Meta-Tracing, RPython, and PyPy

Carl
Friedrich
Bolz

KING'S College LONDON

Software Development Team

#vmss16

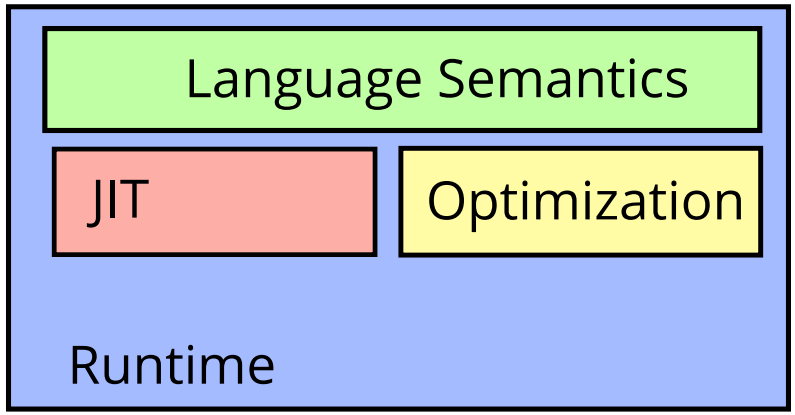
Motivation

An Alternate Approach to VM Construction

- Most productions VMs are written by hand in C/C++
- VMs are very tedious and costly to write
- Particularly for complex dynamically typed languages

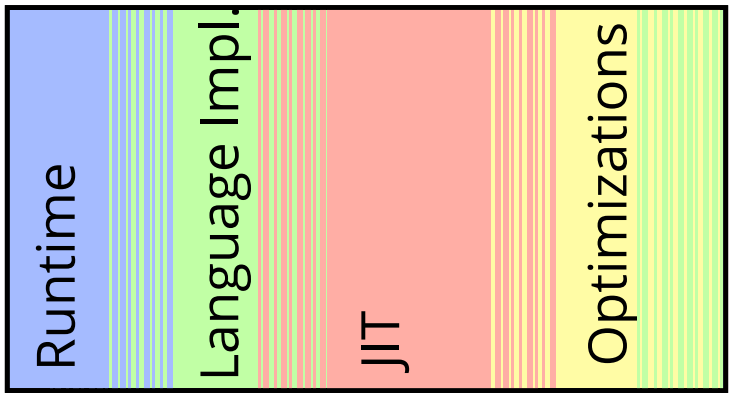
Domain: Language Virtual Machines

- Important goal of VM construction is performance
- A good JIT is needed
- Particularly for dynamically typed languages, where nothing is known statically



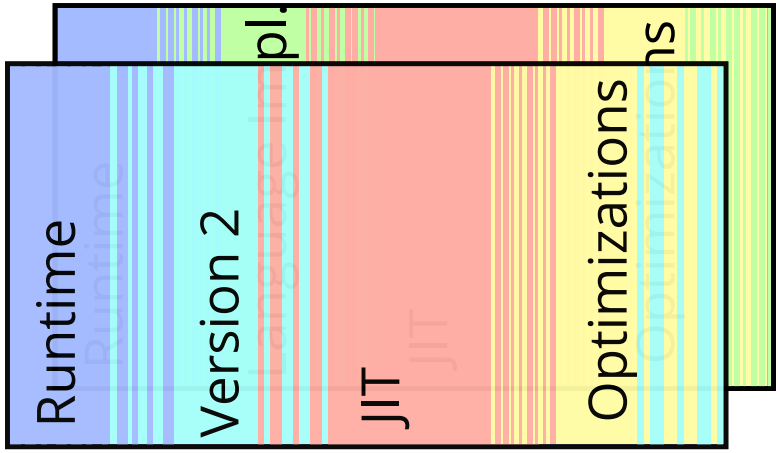
Problem: VMs have Monolithic Architecture

- JITs are complex engineering artifacts
- Architecture is often very complicated, with different concerns tangled up
- Changing the language is a lot of effort
- Very hard to share infrastructure between VMs for different languages



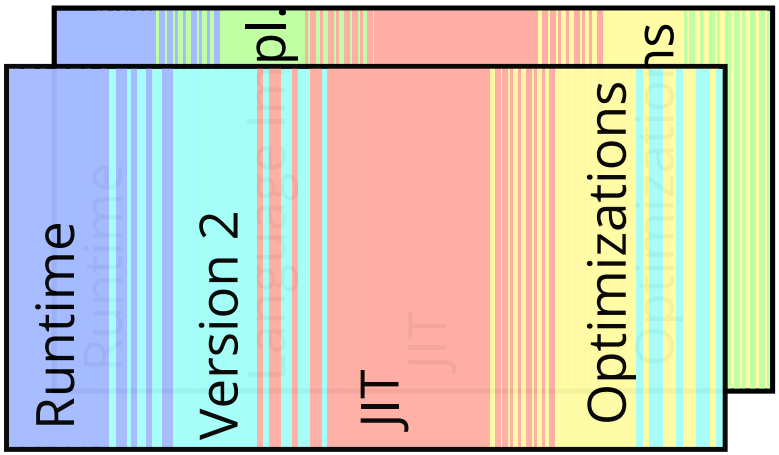
Problem: VMs have Monolithic Architecture

