- Editing with Language boxes.
  - Practical syntactic composition.
  - Traditional “code editor” look and feel.
- Interpreter Composition with Meta-tracing
  - Relatively little engineering effort.
  - Language agnostic JIT optimisations.
- Glue together editor and VM with intermediate representation.

Our Compositions

Our Language Compositions

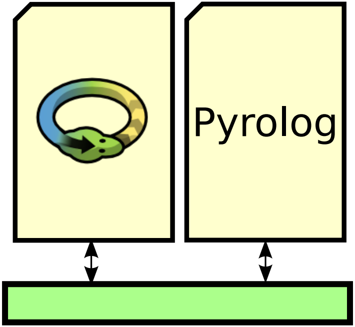
- Unipycation: Python + Prolog
- PyHyp: PHP + Python
- SQPyte: Python + SQLite

Our Language Compositions

- Unipycation: Python + Prolog
- PyHyp: PHP + Python
- SQPyte: Python + SQLite

Unipycation

Python + Prolog



Unipycation

- Fairly coarse “proof of concept” composition.
- Shows that we can glue together meta-tracing interpreters.
- Idiomatic interoperability between Python and Prolog.

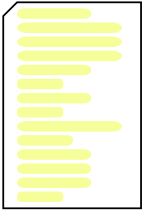
Unipycation Example

```
from uni import Engine

engine = Engine("""
edge(a, c). edge(c, b). edge(c, d). edge(d, e).
edge(b, e). edge(c, f). edge(f, g). edge(e, g).
edge(g, b).

path(From, To, MaxLen, Nodes) :-
    path(From, To, MaxLen, Nodes, 1).
... """)

paths = engine.db.path.iter

for (to, nodes) in paths("b", None, 4, None):
    print("To %s via %s" % (to, nodes))
```

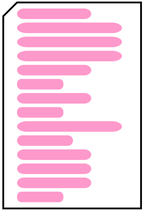
Outcomes of Unipycation

- Proved the concept
  - We wrote some fairly complex composed programs.
- Relatively little engineering effort.
- Gave fairly good performance.
  - Cross-language tracing works.

Unipycation Performance

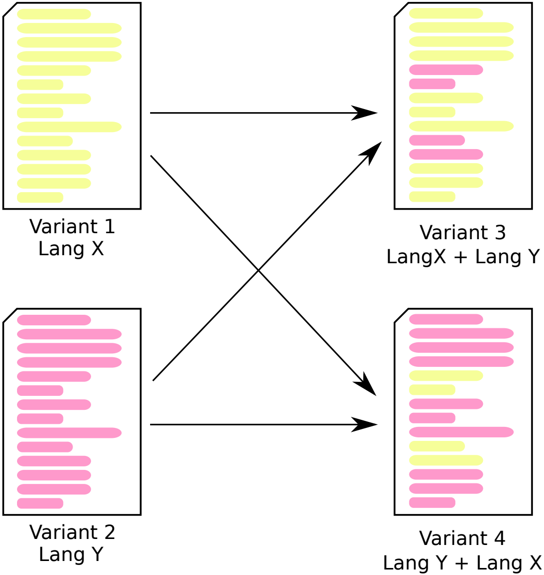


Variant 1
Lang X



Variant 2
Lang Y

Unipycation Performance



Unipycation Performance

Benchmark	$\frac{\mathit{Python}\rightarrow\mathit{Prolog}}{\mathit{Python}}$		$\frac{\mathit{Python}\rightarrow\mathit{Prolog}}{\mathit{Prolog}}$		$\frac{\mathit{Python}\rightarrow\mathit{Prolog}}{\text{Unipycation}}$
SmallFunc	1.276×	±0.081	0.201×	±0.038	1.000×
L1A0R	1.005×	±0.053	0.957×	±0.057	1.000×
L1A1R	1.072×	±0.071	1.034×	±0.069	1.000×
NdL1A1R	5.902×	±0.044	5.635×	±0.216	1.000×
TCons	6.073×	±0.106	13.471×	±0.208	1.000×
Lists	5.969×	±0.025	3.335×	±0.010	1.000×

