Fine-grained Language Composition

Edd Barrett

Lukas Diekmann

Laurence Tratt

KING'S College LONDON

Software Development Team

May 29, 2015

Roadmap

- Language composition background.
- Challenges.
- Our approach.
- Concrete Example: PHP + Python.

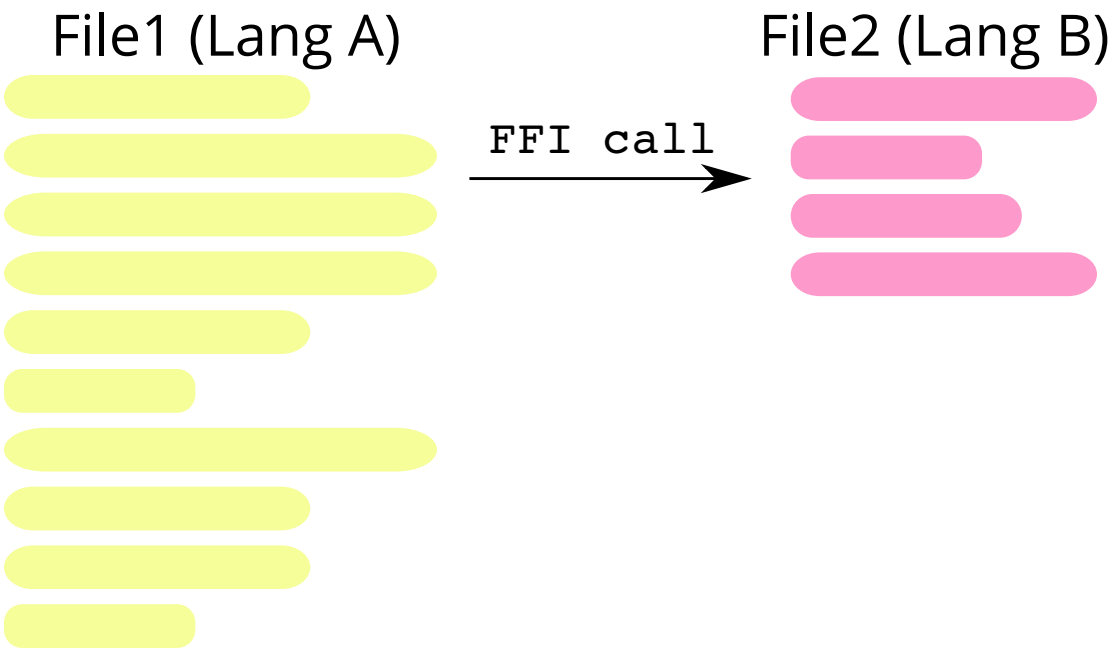
Language Composition

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

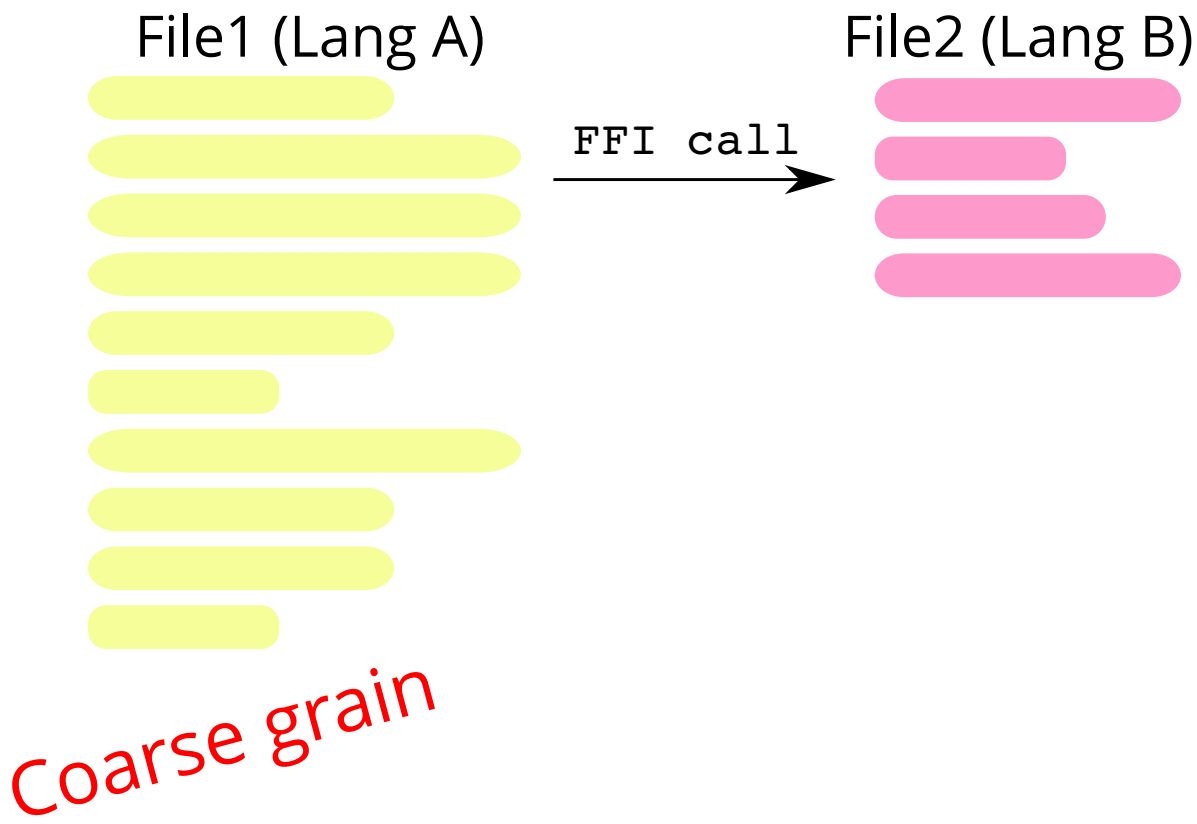
Why Compose Languages?

- Parts of a program are expressed best with different languages.
  - User Interface
  - String manipulation
  - Statistical analysis
  - Constraint solving
  - . . .
- Performance.
  - Start writing in expressive language X.
  - Port bottlenecks to fast language Y.
- Language migration.
  - Gradual reimplementation.

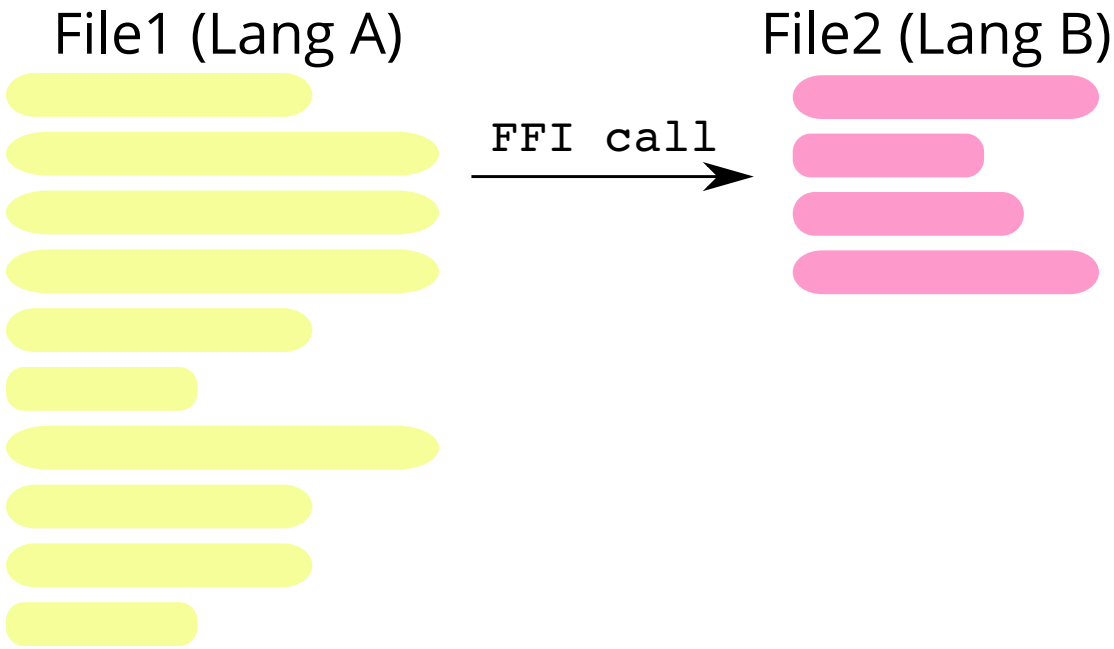
Most languages have a Foreign Function Interface



Most languages have a Foreign Function Interface



Most languages have a Foreign Function Interface

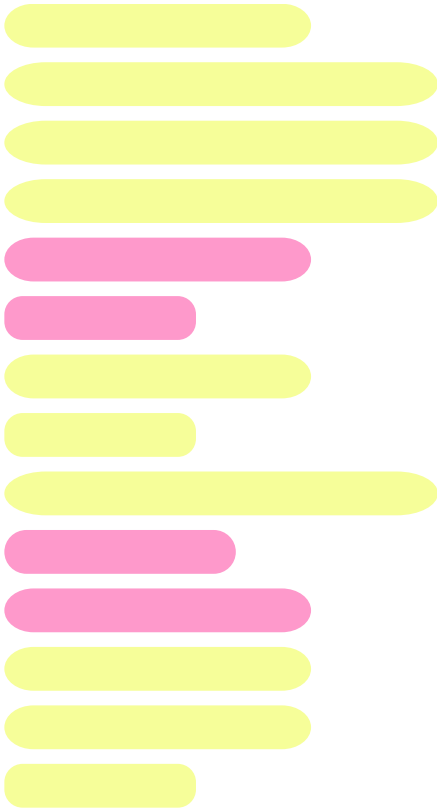


Coarse grain

Limited choice of langs

(second lang is nearly always C)

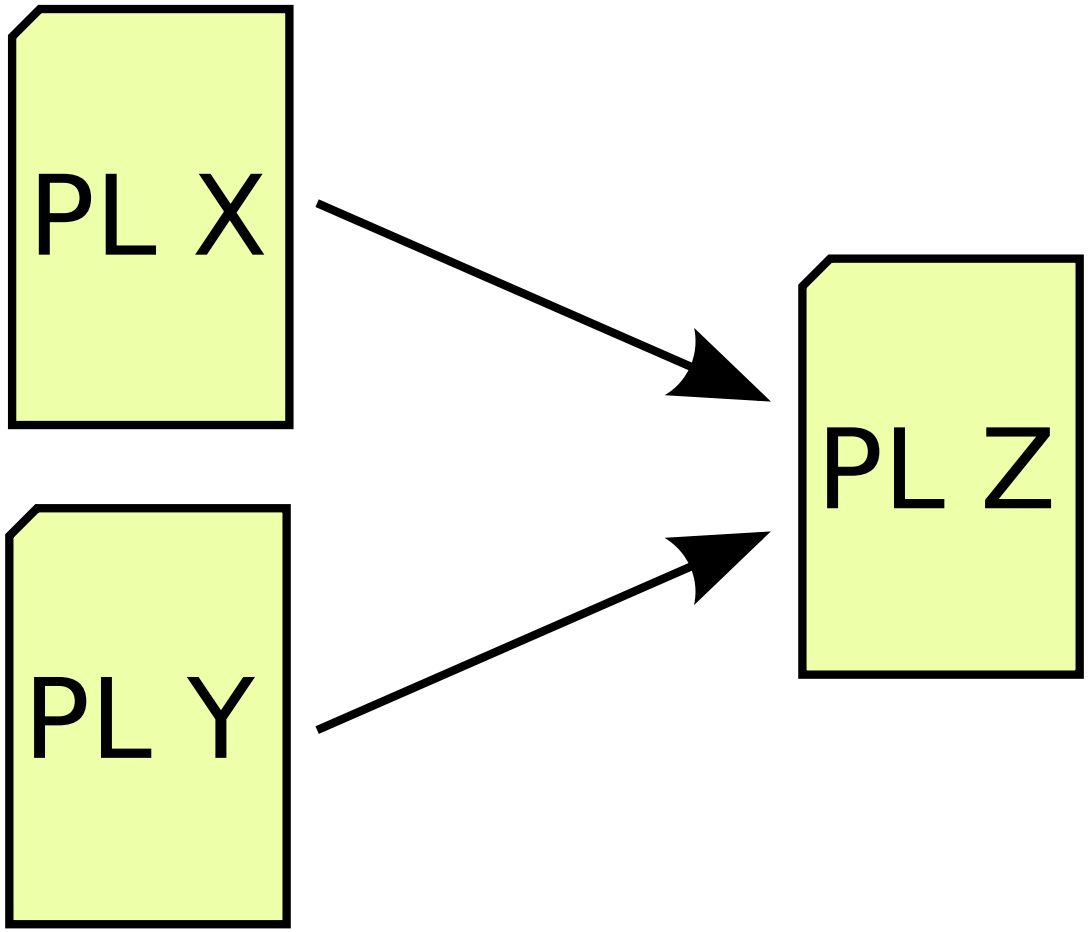
Can we do better?