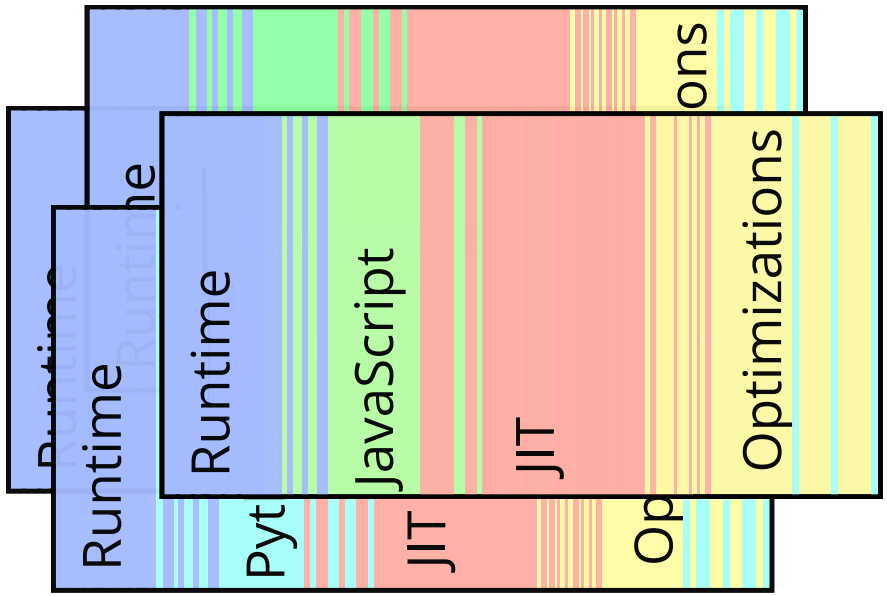
- JITs are complex engineering artifacts
- Architecture is often very complicated, with different concerns tangled up
- Changing the language is a lot of effort
- Very hard to share infrastructure between VMs for different languages



Problem: VMs have Monolithic Architecture

- JITs are complex engineering artifacts
- Architecture is often very complicated, with different concerns tangled up
- Changing the language is a lot of effort
- Very hard to share infrastructure between VMs for different languages





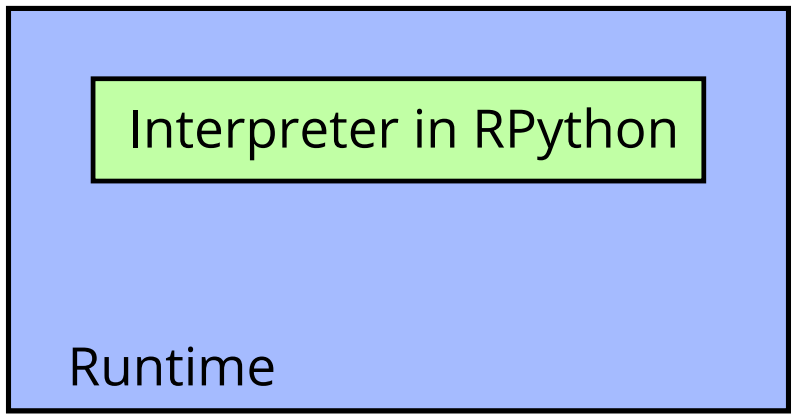
Approach: Separation of Concerns

Separate the following VM implementation concerns:

- Language semantics
- Generic JIT compilation issues
- Generic optimizations
- Language-specific optimizations

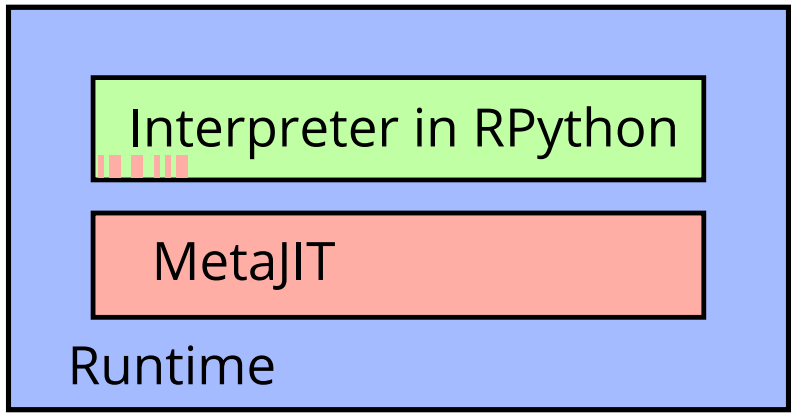
The RPython Project and PyPy

- RPython is a language to implement interpreters
- Interpreters are translated into C-based VMs
- Various extra features are added, e.g. GC
- Most mature interpreter is PyPy: Python in RPython
- Long-running project, many contributors



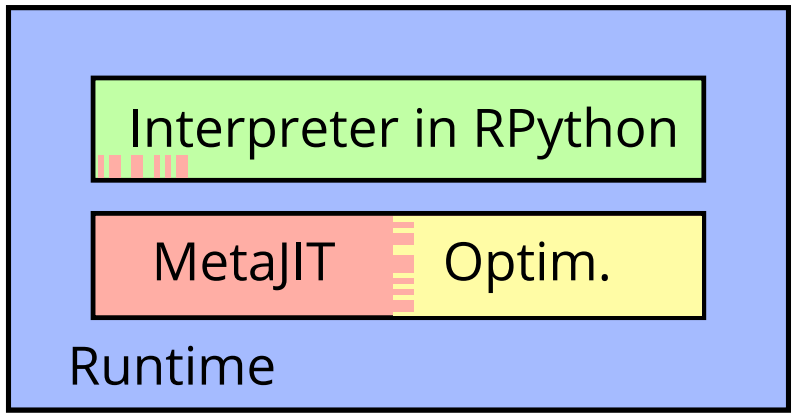
A Meta-Tracing JIT for RPython

- Apply the meta-tracing approach to RPython
- Insert meta JIT into the generated VM
- Contains generic JIT infrastructure: backends, integration, GC
- Needs some hints from the interpreter author
- Hints specify the main dispatch loop and the program counter