Benchmark	$\frac{\mathit{Python}\rightarrow\mathit{Prolog}}{\mathit{Prolog}}$		$\frac{\mathit{Python}\rightarrow\mathit{Prolog}}{\text{Unipycation}}$
sat-models	1.462×	±0.021	1.000×
tube	1.014×	±0.019	1.000×
connect4	1.431×	±0.014	1.000×

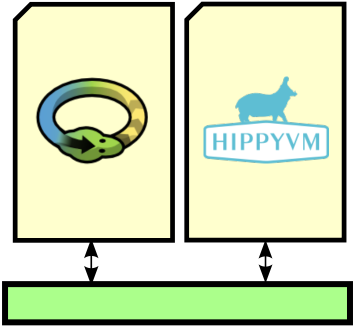
Unipycation Limitations

- Little syntactic integration.
  - Eco served little more than a syntax checker.

- Rudimentary type conversions.
  - Python objects opaque to Prolog.

PyHyp

PHP + Python



Goals of PyHyp

How far can we push language composition?

- Syntactic interoperability harnessing Eco.
- Less opaque type conversions.
- Performance <2–3× over mono-language programs.

Features of PyHyp

- Calling Python functions and methods from PHP
- Calling PHP functions and methods from Python
- Transparent type conversions
- Arbitrary nesting of foreign functions
- Python expressions in PHP
- “Embedding” Python methods inside PHP classes
- Adds support for references to Python
- Cross-language scoping
- Cross-language exceptions

PyHyp Demo

PyHyp Performance

Benchmark	HippyVM	PyHyp$_{\mathrm{PHP}}$	PyHyp$_{\mathrm{Py}}$	PyPy
instchain	0.912 ±0.0011	1.000		0.675 ±0.0007
l1a0r	1.368 ±0.0004	1.000	1.360 ±0.0003	1.340 ±0.0106
l1a1r	1.306 ±0.0017	1.000	1.303 ±0.0016	1.140 ±0.0022
lists	0.975 ±0.0020	1.000	0.560 ±0.0012	0.497 ±0.0010
ref_swap	1.000 ±0.0002	1.000	0.700 ±0.0001	
return_simple	1.000 ±0.0001	1.000	0.778 ±0.0001	0.889 ±0.0001
scopes	4.511 ±0.0025	1.000	0.929 ±0.0005	1.000 ±0.0001
smallfunc	1.000 ±0.0001	1.000	0.750 ±0.0000	1.000 ±0.0001
sum	0.999 ±0.0001	1.000	0.750 ±0.0001	0.874 ±0.0001
sum_meth	0.999 ±0.0001	1.000		0.874 ±0.0002
sum_meth_attr	0.999 ±0.0061	1.000		0.904 ±0.0057

. . .

PyHyp Performance (contd.)

Benchmark	HippyVM	PyHyp$_{\text{PHP}}$	PyHyp$_{\text{Py}}$	PyPy
. . .				
total_list	0.864 ±0.0002	1.000	1.508 ±0.0004	0.587 ±0.0003
walk_list	0.779 ±0.0011	1.000	1.601 ±0.0026	1.080 ±0.0015
deltablue	4.325 ±0.0212	1.000		0.457 ±0.0026
fannkuch	1.848 ±0.0007	1.000	1.891 ±0.0005	1.005 ±0.0004
mandel	0.921 ±0.0005	1.000	0.999 ±0.0003	
richards	0.853 ±0.0010	1.000		0.488 ±0.0005
Geometric Mean	1.222 ±0.0006	1.000	0.963 ±0.0003	0.813 ±0.0007

Worst case: 2.6x overhead

Semantic Friction

Semantic Friction

Implementing desired behaviour: relatively easy

Deciding the correct behaviours: hard

“Semantic friction”

Compromises sometimes must be made.