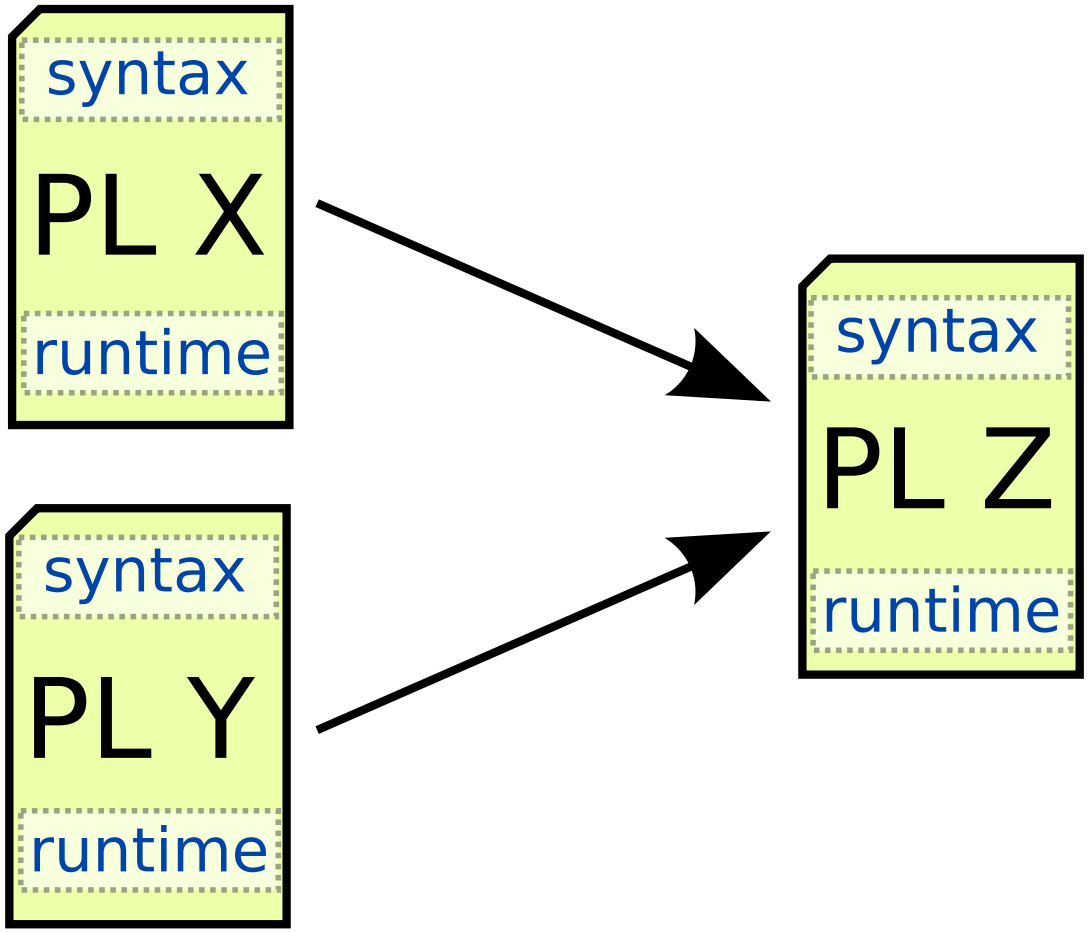
Our Aims:

- Fine-grain composition
  - Composition in the same file
  - Mix {methods, functions, expressions}
  - Integrate scoping
- Arbitrary languages
  - For now, dynamic languages.
- Make the composition fast.

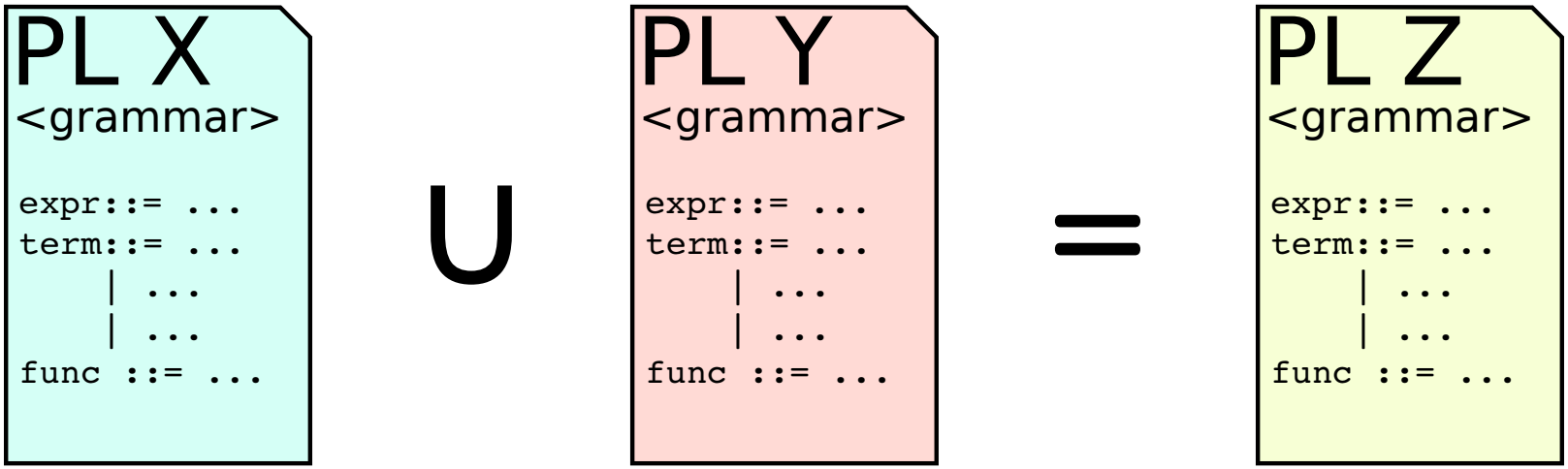
Breaking it Down



Breaking it Down



Composing Syntax



Easy?

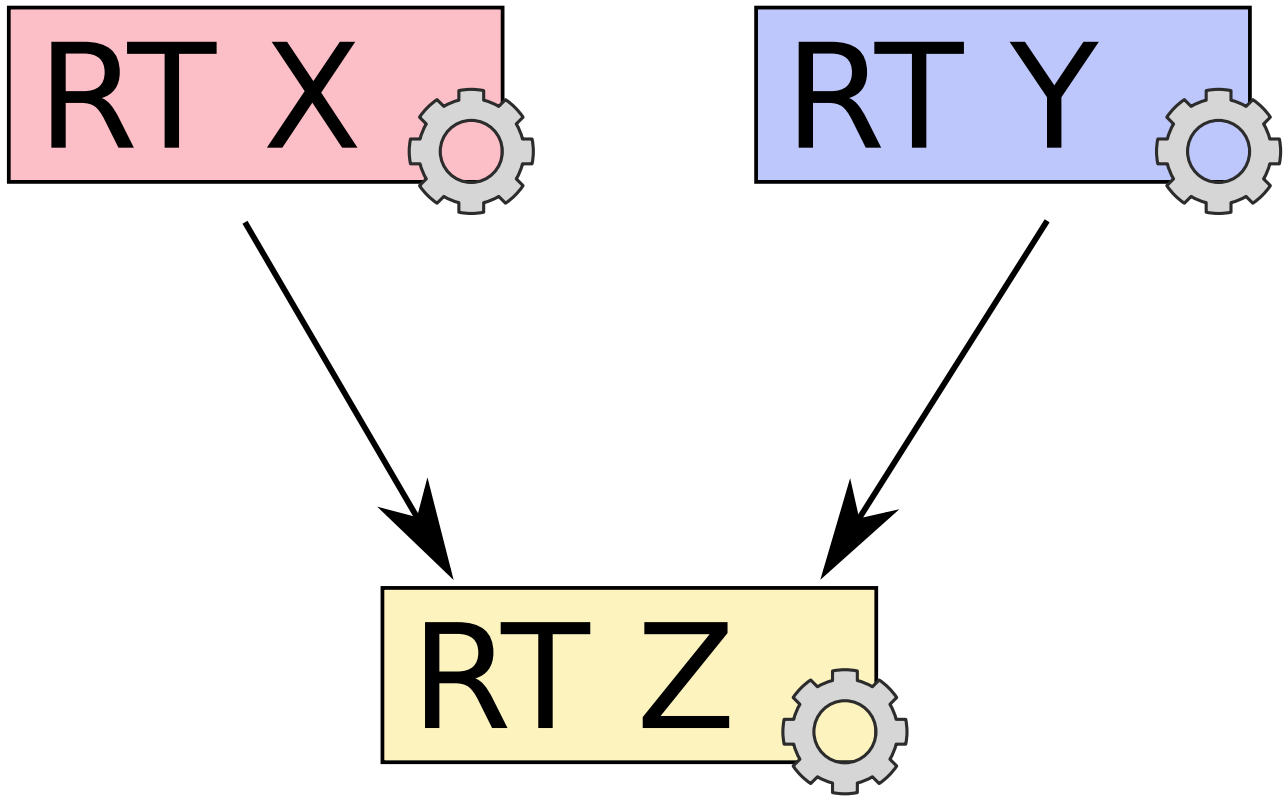
Composing Syntax

- LR $\rightarrow$ Possibly undefined.
- PEG $\rightarrow$ Shadows.
- GLR $\rightarrow$ Ambiguous.

Syntax Directed Editing

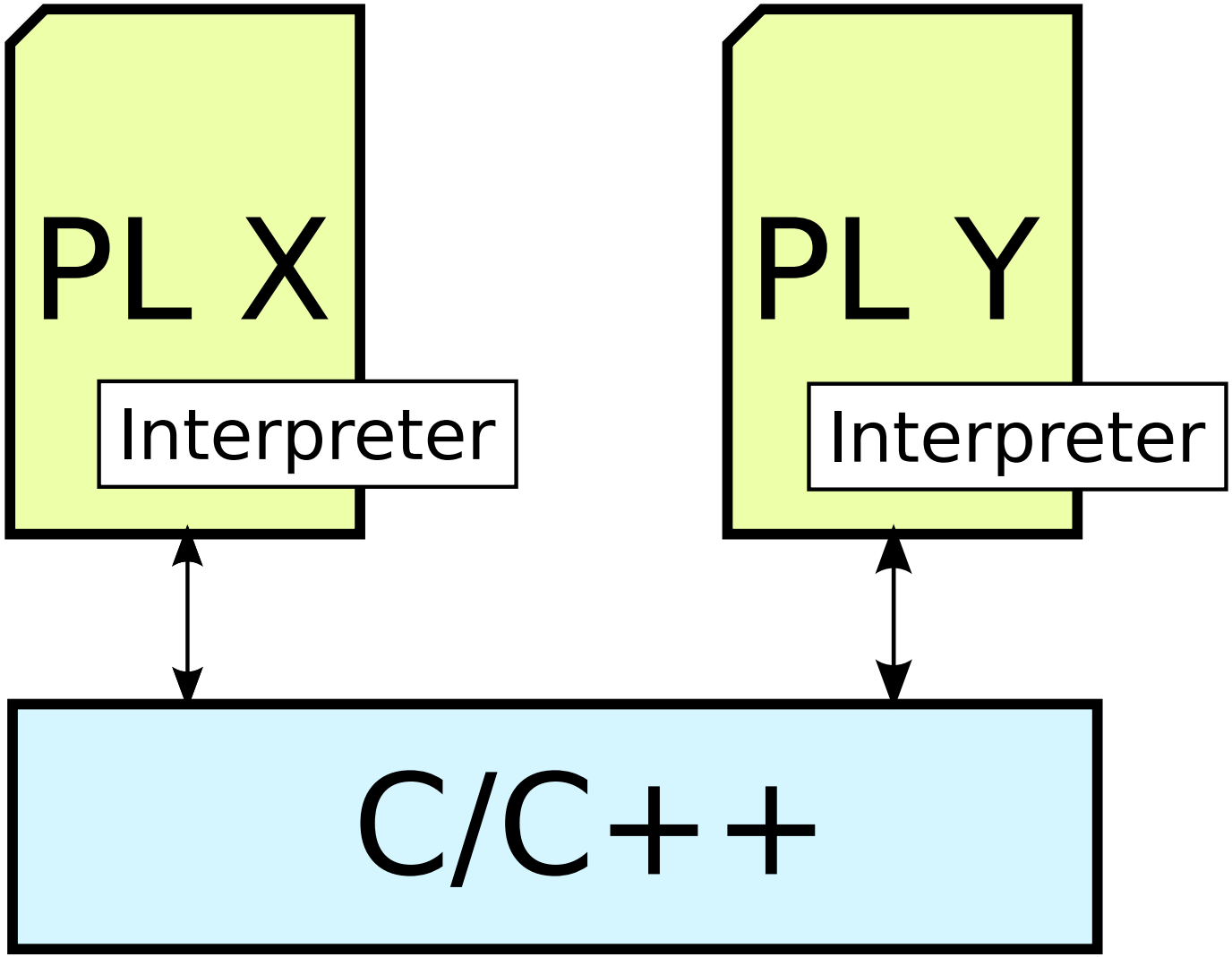
```
public class Say extends <none> implements <none> {

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

   <<methods>>

   <<nested classifiers>>
}
```

