Meta-Tracing in the PyPy Project

Carl Friedrich Bolz[1] Antonio Cuni[1] Maciej Fijałkowski[2]
Michael Leuschel[1] Samuele Pedroni[3] Armin Rigo[1]
many more

[1]Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany

[2]merlinux GmbH, Hildesheim, Germany

[3]Canonical

Foundations of Scription Languages, Dagstuhl, 5th January 2012

Good JIT Compilers for Scripting Languages are Hard

- recent languages like Python, Ruby, JS, PHP have complex core semantics
- many corner cases, even hard to interpret correctly
- particularly in contexts where you have limited resources (like academic, Open Source)

Good JIT Compilers for Scripting Languages are Hard

- recent languages like Python, Ruby, JS, PHP have complex core semantics
- many corner cases, even hard to interpret correctly
- particularly in contexts where you have limited resources (like academic, Open Source)

Problems

1. implement all corner-cases of semantics correctly
2. ... and the common cases efficiently
3. while maintaing reasonable simplicity in the implementation

Example: Attribute Reads in Python

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

Example: Attribute Reads in Python

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

- check for `x.__getattribute__`, if there, call it

Example: Attribute Reads in Python

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

- check for `x.__getattribute__`, if there, call it
- look for the attribute in the object's dictionary, if it's there, return it

Example: Attribute Reads in Python

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

- check for `x.__getattribute__`, if there, call it
- look for the attribute in the object's dictionary, if it's there, return it
- walk up the MRO and look in each class' dictionary for the attribute

Example: Attribute Reads in Python

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

- check for `x.__getattribute__`, if there, call it
- look for the attribute in the object's dictionary, if it's there, return it
- walk up the MRO and look in each class' dictionary for the attribute
- if the attribute is found, call its `__get__` attribute and return the result

Example: Attribute Reads in Python

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

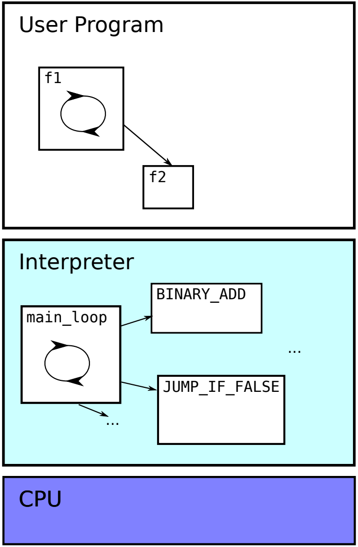
- check for `x.__getattribute__`, if there, call it
- look for the attribute in the object's dictionary, if it's there, return it
- walk up the MRO and look in each class' dictionary for the attribute
- if the attribute is found, call its `__get__` attribute and return the result
- if the attribute is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

Example: Attribute Reads in Python

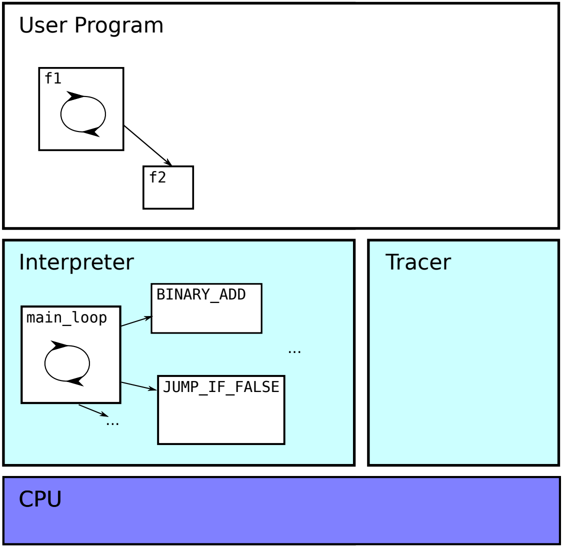
What happens when an attribute `x.m` is read? (simplified)

- check for `x.__getattribute__`, if there, call it
- look for the attribute in the object's dictionary, if it's there, return it
- walk up the MRO 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`