A Meta-Tracing JIT for RPython

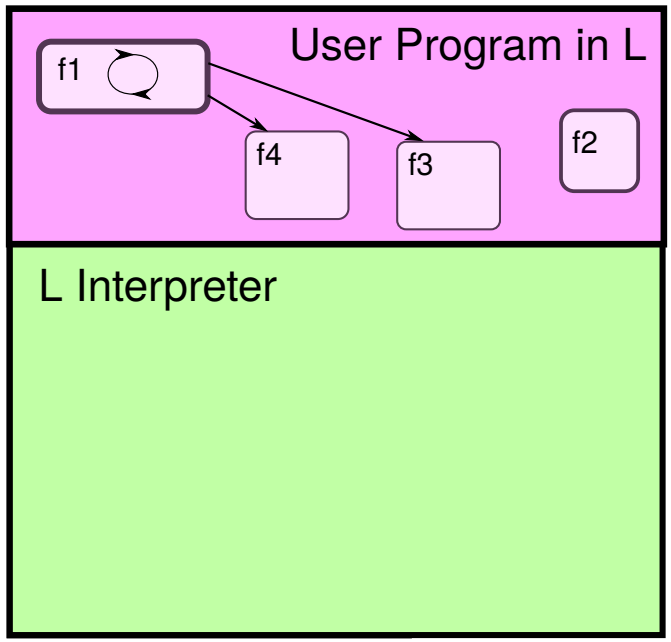
- Apply the meta-tracing approach to RPython
- Insert meta JIT into the generated VM
- Contains generic JIT infrastructure: backends, integration, GC
- Needs some hints from the interpreter author
- Hints specify the main dispatch loop and the program counter
- Typical optimizations, plus some specific ones



Meta-Tracing

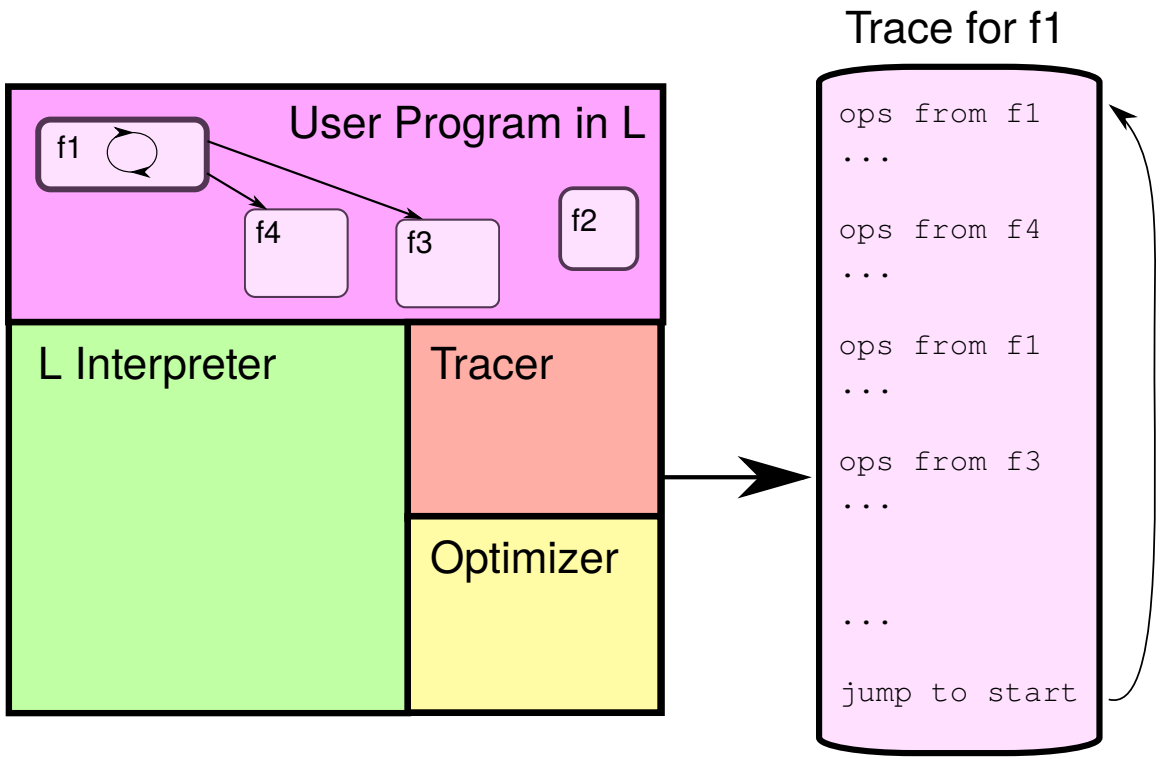
Tracing JITs

- Dream: add a simple JIT component to an interpreter
- Starts out interpreting and focuses on loops.
- Tracer records activity of interpreter for important loops.
- Conditions are turned into guards.



Tracing JITs

- Dream: add a simple JIT component to an interpreter
- Starts out interpreting and focuses on loops.
- Tracer records activity of interpreter for important.
- Conditions are turned into guards.