Composing Runtimes

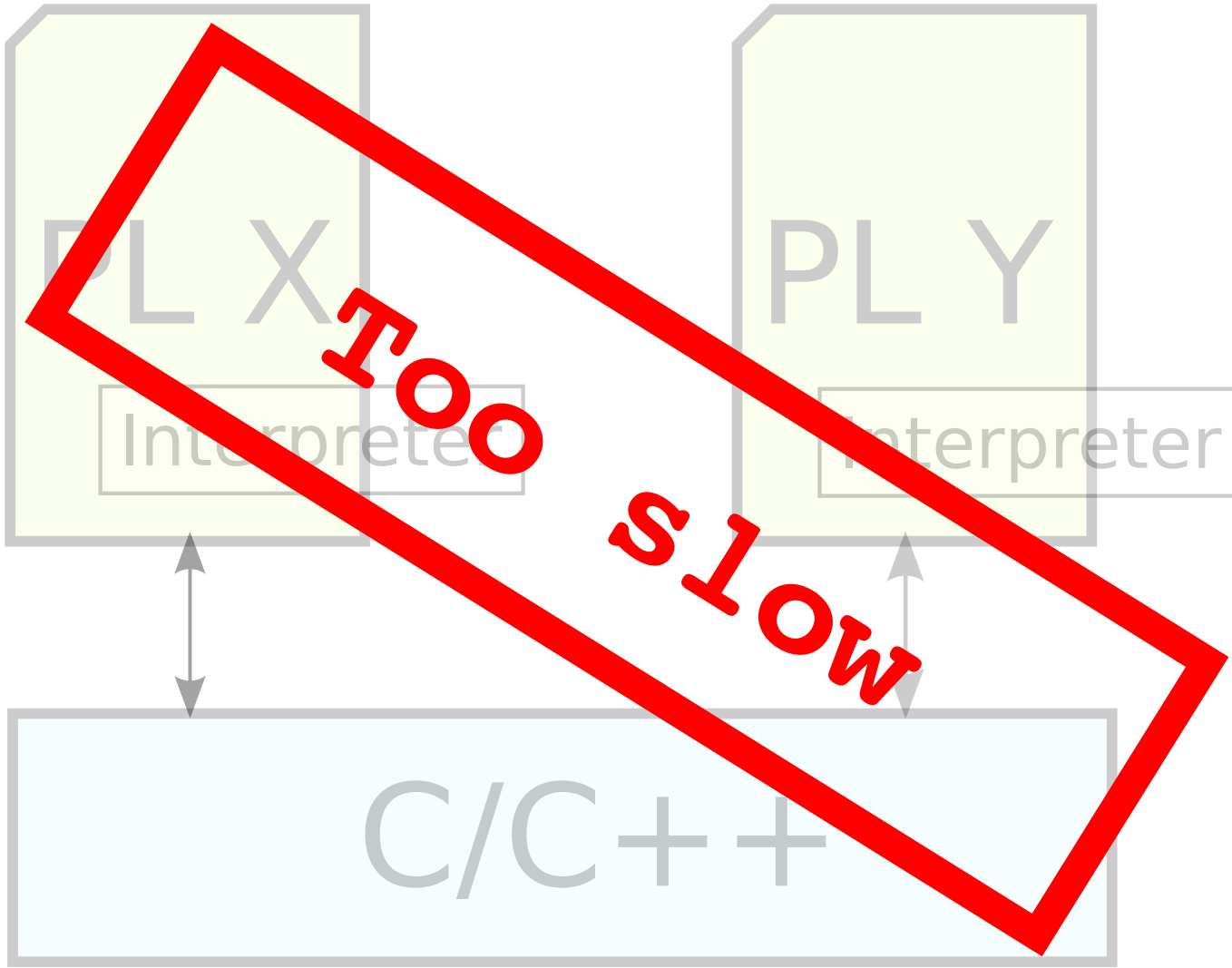


Easy?