Semantic Friction: References

```
function swap(&$a, &$b) {
    $temp = $a;
    $a = $b;
    $b = $temp;
}

...

$x = 1; $y = 2;
swap($x, $y);
echo "$x $y";
```

Pure PHP – Prints "2 1"

Semantic Friction: References

```
@php_decor(refs=(0, 1))
def swap(a, b):
    temp = a.deref()
    a.store(b.deref())
    b.store(temp)


...

$x = 1; $y = 2;
swap($x, $y);
echo "$x $y";
```

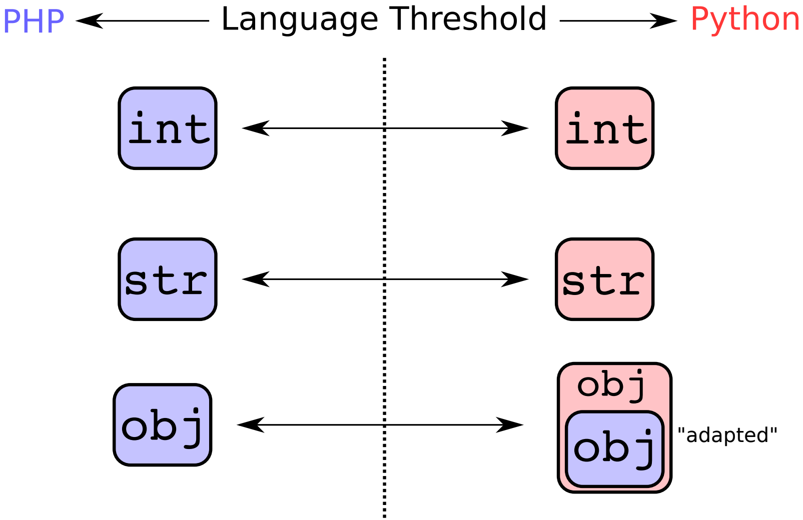
Callee in Python

Semantic Friction: References

```
function swap(&$a, &$b) {
    $temp = $a;
    $a = $b;
    $b = $temp;
}

...

x = PHPRef(1); y = PHPRef(2)
swap(x, y);
print("%s %s" % (x.deref(), y.deref()))
```

Caller in Python

Semantic Friction: Array/Dict/List Conversions



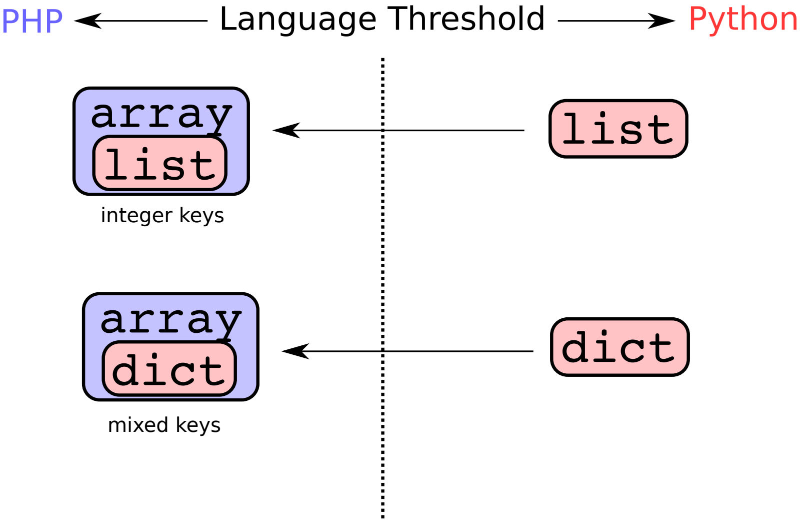
Semantic Friction: Array/Dict/List Conversions