User program (lang *FL*)

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

Tracing JITs

User program (lang *FL*)	Trace when x is set to 6

`if x < 0:`
`  x = x + 1`
`else:`
`  x = x + 2`
`x = x + 3`
	`guard_type(x, int)`
	`guard_type(0, int)`
	`guard_not_less_than(x, 0)`
	`guard_type(x, int)`
	`guard_type(2, int)`
	`x = int_add(x, 2)`
	`guard_type(x, int)`
	`guard_type(3, int)`
	`x = int_add(x, 3)`

Tracing JITs

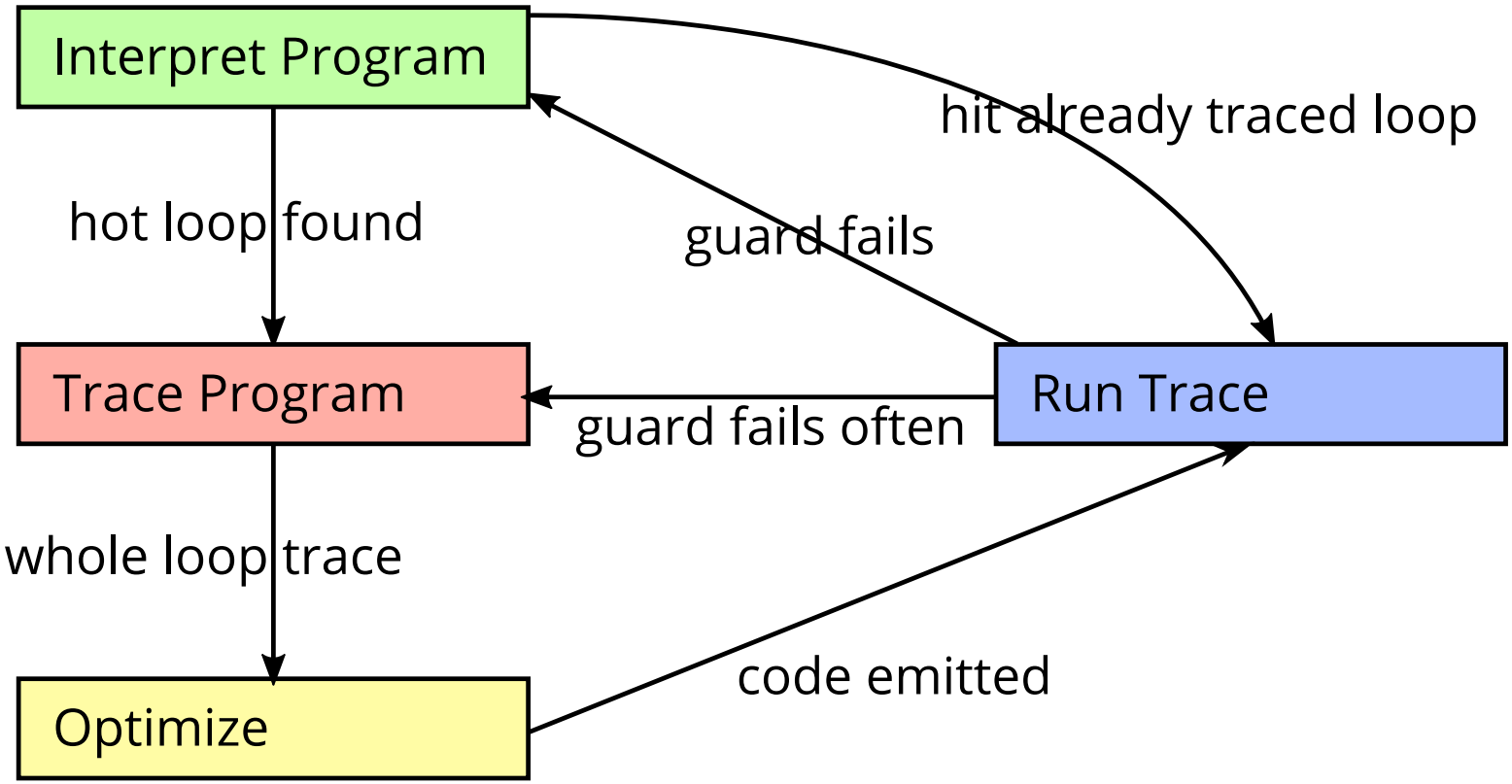
User program (lang *FL*)	Optimised trace

`if x < 0:`
`  x = x + 1`
`else:`
`  x = x + 2`
`x = x + 3`
	`guard_type(x, int)`
	`guard_not_less_than(x, 0)`
	`x = int_add(x, 5)`