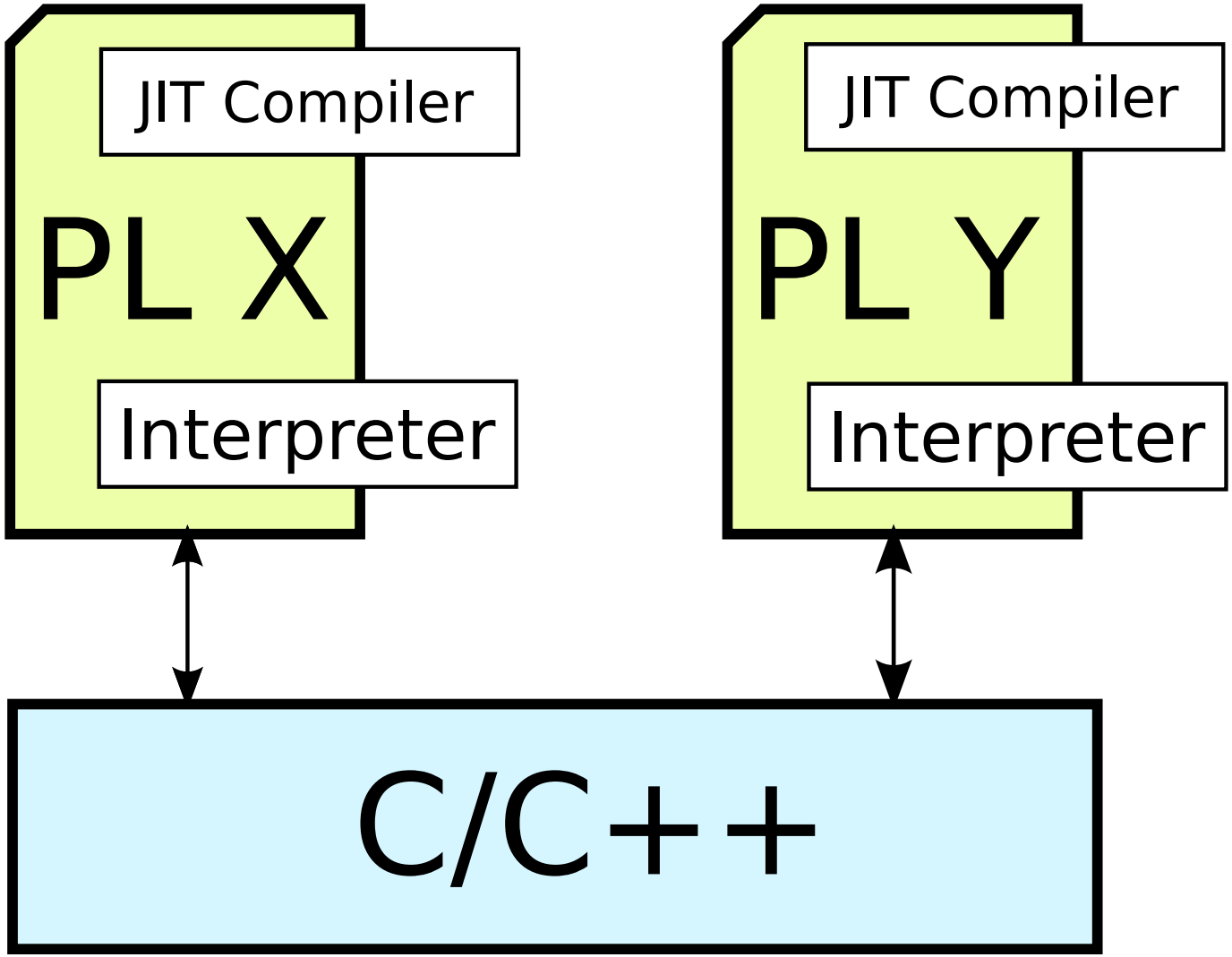
Runtime composition



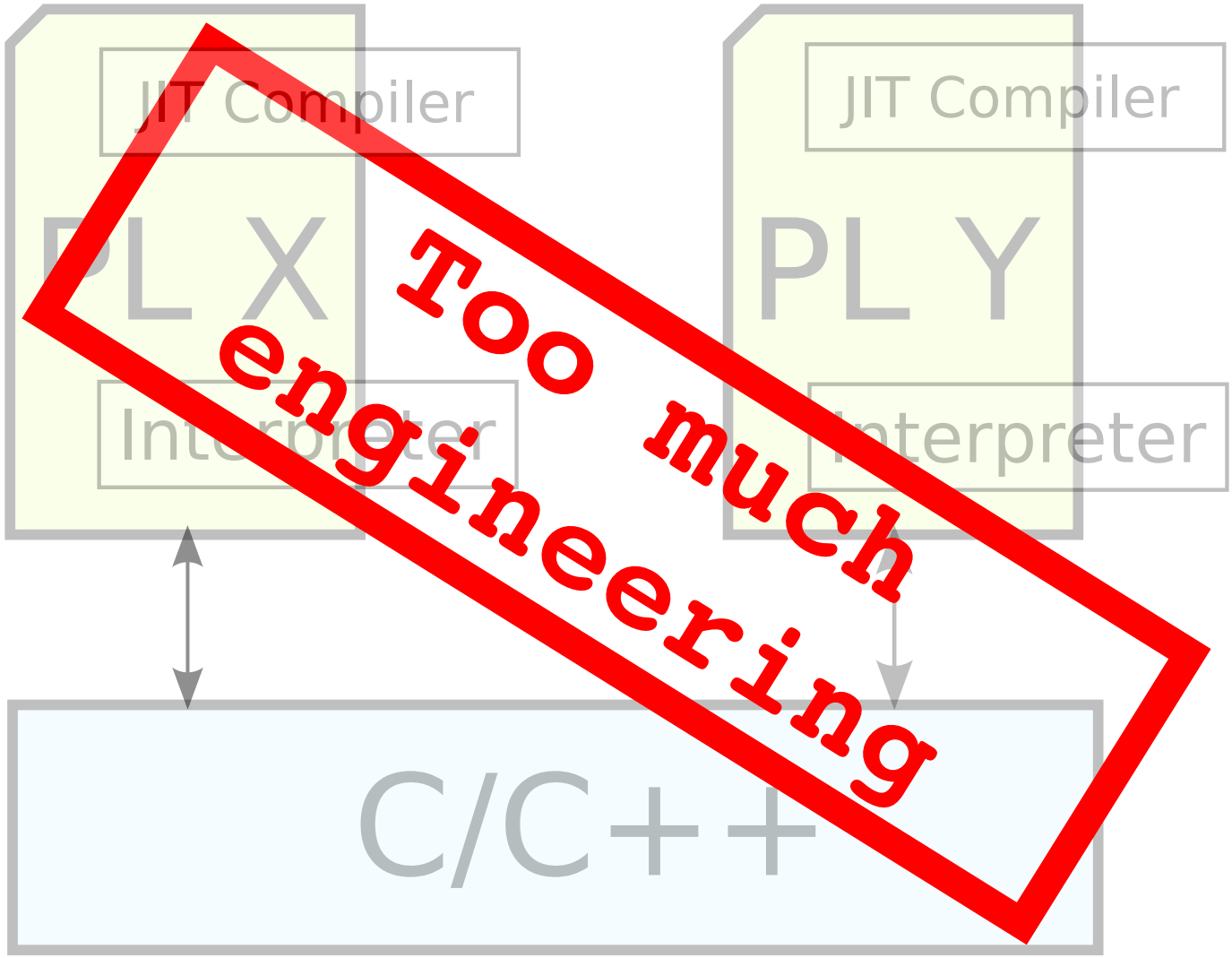
Runtime composition



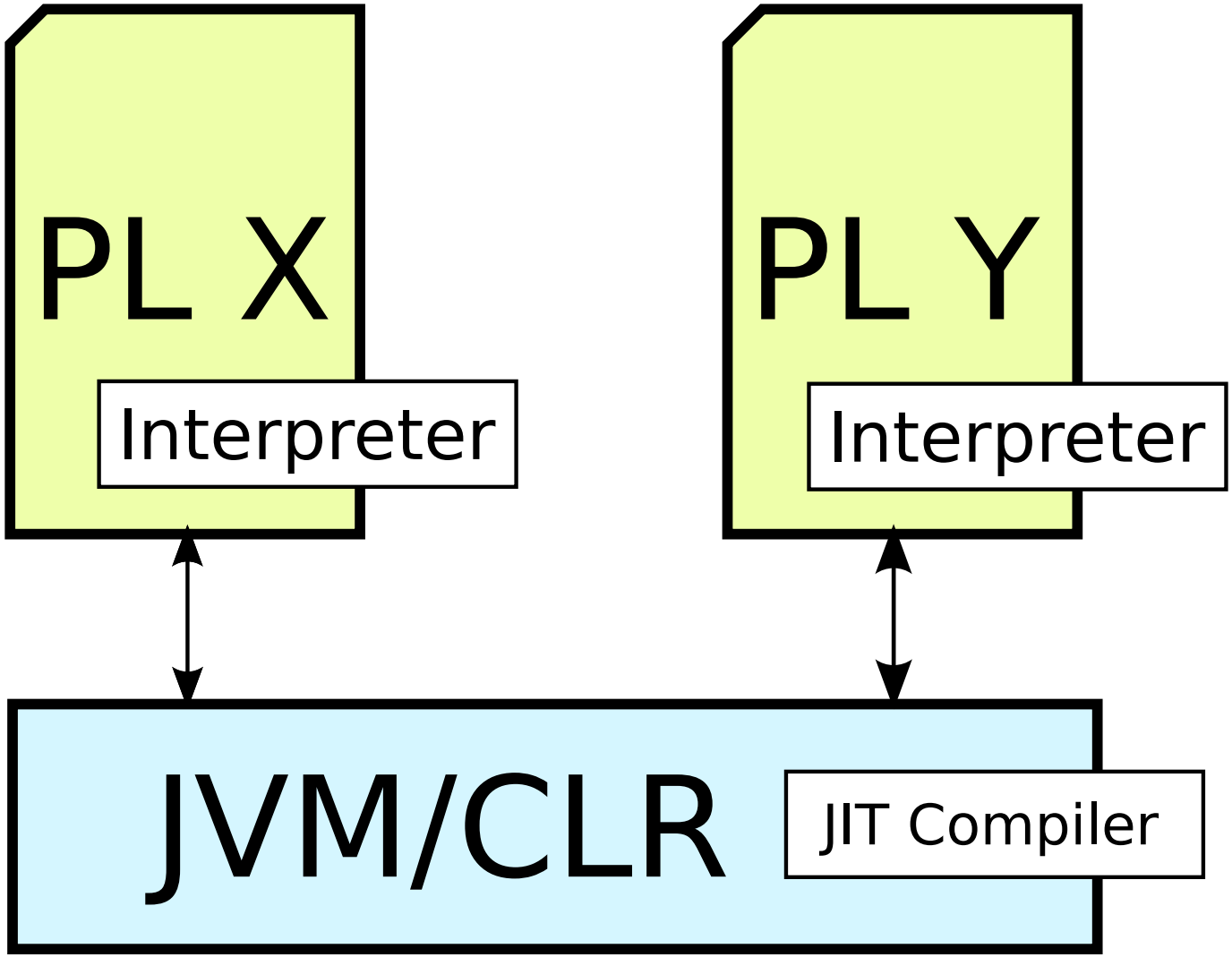
Runtime composition



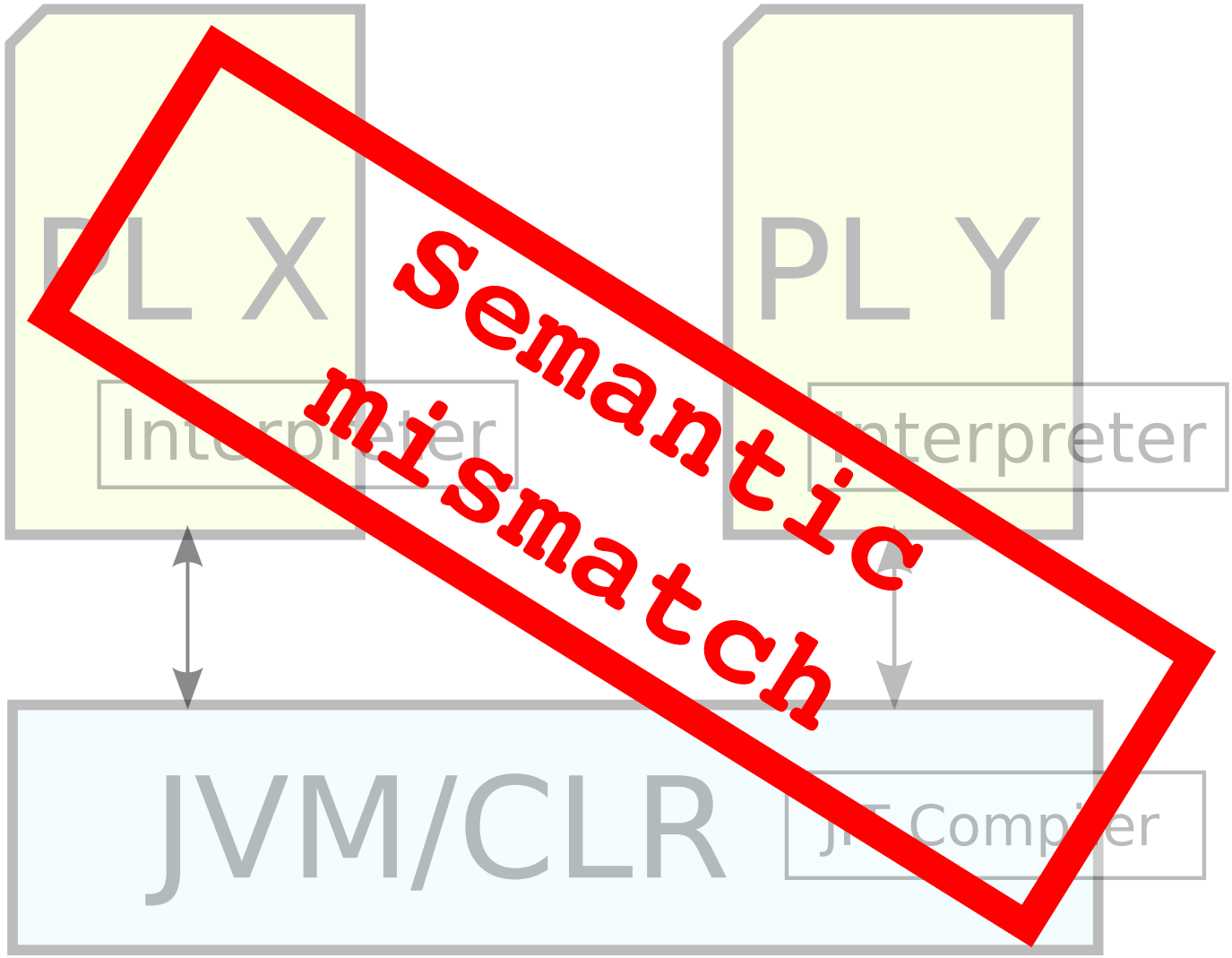
Runtime composition



Runtime composition



Runtime composition



Summary:

We need a practical way of composing syntax and runtimes.

Summary:

We need a practical way of composing syntax and runtimes.

Language Boxes + Meta-tracing

Language Boxes

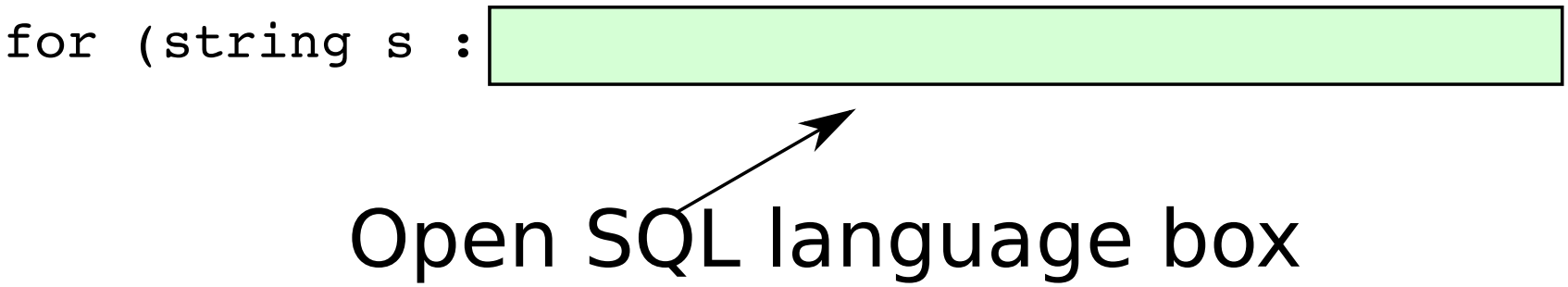
- Borrows ideas from SDE.
- Palatable editing experience.
- Simple and practical way to compose grammars.

Language Boxes: E.g. Java + SQL

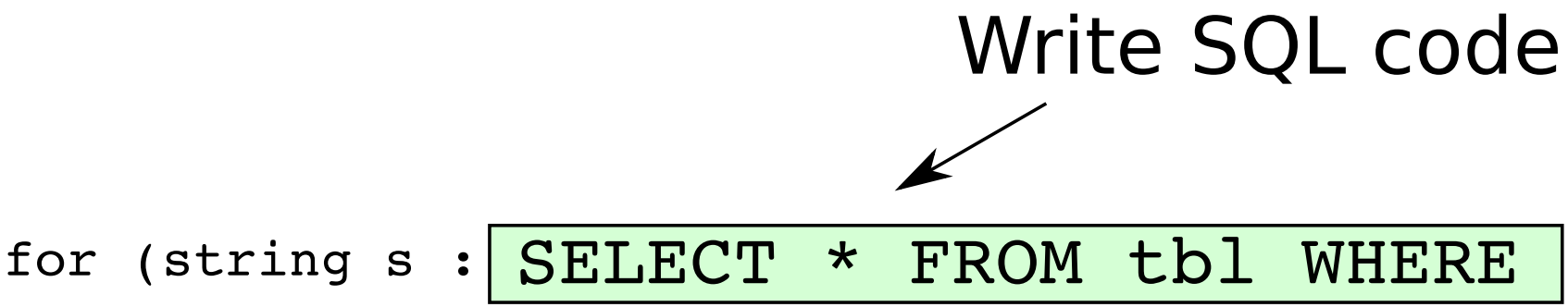
Begin writing Java code

```
for (string s :
```

Language Boxes: E.g. Java + SQL



Language Boxes: E.g. Java + SQL

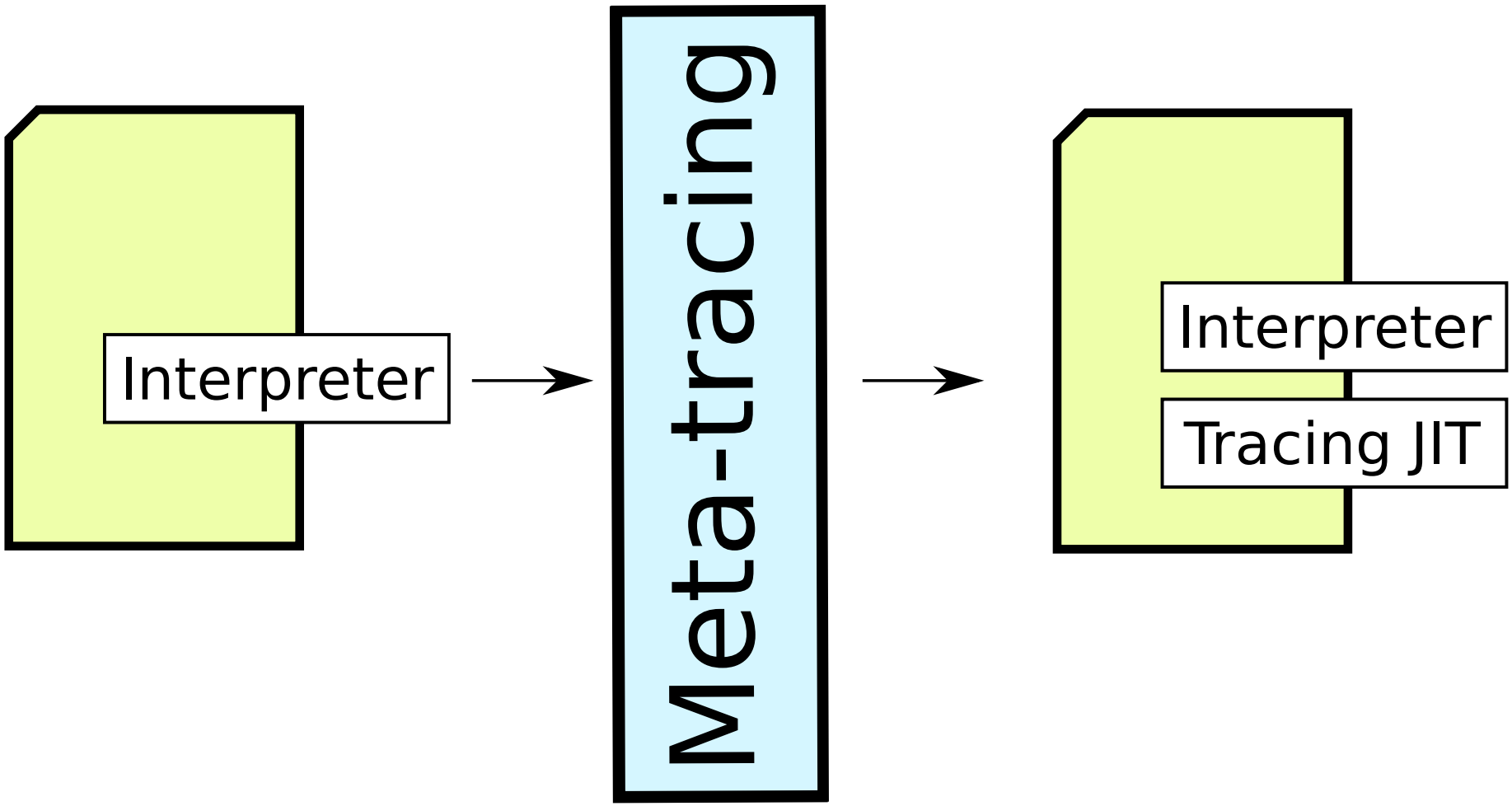


Language Boxes: E.g. Java + SQL

```
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)
        if off < 0: can_enter_jit(pc)
        pc += off
    elif ...:
        ...
```