	PHP	Python
Sequence type	`array`	`list`
Mapping type	`array`	`dict`

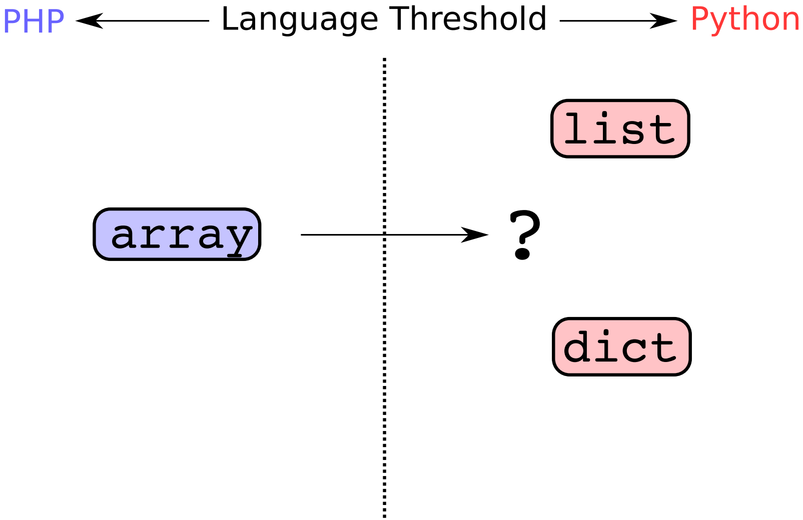
Semantic Friction: Array/Dict/List Conversions

```
php > $a = ["this", "is", "a", "list"];
php > print_r($a);
Array
(
    [0] => this
    [1] => is
    [2] => a
    [3] => list
)
```

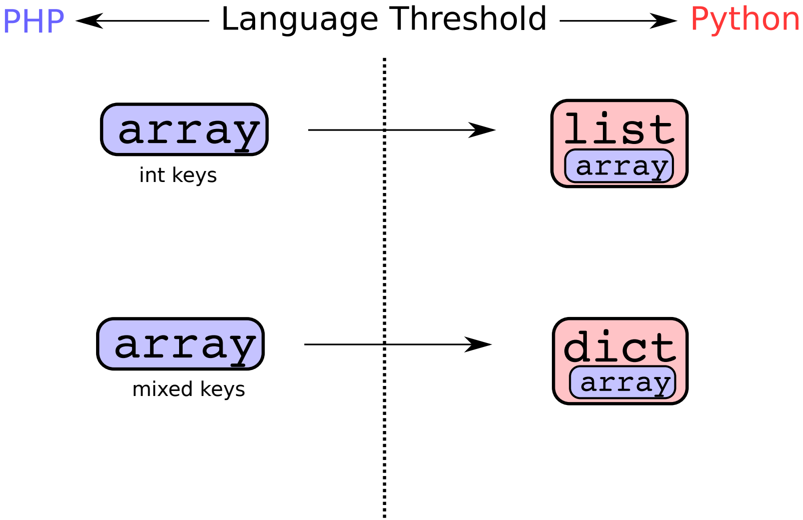
Semantic Friction: Array/Dict/List Conversions



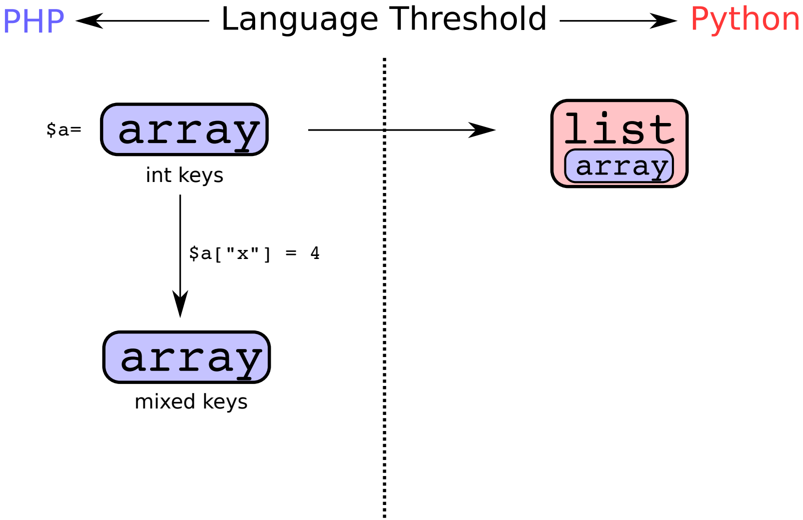
Semantic Friction: Array/Dict/List Conversions