Guards

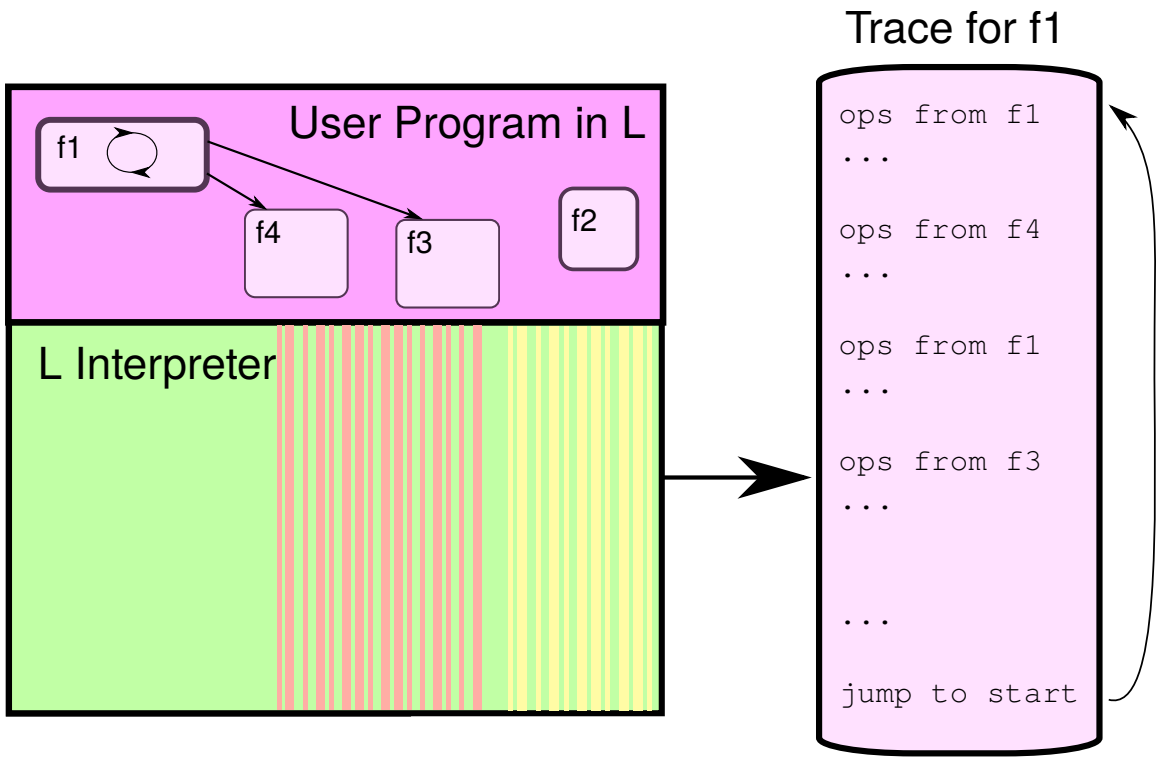
- Conditions are turned into guards.
- They check that the same control flow is followed.
- When they fail, go back to interpretation.
- Side traces are attached to commonly failing guards.
- Tracing works best if subsequent iterations of a loop follow the same control flow.

State Transitions Tracing JIT



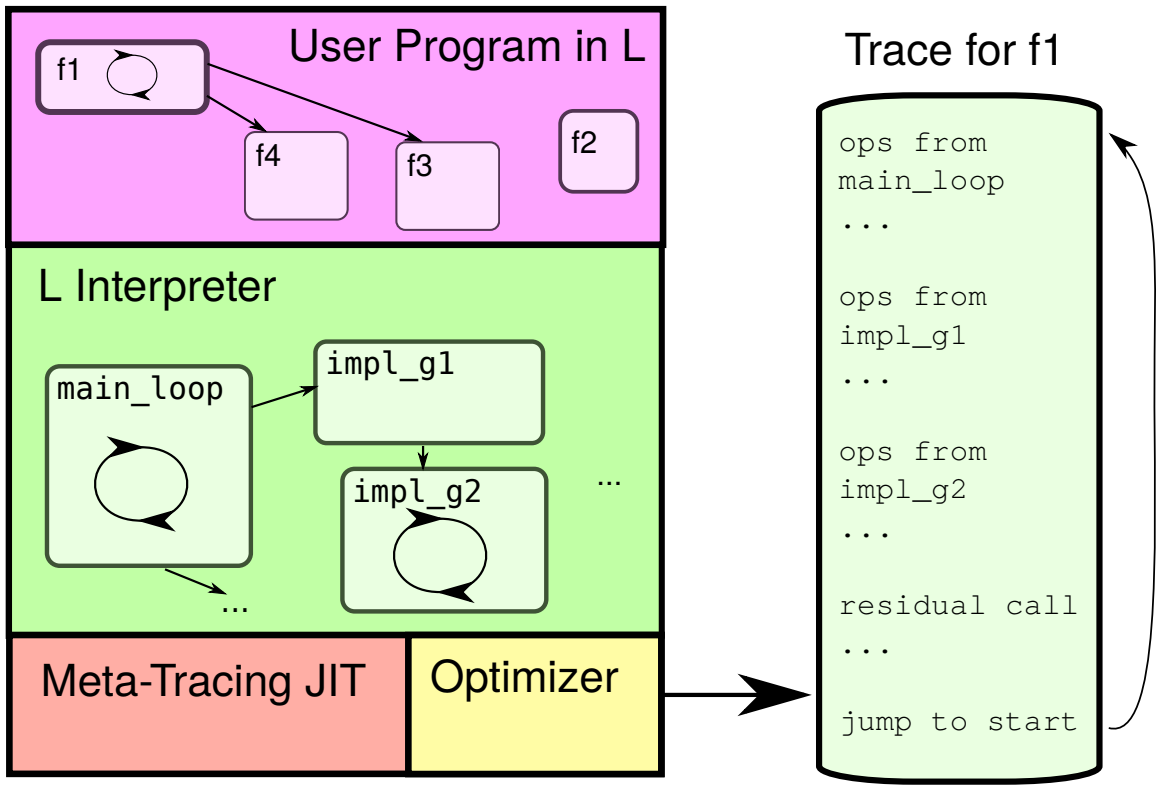
In Practice

- Components end up tangled and messy
- Many versions not solved
- Still need to start from scratch for every language



Meta-Tracing

- Solution: Trace the interpreter, not the program.
- Interpreters are big loops with complex control flow.