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
```

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
```

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
```

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

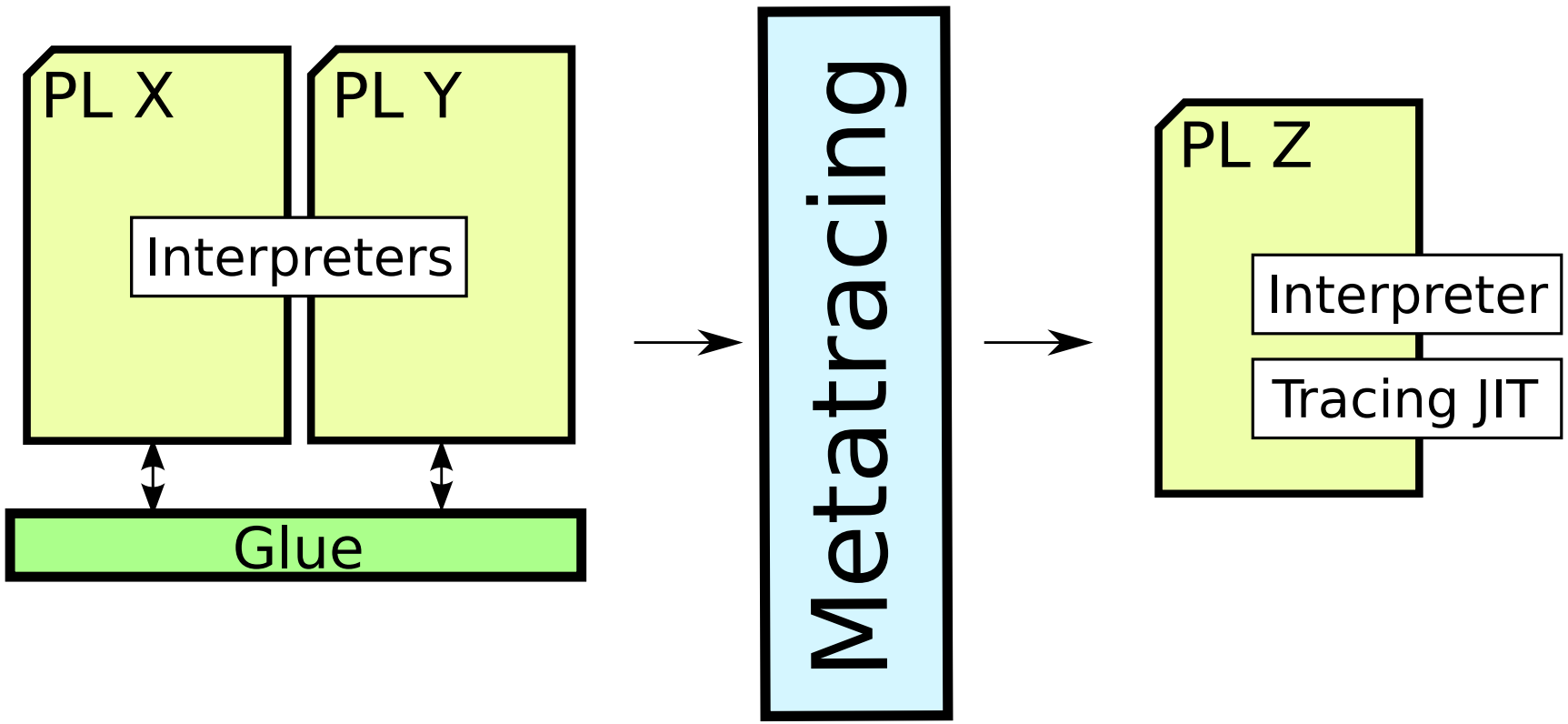
```
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
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)
```

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)
```

The “essence” of the loop

How Does this Apply to VM Composition?



Summarising our Approach

- Editing with Language boxes.
  - Traditional “code editor” look and feel.
  - Practical syntactic composition.

- Interpreter Composition with Meta-tracing
  - Relatively little engineering effort.
  - Compose any two languages written in RPython.
  - Language agnostic JIT optimisations.

Our Compositions

Our Language Compositions

Eco + RPython

Our Language Compositions

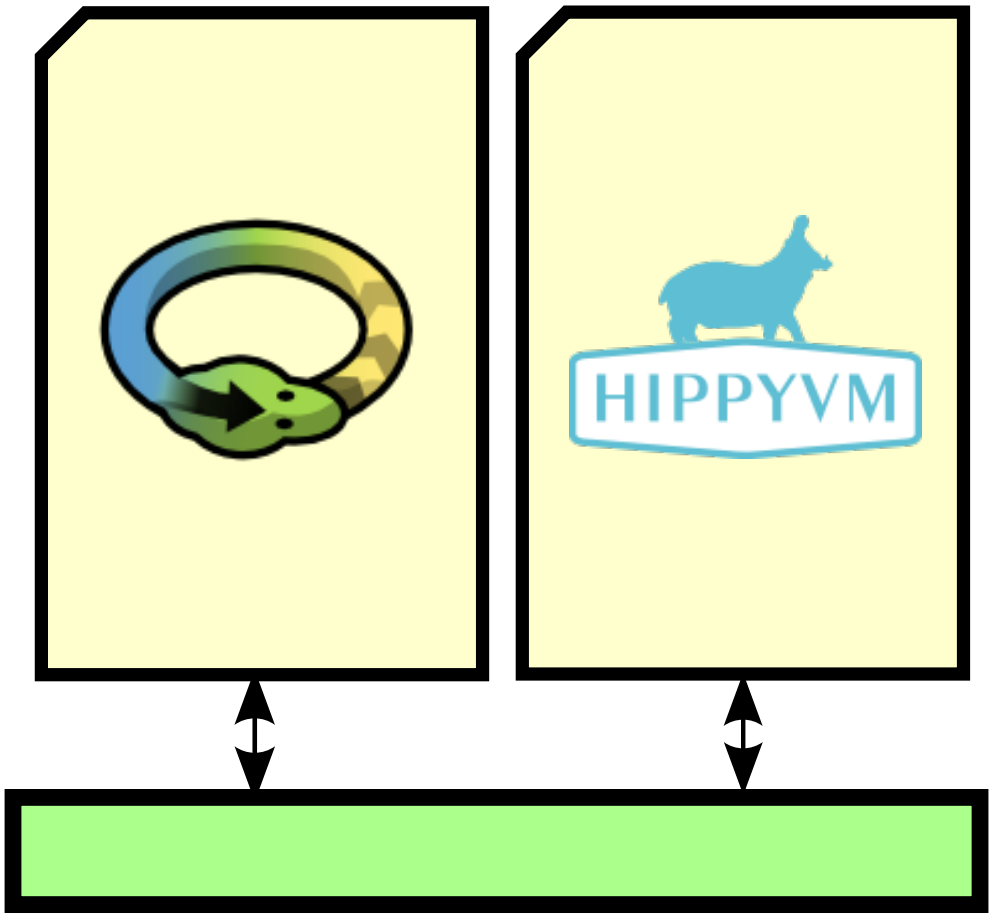
- Python + Prolog
- PHP + Python
- Python + SQLite

Our Language Compositions

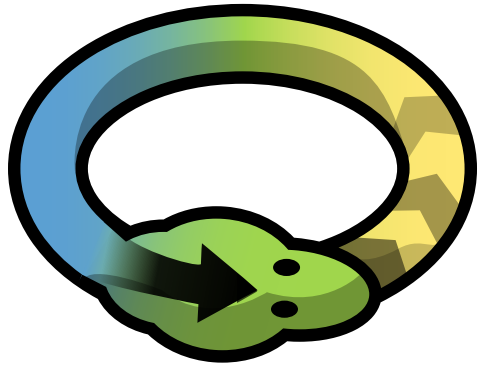
- Python + Prolog
- PHP + Python = PyHyp
- Python + SQLite

PyHyp

PyHyp



PyPy

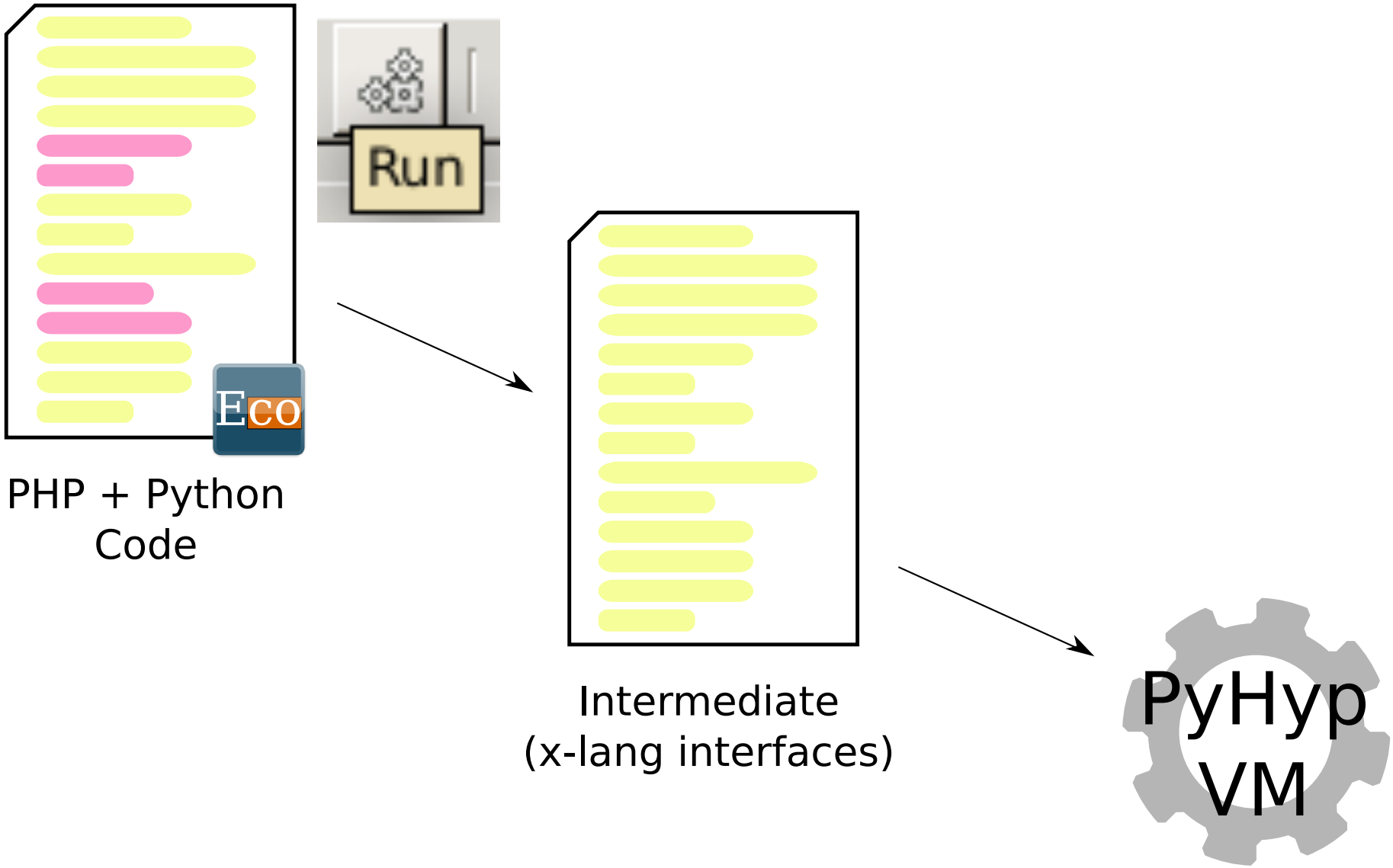


- Fast, industrial strength, Python interpreter
- Spawned the RPython tool-chain.
- Targets Python 2.7 or 3.x
  - (We use the 2.7 branch)

HippyVM

- Young PHP 5.4 interpreter.
- Not yet complete.
- Performance competitive with HHVM.