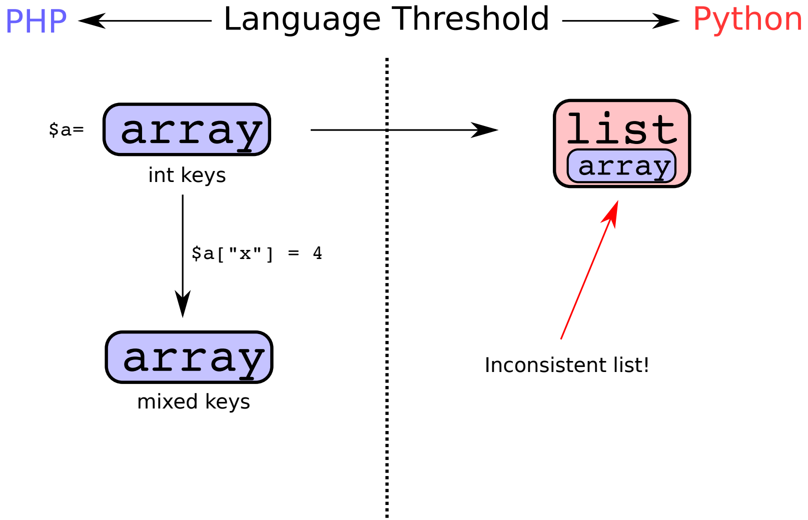
Semantic Friction: Array/Dict/List Conversions



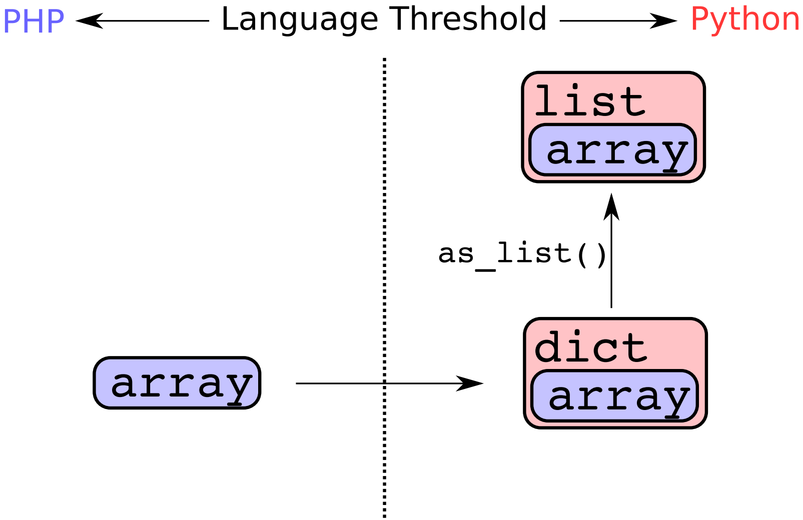
Semantic Friction: Array/Dict/List Conversions



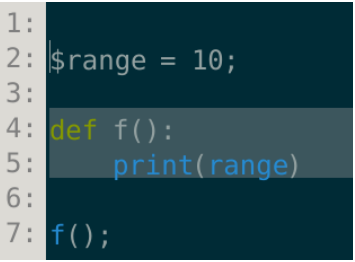
Semantic Friction: Array/Dict/List Conversions



Semantic Friction: Array/Dict/List Conversions

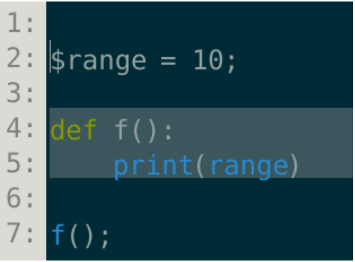


Semantic Friction: Cross Language Scoping



```

$range = 10;

def f():
    print(range)

f();
```