Adding a JIT to an RPython interpreter

```
...
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 ...:
        ...
```

Observation: interpreters are big loops.

Adding a JIT to an RPython interpreter

```
...
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 ...:
        ...
```

Observation: interpreters are big loops.

Meta-Tracing

- A lot of the interpreter data structure manipulations are optimized away.
- Examples: Stack manipulation etc.
- Some technical challenges, but separation works well.

Generic Optimizations

- Typical compiler optimizations
- Easy to implement, because of traces

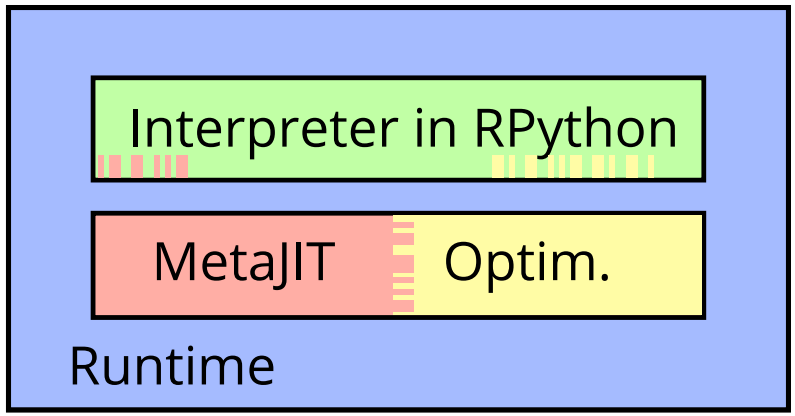
Generic Optimizations

- Typical compiler optimizations
- Easy to implement, because of traces
- Interesting new one: allocation removal
- Dynamic languages allocate a lot of objects, e.g. for primitive boxes
- Objects often have limited predetermined lifetime
- a + b * c
- Remove intermediate allocations in traces

Runtime Feedback

Language-Specific Runtime Feedback

- JIT gets its power by observing the running program
- Bare meta-tracing does not know any details of the language implemented
- Language implementer can provide extra information with more hints
- These often express how the language is typically used
- Such information is only implicit or absent from the interpreter source



Example: Language-Specific Runtime Feedback

At a method callsite, the called method is often always the same:

```
def lookup(cls, methname):
    ...

def send(obj, messagename, arguments):
    cls = obj.getclass()

    meth = lookup(cls, messagename)
    return meth.call(obj, arguments)
```

Example: Language-Specific Runtime Feedback

At a method callsite, the called method is often always the same:

```
def lookup(cls, methname):
    ...

def send(obj, messagename, arguments):
    cls = obj.getclass()
    promote(cls)
    meth = lookup(cls, messagename)
    return meth.call(obj, arguments)
```

Example: Language-Specific Runtime Feedback

At a method callsite, the called method is often always the same:

```
@elidable
def lookup(cls, methname):
    ...

def send(obj, messagename, arguments):
    cls = obj.getclass()
    promote(cls)
    meth = lookup(cls, messagename)
    return meth.call(obj, arguments)
```

Case Studies

PyPy, an RPython VM for Python 

- How the RPython project got started.
- Very compatible and fast implementation of Python 2.7.
- Big developer community with various interests.
- Around 60K LoC (interpreter) and 190K LoC (modules) of RPython code.