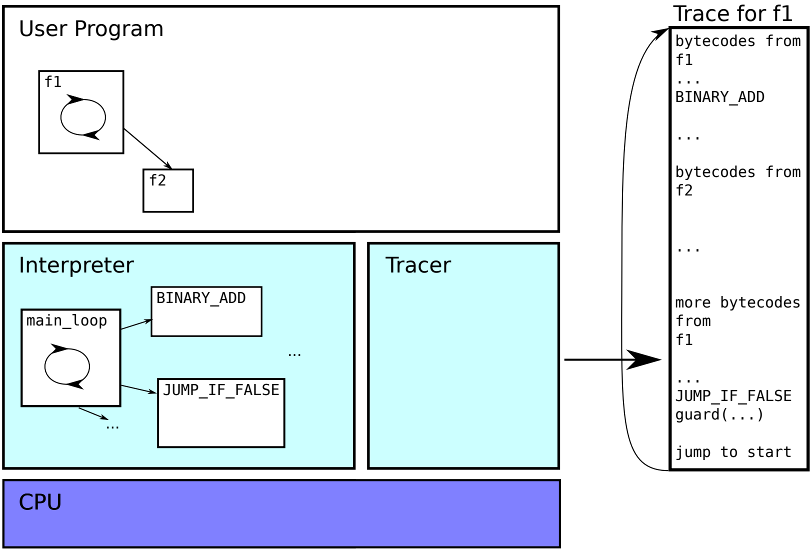
An Interpreter



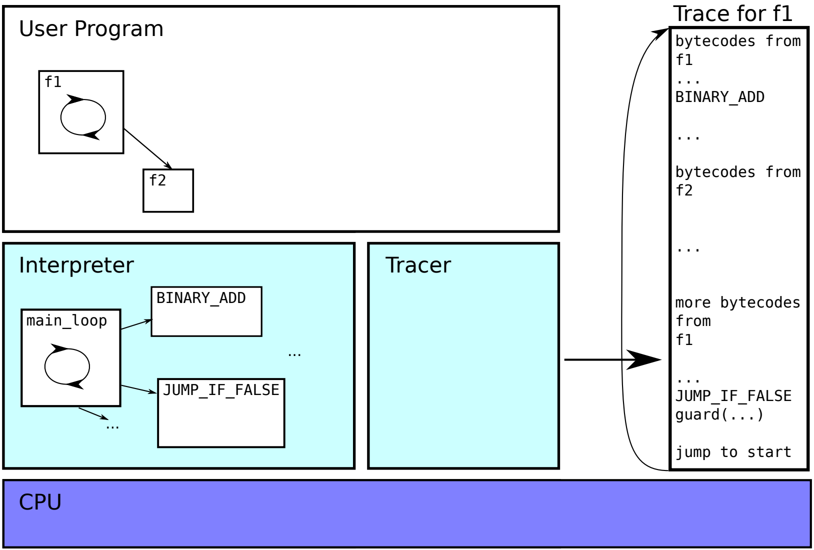
A Tracing JIT



A Tracing JIT



A Tracing JIT



Tracing JITs

Advantages:

- can be added to existing VM
- interpreter does a lot of work
- can fall back to interpreter for uncommon paths

Tracing JITs

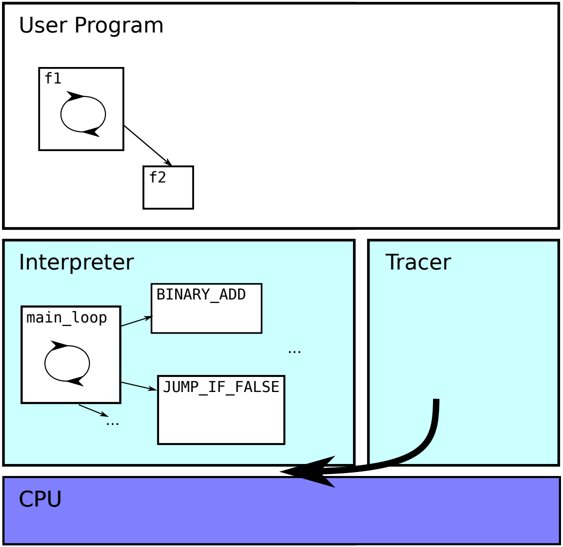
Advantages:

- can be added to existing VM
- interpreter does a lot of work
- can fall back to interpreter for uncommon paths