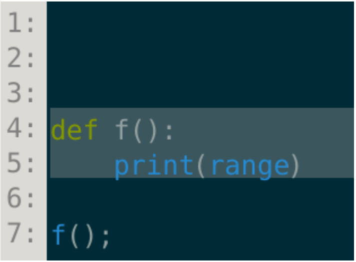
Semantic Friction: Cross Language Scoping



```

$range = 10;

def f():
    print(range)

f();
```

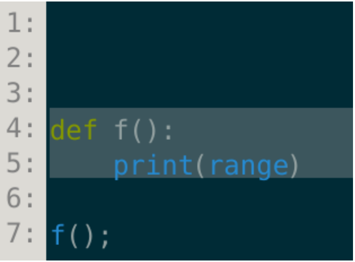
Should print: `10`

Semantic Friction: Cross Language Scoping



```



def f():
    print(range)

f();
```

Semantic Friction: Cross Language Scoping



```



def f():
    print(range)

f();
```

Should print: `<built-in function range>`

Semantic Friction: Cross Language Scoping

If a variable is not bound it the current box:

1. Search boxes outwards starting with the parent box.
2. Look in the "global" namespace of the current language.
3. Look in the "global" namespace of the other language.

"Globals"

- Python: {builtins}
- PHP: {functions, classes}

Conclusions

Conclusions

- Language boxes:
  - Practical syntax composition.
  - Decent editor experience.
- Meta-tracing:
  - Compositions with relatively little effort.
  - Good performance.
- Implementing x-lang behaviours is easy.
- Designing x-lang behaviours is hard.
  - Semantic friction.

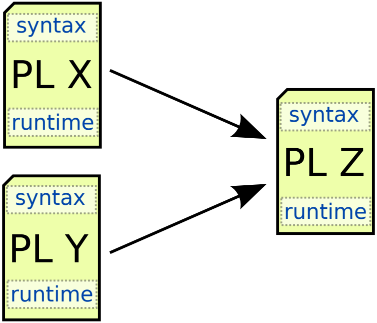
Future Work

- Tools for composed programs
  - Debugging
  - Profiling
  - Version control
  - ...
- Statically typed/functional languages.
- Compositions with $>$2 languages involved.

References

- *Parsing Composed Grammars with Language Boxes* Lukas Diekmann, Laurence Tratt.
- *Eco: A Language Composition Editor* Lukas Diekmann, Laurence Tratt.
- *Unipycation: A Case Study in Cross-language Tracing*, Edd Barrett, Carl Friedrich Bolz and Laurence Tratt
- *Approaches to Interpreter Composition*, Edd Barrett, Carl Friedrich Bolz and Laurence Tratt
- *Fine-grained Language Composition: A Case Study*, Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, Laurence Tratt
- *Making an Embedded DBMS JIT-friendly*, Carl Friedrich Bolz, Darya Kurilova, Laurence Tratt

Thanks



Language Boxes + Meta-tracing

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
  elif instr == INSTR_IF:
    result = stack.pop()
    if result == True:
      program_counter += 1
    else:
      program_counter +=
        read_jump_if_instruction()
  elif instr == INSTR_ADD:
    lhs = stack.pop()
    rhs = stack.pop()
    if isinstance(lhs, int)
     and isinstance(rhs, int):
      stack.push(lhs + rhs)
    else: ...
    program_counter += 1
```

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

Meta-tracing JITs

FL Interpreter	User program (lang *FL*)

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

```
assume x == 6
if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3
```

Meta-tracing JITs

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

Initial trace

```
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
...
```

Meta-tracing JITs

Initial trace in full