Hints in PyPy

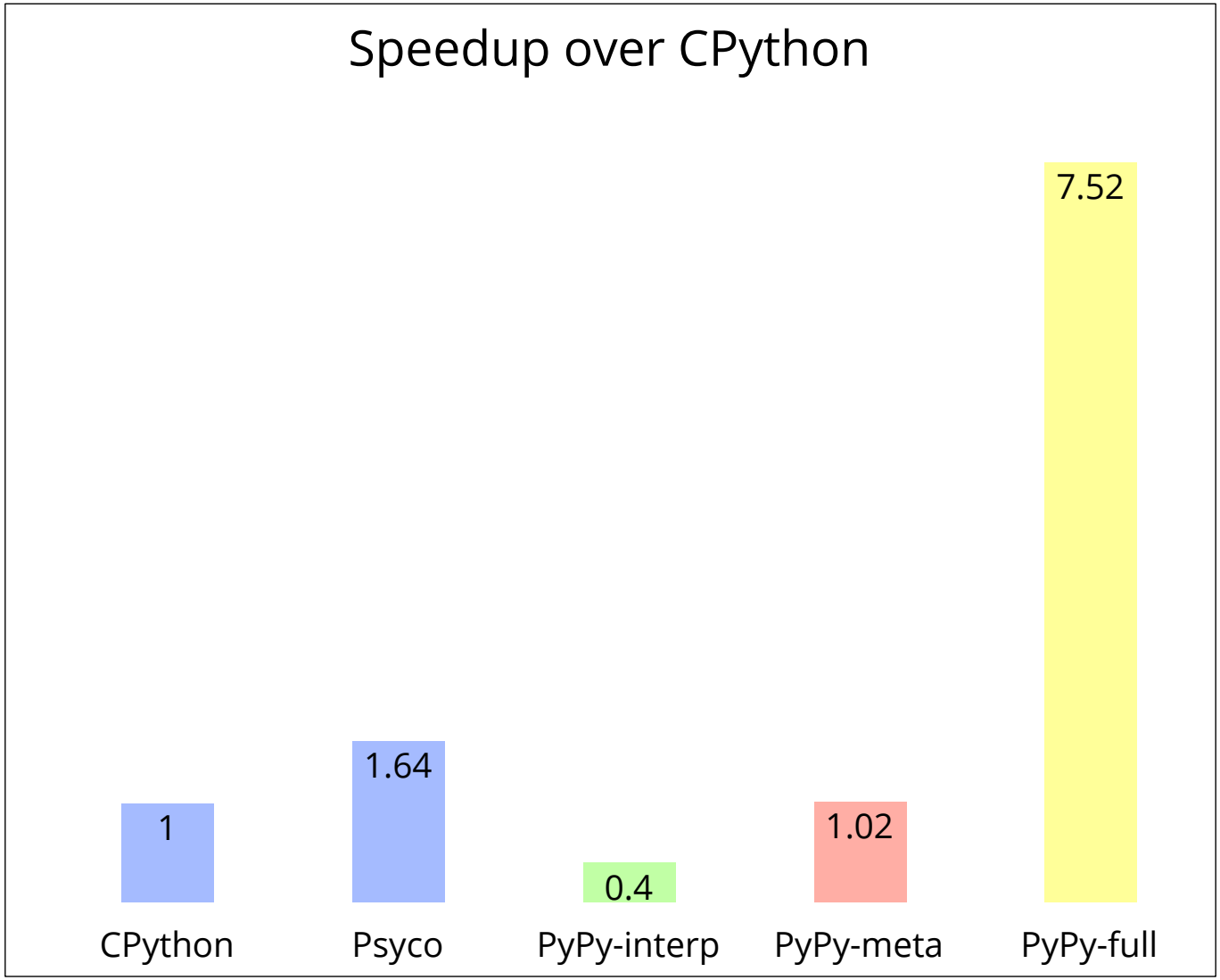
- About 400 hints in the interpreter.
- Continuous process to add more.
- Mostly concerned with core features of the language.
  - Global lookups
  - Method calls
  - Attribute reads
  - Data structures (e.g. lists and sets)

Example: Attribute Reads in Python

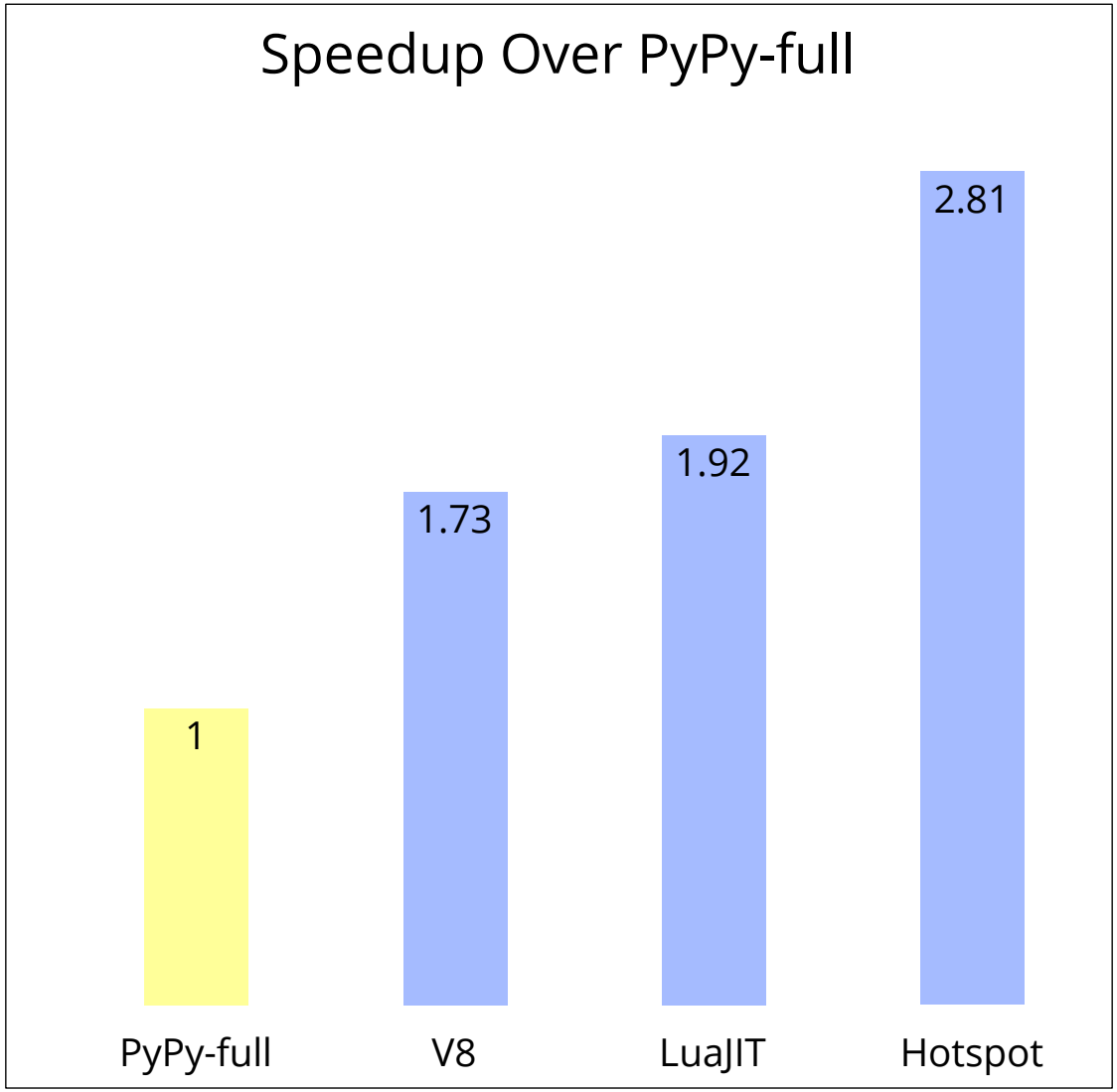
What happens when an attribute `x.m` is read? (simplified)