Edit/Execute



Features of PyHyp

- FFI-like features
  - Calling Python functions and methods from PHP
  - Calling PHP functions and methods from Python
  - Automatic type “conversion”

- Advanced features
  - Adds support for references to Python
  - Arbitrary nesting of foreign functions
  - Cross-language scoping
  - Python expressions in PHP
  - “Embedding” Python methods inside PHP classes
  - Access modifiers

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";
```

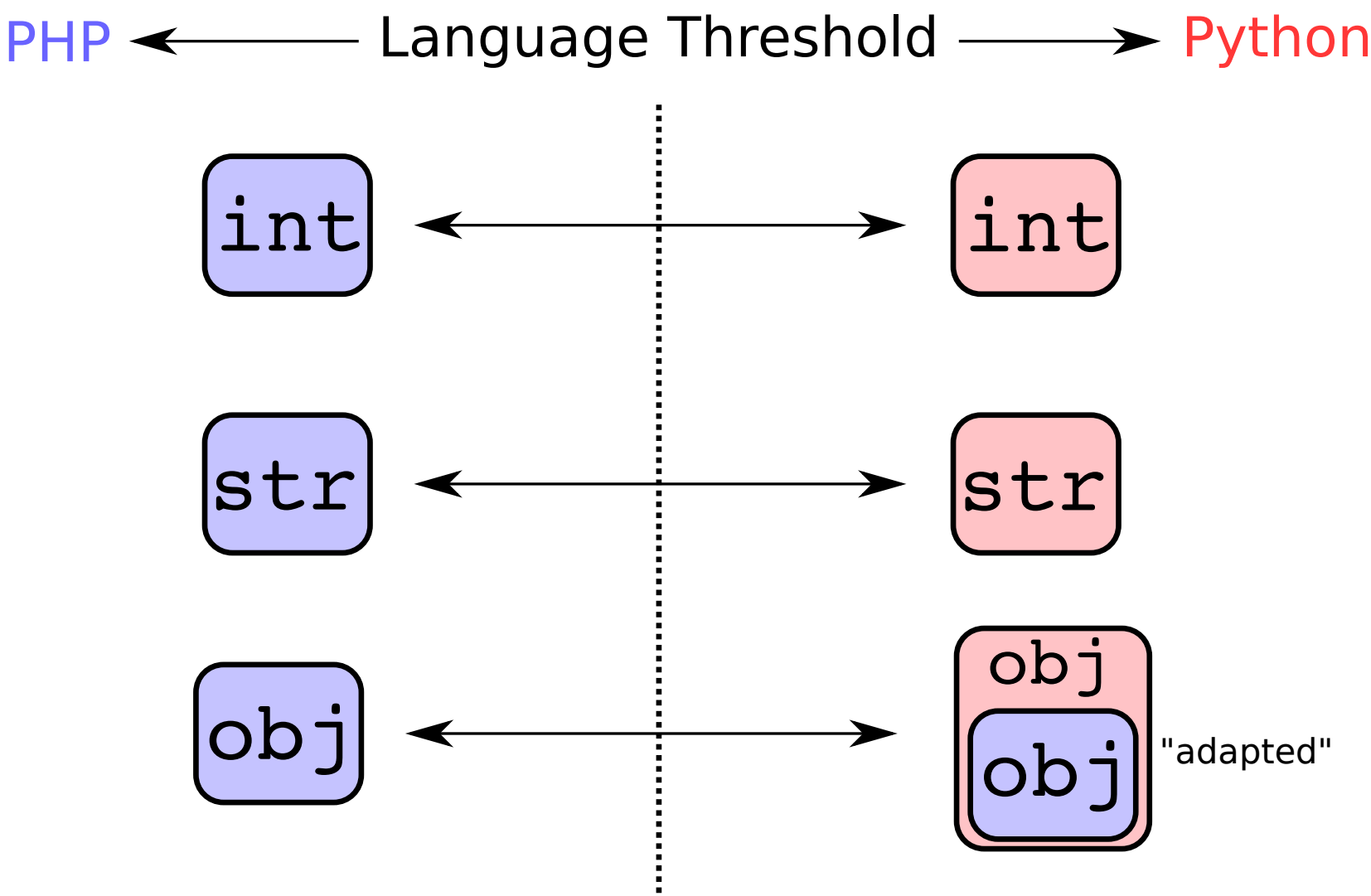
Callee in Python

Semantic Friction: References

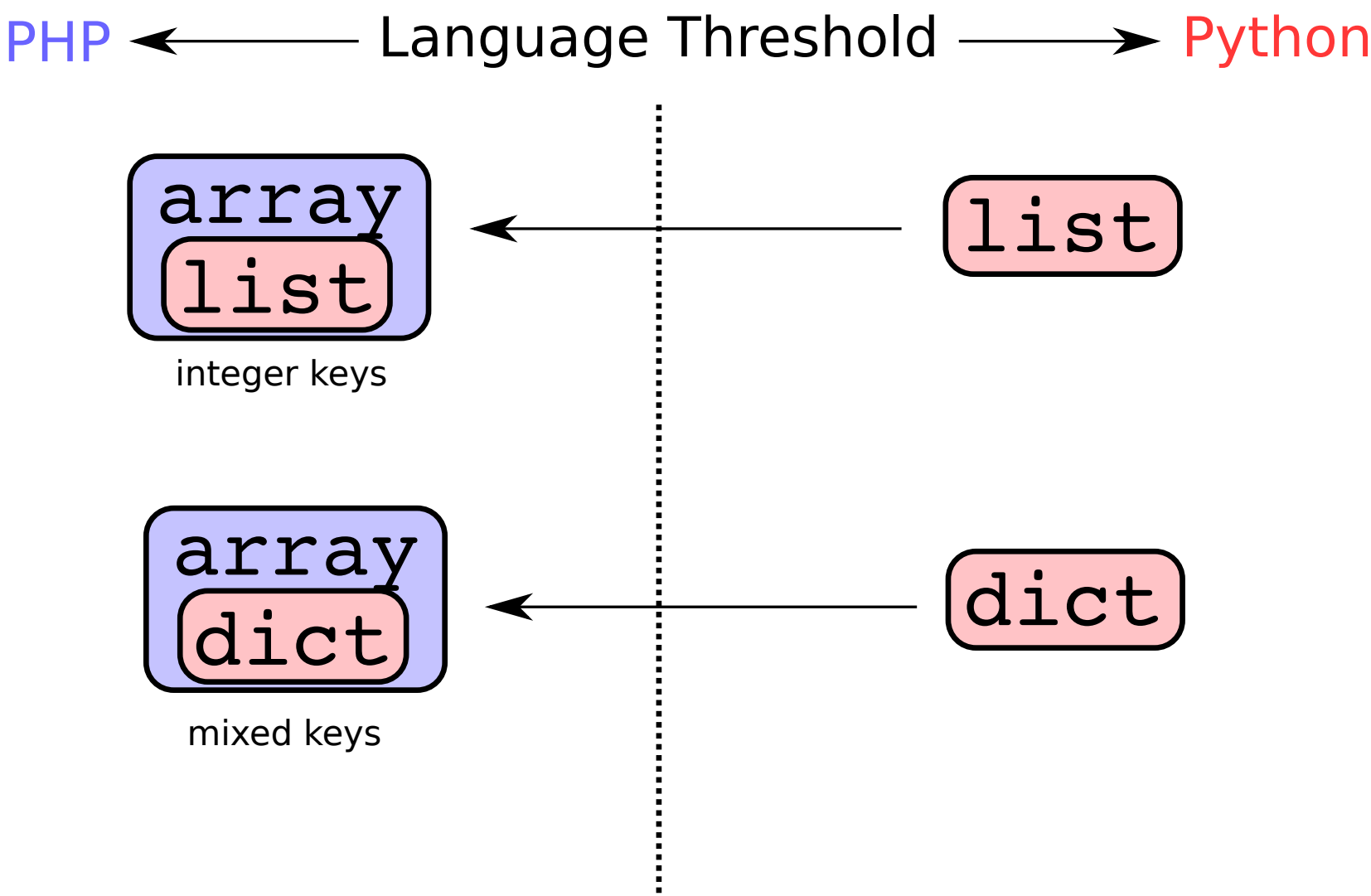
```
function swap(&$a, &$b) {
    $temp = $a;
    $a = $b;
    $b = $temp;
}

...

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

Caller in Python

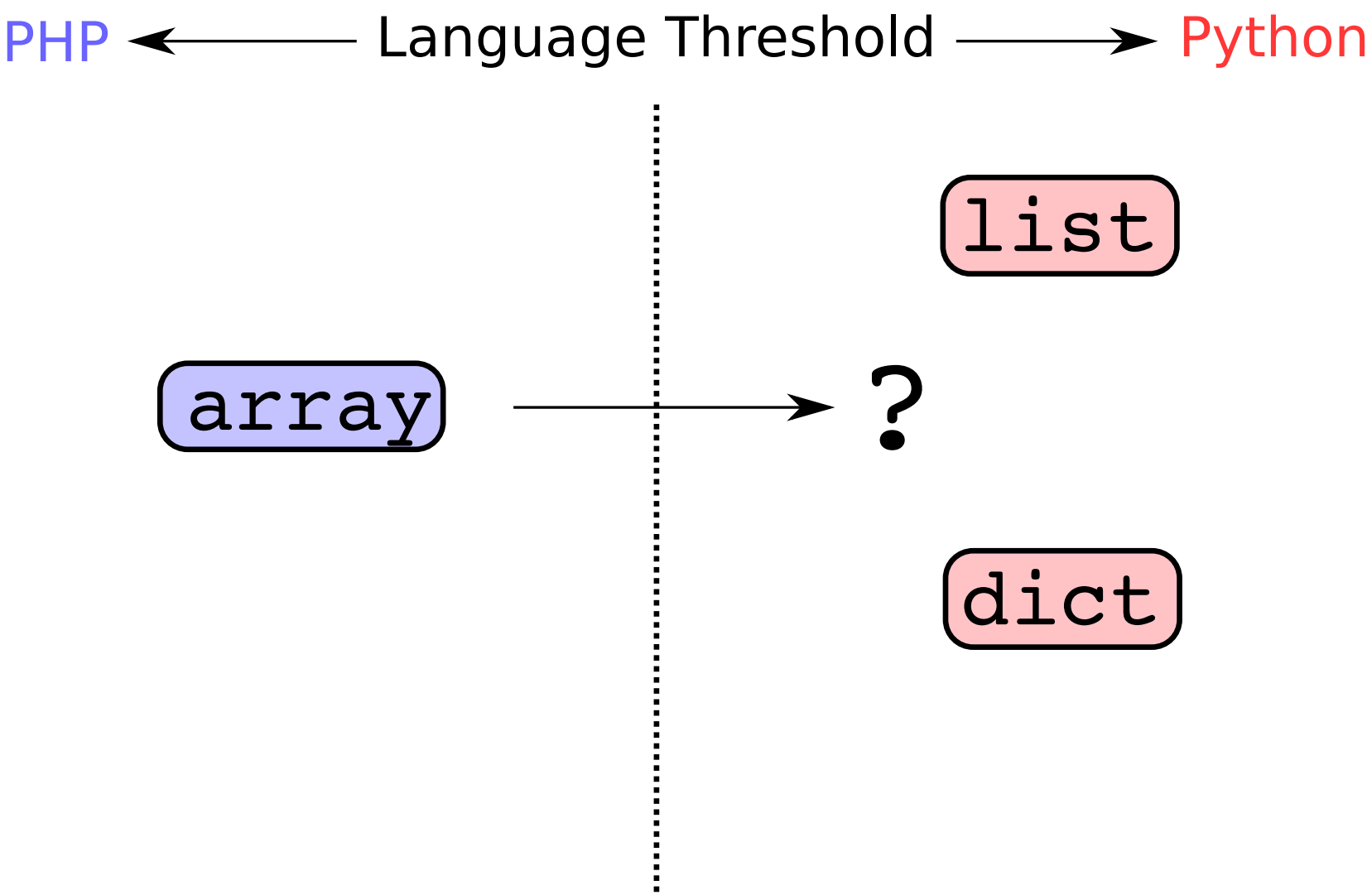
Semantic Friction: Array/Dict/List Conversions