```
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
v14 = add(v11, 2)
v15 = load_instruction(v14)
guard_eq(v15, INSTR_VAR_GET)
v16 = dict_get(v2, "x")
list_append(v1, v16)
v17 = add(v14, 1)
v18 = load_instruction(v17)
guard_eq(v18, INSTR_INT)
list_append(v1, 2)
v19 = add(v17, 1)
v20 = load_instruction(v19)
guard_eq(v20, INSTR_ADD)
v21 = list_pop(v1)
v22 = list_pop(v1)
guard_type(v21, int)
guard_type(v22, int)
v23 = add(v22, v21)
list_append(v1, v23)
v24 = add(v19, 1)
v25 = load_instruction(v24)
guard_eq(v25, INSTR_VAR_SET)
v26 = list_pop(v1)
dict_set(v2, "x", v26)
v27 = add(v24, 1)
v28 = load_instruction(v27)
guard_eq(v28, INSTR_VAR_GET)
v29 = dict_get(v2, "x")
list_append(v1, v29)
v30 = add(v27, 1)
v31 = load_instruction(v30)
guard_eq(v31, INSTR_INT)
list_append(v1, 3)
v32 = add(v30, 1)
v33 = load_instruction(v32)
guard_eq(v33, INSTR_ADD)
v34 = list_pop(v1)
v35 = list_pop(v1)
guard_type(v34, int)
guard_type(v35, int)
v36 = add(v35, v34)
list_append(v1, v36)
v37 = add(v32, 1)
v38 = load_instruction(v37)
guard_eq(v38, INSTR_VAR_SET)
v39 = list_pop(v1)
dict_set(v2, "x", v39)
v40 = add(v37, 1)
```

Trace optimisation (1)

Removing constants (from `jit_merge_point`)

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
list_append(v1, v4)
list_append(v1, 0)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v13 = list_pop(v1)
guard_false(v13)
v16 = dict_get(v2, "x")
list_append(v1, v16)
list_append(v1, 2)
v21 = list_pop(v1)
v22 = list_pop(v1)
guard_type(v21, int)
guard_type(v22, int)
v23 = add(v22, v21)
list_append(v1, v23)
v26 = list_pop(v1)
dict_set(v2, "x", v26)
v29 = dict_get(v2, "x")
list_append(v1, v29)
list_append(v1, 3)
v34 = list_pop(v1)
v35 = list_pop(v1)
guard_type(v34, int)
guard_type(v35, int)
v36 = add(v35, v34)
list_append(v1, v36)
v39 = list_pop(v1)
dict_set(v2, "x", v39)
```

Optimisation #2 & #3

List folded trace

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v16 = dict_get(v2, "x")
guard_type(v16, int)
v23 = add(v16, 2)
dict_set(v2, "x", v23)
v29 = dict_get(v2, "x")
guard_type(v29, int)
v36 = add(v29, 3)
dict_set(v2, "x", v36)
```

Optimisation #2 & #3

List folded trace

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v16 = dict_get(v2, "x")
guard_type(v16, int)
v23 = add(v16, 2)
dict_set(v2, "x", v23)
v29 = dict_get(v2, "x")
guard_type(v29, int)
v36 = add(v29, 3)
dict_set(v2, "x", v36)
```

Dict folded trace

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 2)
guard_type(v23, int)
v36 = add(v23, 3)
dict_set(v2, "x", v36)
```

Optimisation #4 & #5

Type folded trace

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 2)
v36 = add(v23, 3)
dict_set(v2, "x", v36)
```

Optimisation #4 & #5

Type folded trace	Arithmetic folded trace

v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 2)
v36 = add(v23, 3)
dict_set(v2, "x", v36)
	v1 = <stack>
	v2 = <vars>
	v4 = dict_get(v2, "x")
	guard_type(v4, int)
	guard_not_less_than(v4, 0)
	v23 = add(v4, 5)
	dict_set(v2, "x", v23)

Optimisation #4 & #5

Type folded trace	Arithmetic folded trace

v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 2)
v36 = add(v23, 3)
dict_set(v2, "x", v36)
	v1 = <stack>
	v2 = <vars>
	v4 = dict_get(v2, "x")
	guard_type(v4, int)
	guard_not_less_than(v4, 0)
	v23 = add(v4, 5)
	dict_set(v2, "x", v23)

Trace optimisation: from 72 trace elements to 7.