- check for the presence of `x.__getattribute__`, if there, call it
- look for the name of the attribute in the object's dictionary, if it's there, return it
- walk up the MRO of the object and look in each class' dictionary for the attribute
- if the attribute is found, call its `__get__` attribute and return the result
- if the attribute is not found, look for `x.__getattr__`, if there, call it
- raise an `AttributeError`

Python Benchmarks



Shootout Benchmarks

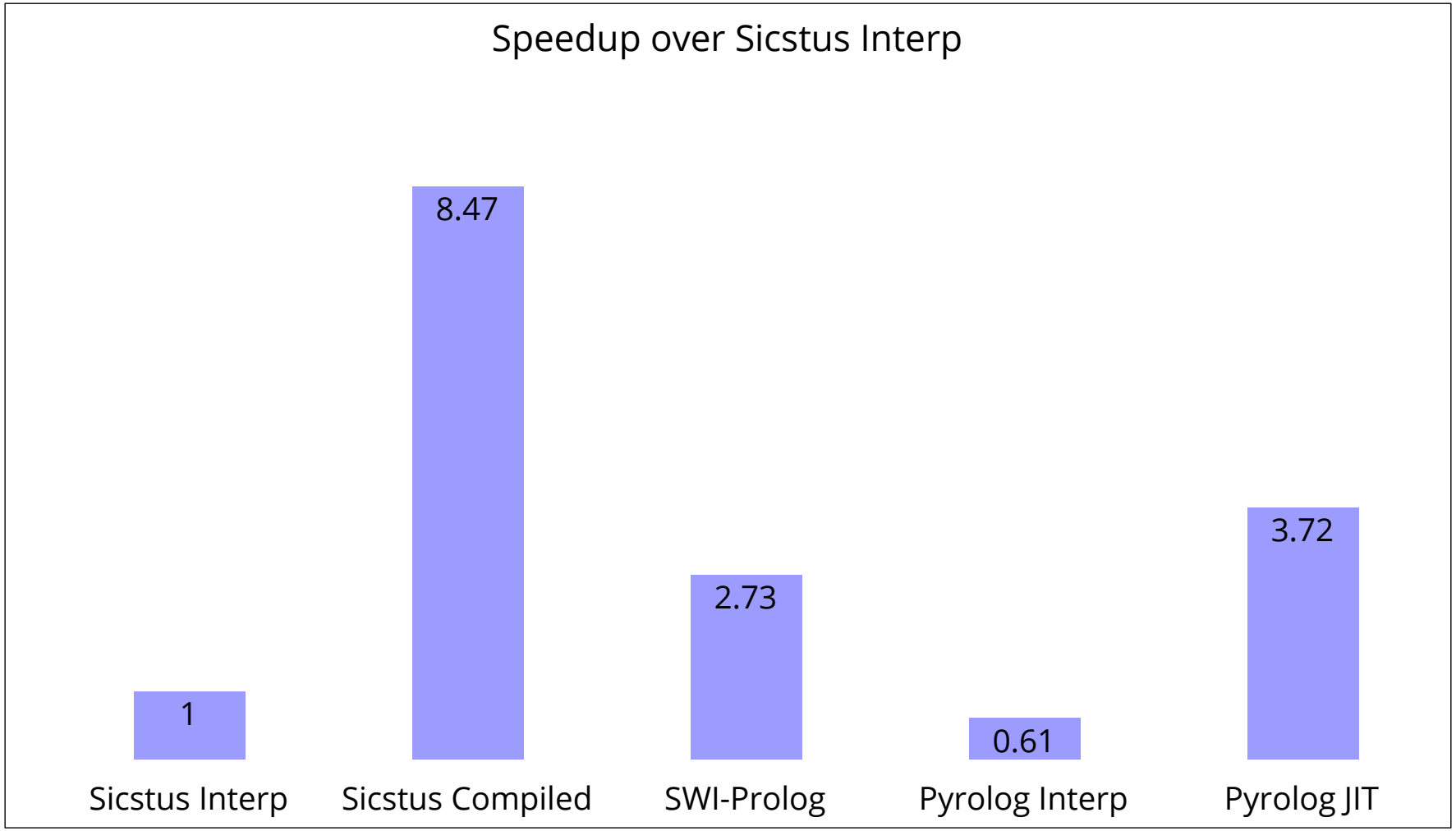


Demo

Pyrolog

- A Prolog interpreter in RPython
- Very different execution model than Python
- Still decent performance improvements
- About 15K LoC of RPython code
- 70 hints

Prolog Benchmarks



Summary

- Using the RPython JIT, most of the JIT infrastructure is shared between languages
- Only an interpreter and some hints are needed
- Can support languages as different as Prolog and Python
- Other languages: PHP, Ruby, Smalltalk, Racket, SQLite, CPU emulators, ...
- “Gives you 80% of a great JIT 20% of the effort”