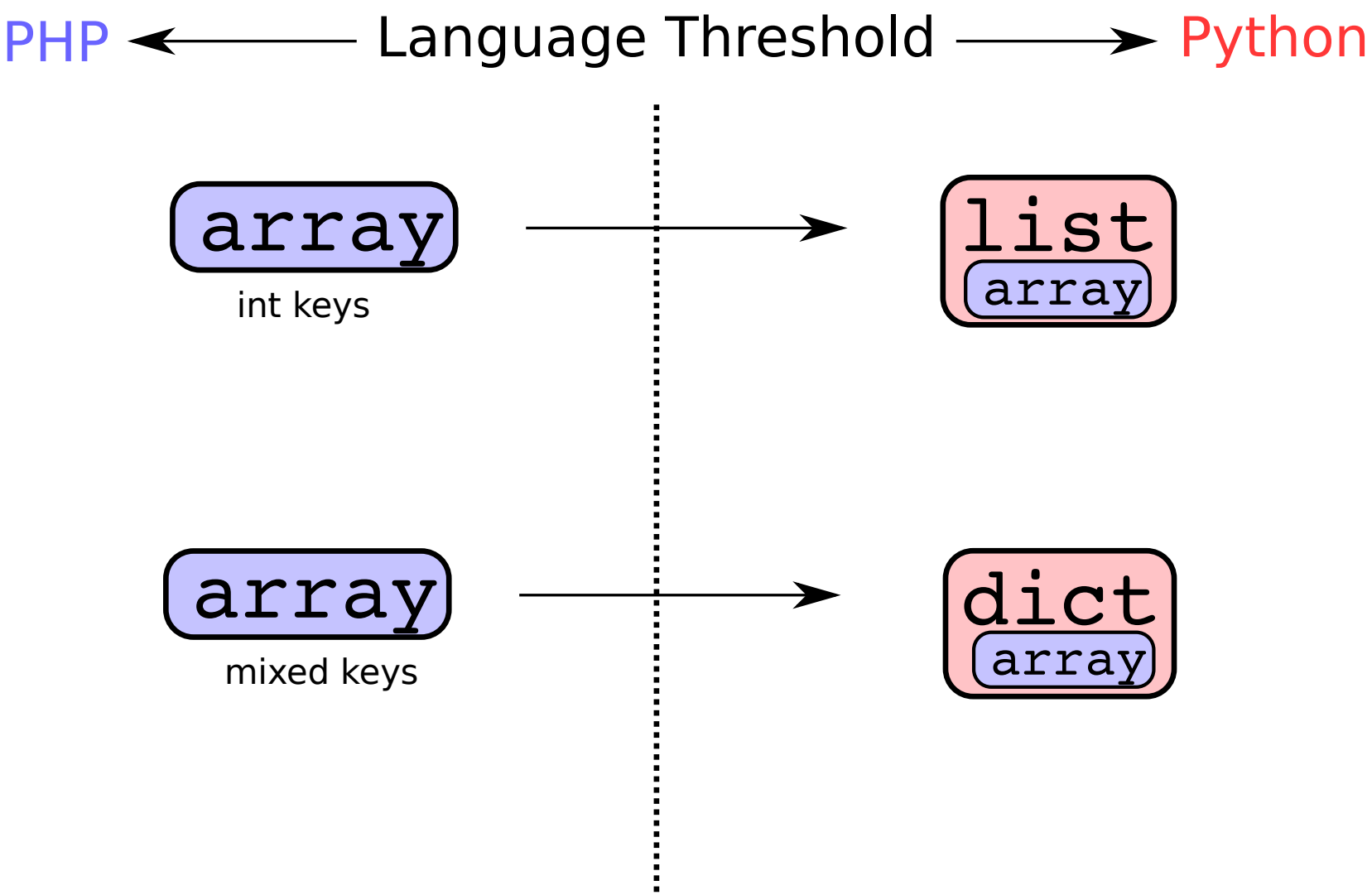
Semantic Friction: Array/Dict/List Conversions



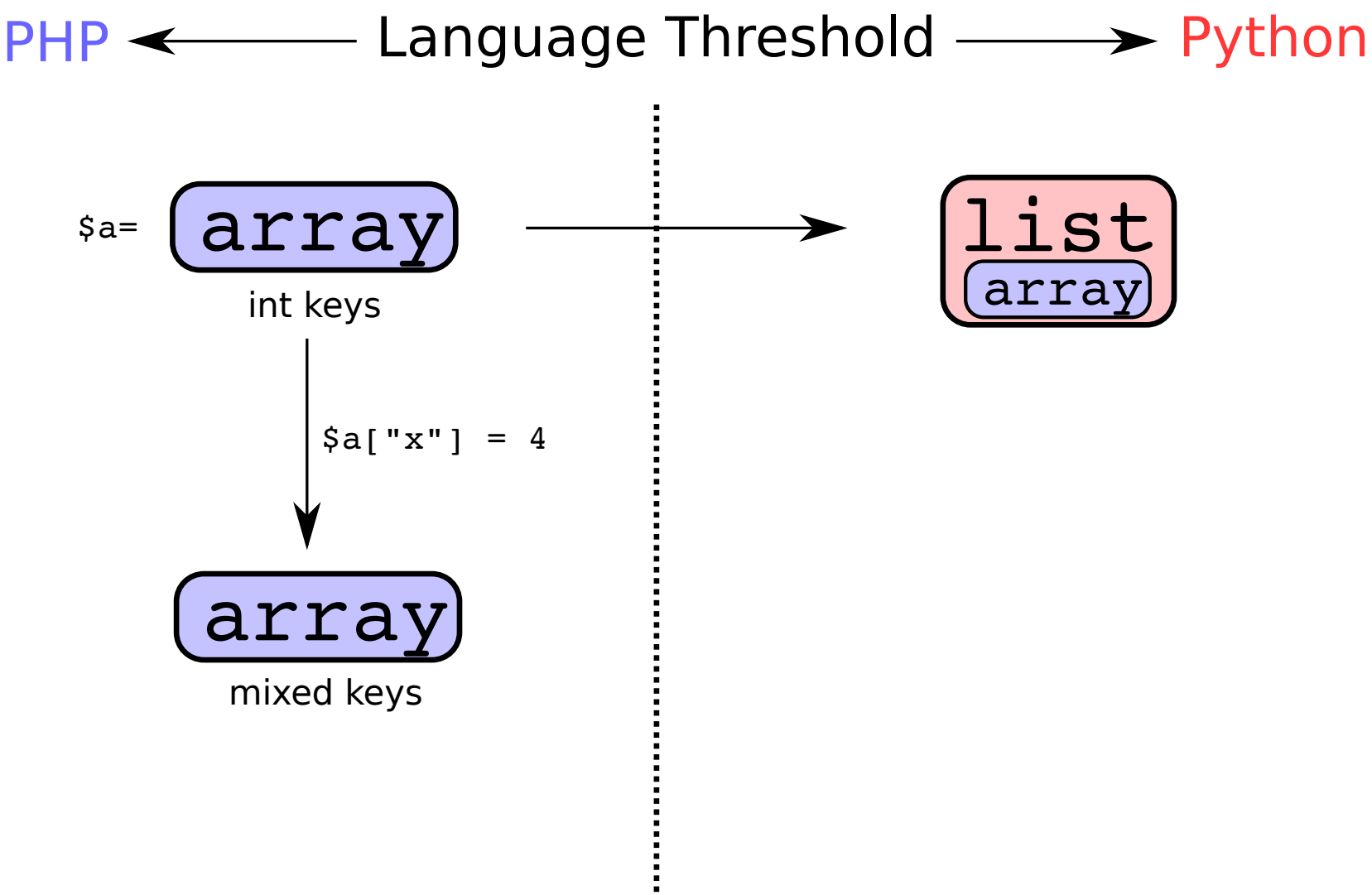
Semantic Friction: Array/Dict/List Conversions



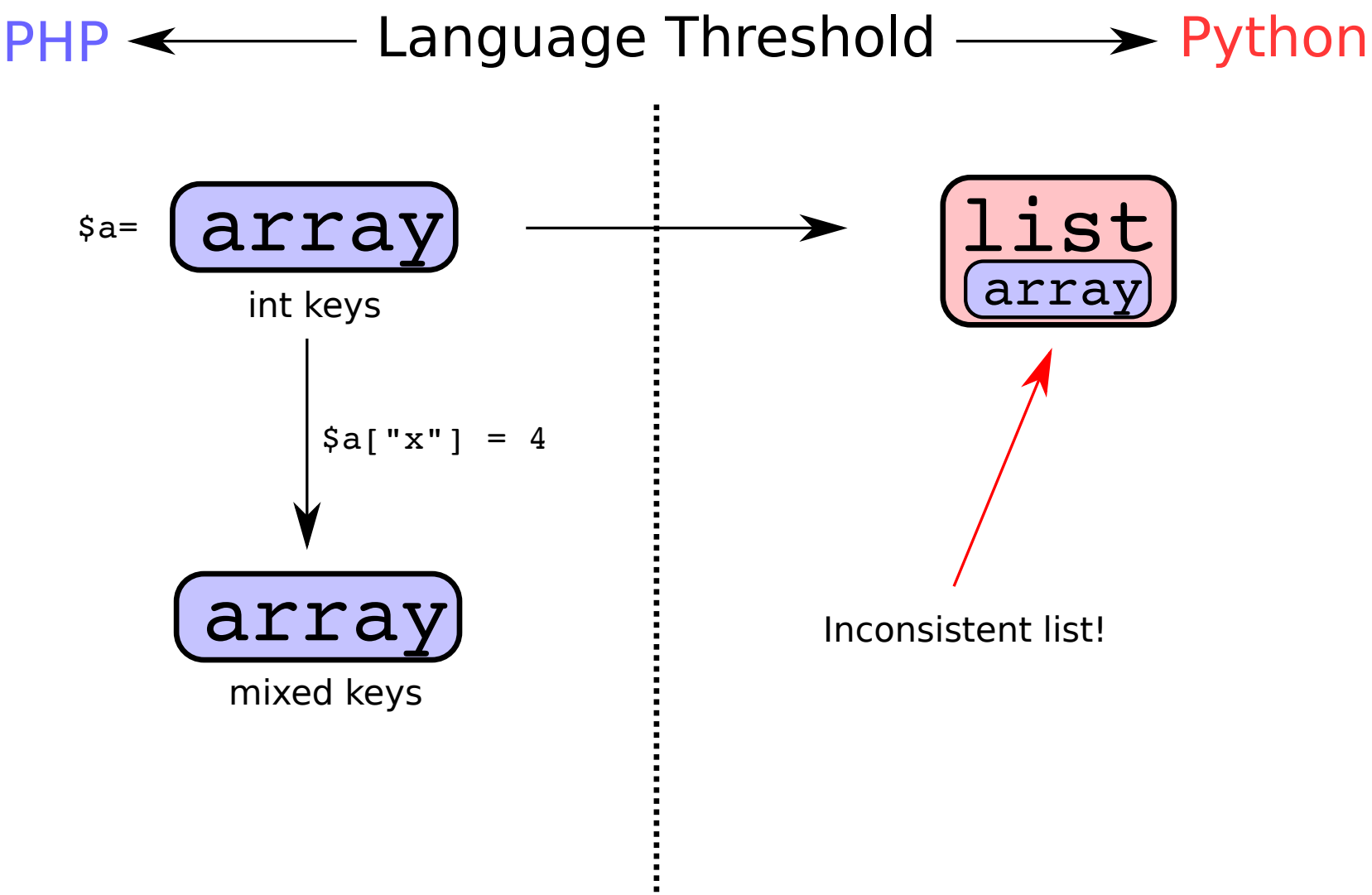
Semantic Friction: Array/Dict/List Conversions



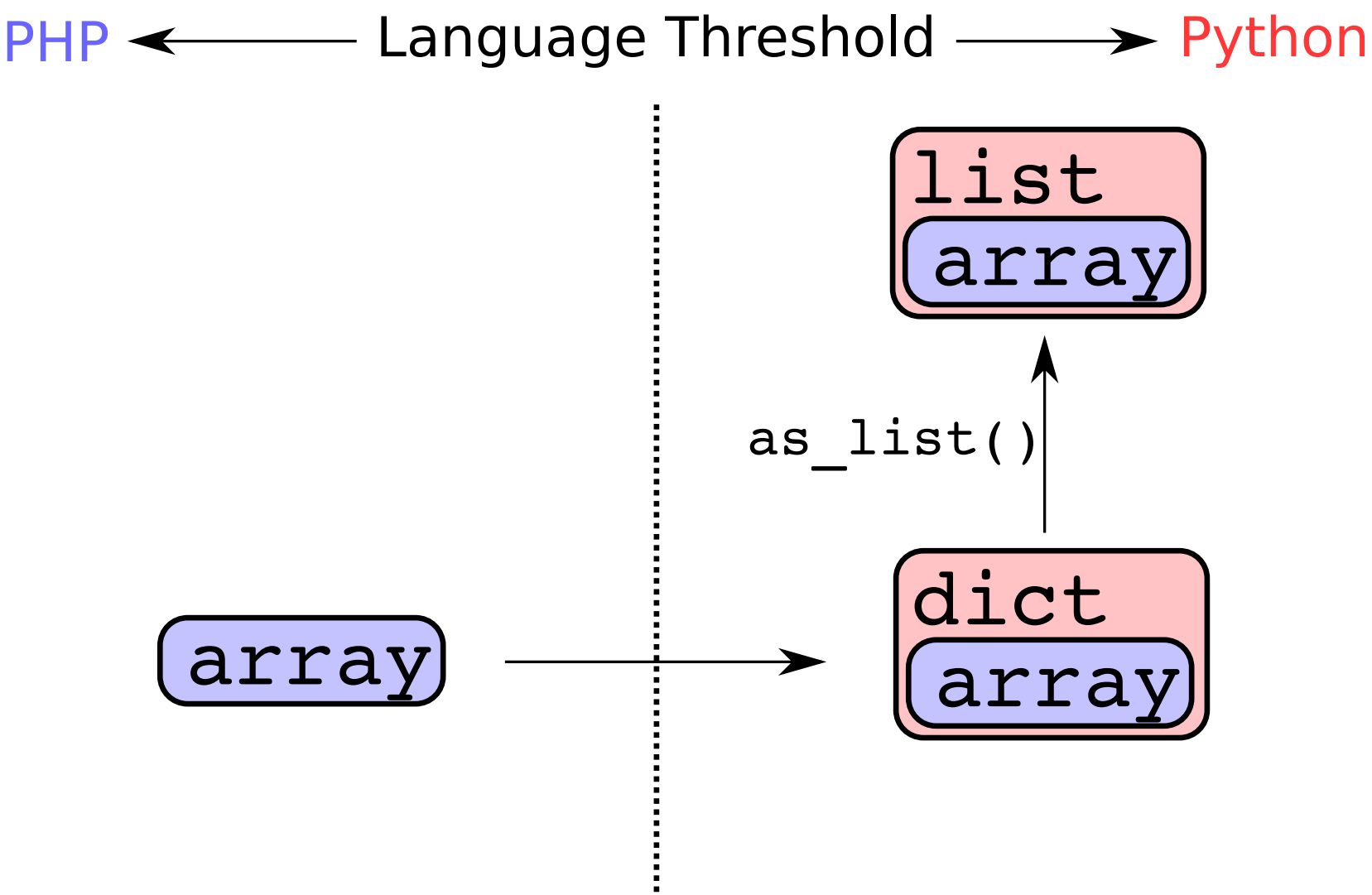
Semantic Friction: Array/Dict/List Conversions



Semantic Friction: Array/Dict/List Conversions



Semantic Friction: Array/Dict/List Conversions



Semantic Friction: Cross Language Scoping

```
1:
2: $range = 10;
3:
4: def f():
5:     print(range)
6:
7: f();
```

Semantic Friction: Cross Language Scoping

```
1:
2: $range = 10;
3:
4: def f():
5:     print(range)
6:
7: f();
```

Should print: `10`

Semantic Friction: Cross Language Scoping

```
1:
2:
3:
4: def f():
5:     print(range)
6:
7: f();
```

Semantic Friction: Cross Language Scoping

```
1:
2:
3:
4: def f():
5:     print(range)
6:
7: 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 “omnipresent” namespace of the current language.
3. Look in the “omnipresent” namespace of the other language.

“Omnipresents”

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

Experimental Evaluation

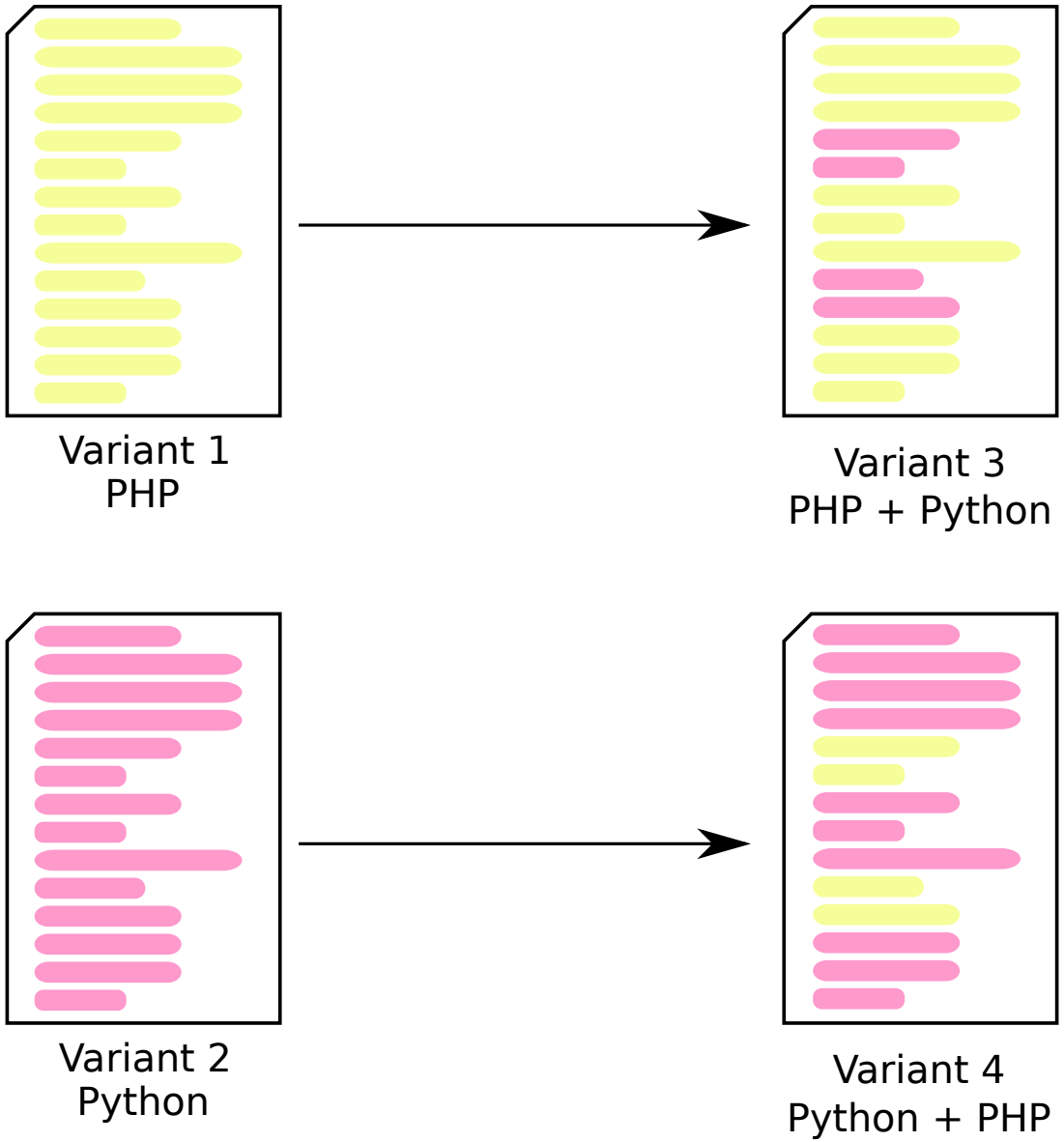
Benchmarks

Microbenchmarks

“Larger” benchmarks

Benchmark Variants

Benchmarks


Variant 1
PHP
Variant 3
PHP + Python
Variant 2
Python
Variant 4
Python + PHP

“Good Performance”

Composed variant on PyHyp should perform “close” to mono variants on constituent interpreters.

Aim for between 1-2x slower. 3x is too slow.

For completeness, benchmark against other PHP and Python implementations too.

Microbenchmarks: Relative to PyHyp Variant3

Benchmark	CPython	HHVM	HippyVM	PyHyp$_m$	PyPy	Zend
instchain	22.172 ±0.0859	6.209 ±0.0234	0.969 ±0.0036	0.967 ±0.0039	0.477 ±0.0019	24.248 ±0.1191
l1a0r	71.633 ±1.4869	3.770 ±0.0793	1.230 ±0.0255	1.230 ±0.0256	1.231 ±0.0298	37.752 ±1.1719
l1a1r	76.171 ±0.1207	3.000 ±0.0038	1.285 ±0.0003	1.285 ±0.0002	1.144 ±0.0077	41.052 ±0.3498
lists	7.485 ±0.0227	0.879 ±0.0072	0.966 ±0.0037	0.977 ±0.0041	0.520 ±0.0018	16.106 ±0.0736
ref_swap		6.911 ±0.0054	1.000 ±0.0003	1.000 ±0.0003		55.360 ±0.7395
return_simple	108.576 ±0.2690	6.915 ±0.0009	1.000 ±0.0001	1.000 ±0.0002	0.889 ±0.0002	83.708 ±0.7264
scopes	123.284 ±1.5081	14.969 ±0.0391	4.528 ±0.0099	4.512 ±0.0588	1.000 ±0.0003	156.443 ±0.5742
smallfunc	184.778 ±0.3071	12.818 ±0.0099	1.000 ±0.0003	1.000 ±0.0003	1.000 ±0.0003	243.318 ±0.8453
sum	299.582 ±0.3659	19.083 ±0.0172	1.000 ±0.0005	1.000 ±0.0005	0.874 ±0.0003	427.513 ±2.6441
sum_meth	328.894 ±1.2870	23.714 ±0.0955	0.998 ±0.0030	0.999 ±0.0030	0.873 ±0.0026	456.739 ±2.9270
sum_meth_attr	127.800 ±0.1907	17.819 ±0.2655	1.001 ±0.0019	1.116 ±0.0015	0.925 ±0.0016	148.167 ±0.6554
total_list	14.266 ±0.0248	2.080 ±0.0031	0.695 ±0.0005	0.696 ±0.0019	0.510 ±0.0014	30.356 ±0.1679
walk_list	4.869 ±0.0340	0.373 ±0.0025	0.773 ±0.0060	0.774 ±0.0107	1.099 ±0.0082	10.700 ±0.0846

Larger Benchmarks: Relative to PyHyp Variant3

Benchmark	CPython	HHVM	HippyVM	PyHyp$_m$	PyPy	Zend
deltablue	19.199 ±0.6900	860.108 ±31.1392	4.739 ±0.1684	4.888 ±0.1766	0.405 ±0.0151	181.209 ±6.7871
fannkuch	18.616 ±0.0362	3.212 ±0.0133	1.869 ±0.0034	1.879 ±0.0024	1.009 ±0.0046	14.998 ±0.1032
mandel		0.883 ±0.0006	1.013 ±0.0089	1.003 ±0.0011		8.290 ±0.0623
richards	28.291 ±0.1091	12.726 ±0.1296	0.745 ±0.0036	0.766 ±0.0044	0.531 ±0.0026	27.081 ±0.1445

Overall: Relative to PyHyp Variant3

Benchmark	CPython	HHVM	HippyVM	PyHyp$_m$	PyPy	Zend
Geometric Mean	48.575 ±0.1493	6.698 ±0.0188	1.206 ±0.0032	1.218 ±0.0035	0.785 ±0.0024	56.521 ±0.1833

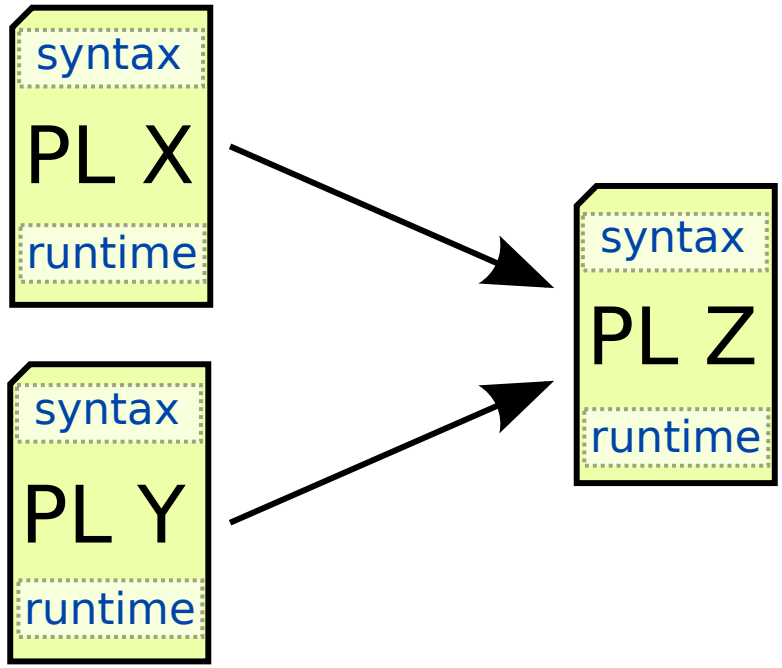
Conclusions

- Language boxes:
  - Practical composition of PL syntax.
  - Decent editor experience.
- Meta-tracing:
  - Compositions with relatively little effort.
  - Overall good performance.
- Implementing x-lang behaviours is easy.
- Designing x-lang behaviours is hard.
  - Thanks to semantic friction.

Future Work

- Debugging
  - Proper backtrace information.
  - Cross-language debugger.
- Compositions with >2 languages involved.
- Statically typed languages.

Thanks



Language Boxes + Meta-tracing

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
```

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
```

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
```

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)
```

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)
```

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

```
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)
```

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, 5)
dict_set(v2, "x", v23)
```

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)
```

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, 5)
dict_set(v2, "x", v23)
```

Trace optimisation: from 72 trace elements to 7.