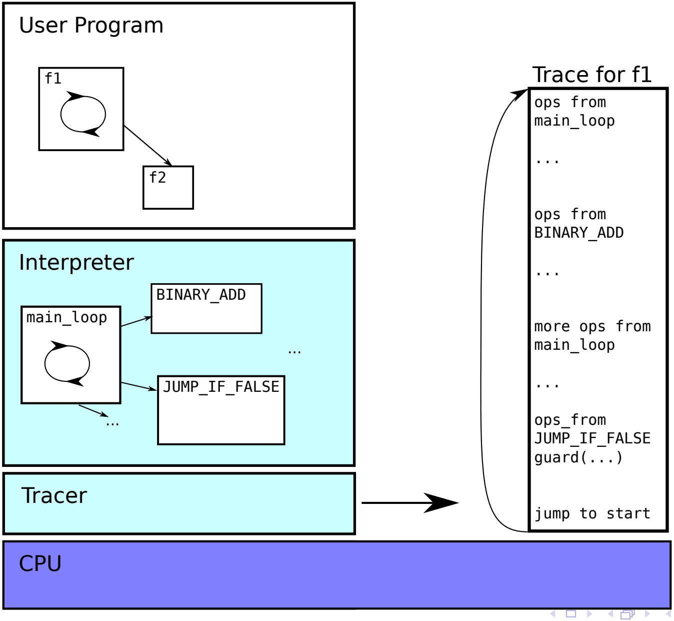
Problems

- traces typically contain bytecodes
- many scripting languages have bytecodes that contain complex logic
- need to expand the bytecode in the trace into something more explicit
- this duplicates the language semantics in the tracer/optimizer

Idea of Meta-Tracing



Meta-Tracing



User Program

f1

f2

Interpreter

main_loop

BINARY_ADD

JUMP_IF_FALSE

...

...

Tracer

CPU

Trace for f1

```
ops from
main_loop

...

ops from
BINARY_ADD

...

more ops from
main_loop

...

ops_from
JUMP_IF_FALSE
guard(...)

jump to start
```

Meta-Tracing JITs

Advantages:

- semantics are always like that of the interpreter
- trace fully contains language semantics
- meta-tracers can be reused for various interpreters

Meta-Tracing JITs

Advantages:

- semantics are always like that of the interpreter
- trace fully contains language semantics
- meta-tracers can be reused for various interpreters

a few meta-tracing systems have been built:

- Sullivan et.al. describe a meta-tracer using the Dynamo RIO system
- Yermolovich et.al. run a Lua implementation on top of a tracing JS implementation
- SPUR is a tracing JIT for CLR bytecodes, which is used to speed up a JS implementation in C#

PyPy

A general environment for implementing scripting languages

PyPy

A general environment for implementing scripting languages

Approach

- write an interpreter for the language in RPython
- compilable to an efficient C-based VM

PyPy

A general environment for implementing scripting languages

Approach

- write an interpreter for the language in RPython
- compilable to an efficient C-based VM
- (RPython is a restricted subset of Python)

PyPy

A general environment for implementing scripting languages

Approach

- write an interpreter for the language in RPython
- compilable to an efficient C-based VM
- (RPython is a restricted subset of Python)

PyPy's Meta-Tracing JIT

- PyPy contains a meta-tracing JIT for interpreters in RPython
- needs a few source-code hints (or annotations) in the interpreter
- allows interpreter-author to express language specific type feedback
- contains powerful general optimizations

PyPy's Meta-Tracing JIT

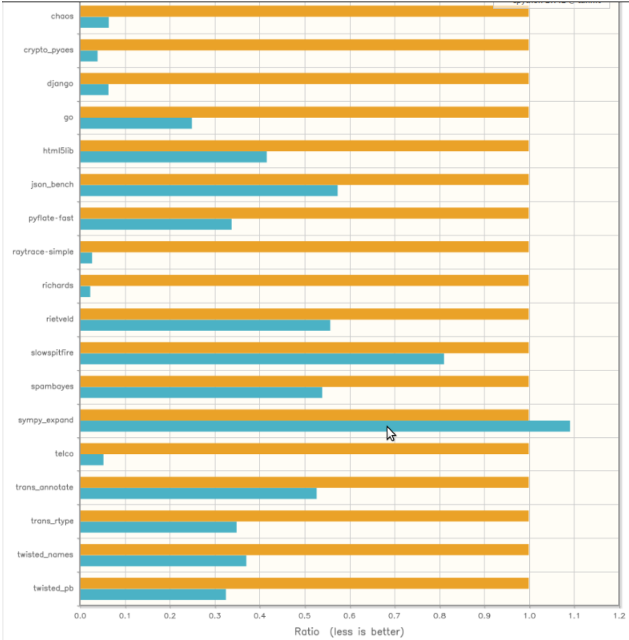
- PyPy contains a meta-tracing JIT for interpreters in RPython
- needs a few source-code hints (or annotations) in the interpreter
- allows interpreter-author to express language specific type feedback
- contains powerful general optimizations
- general techniques to deal with reified frames

Language Implementations Done with PyPy

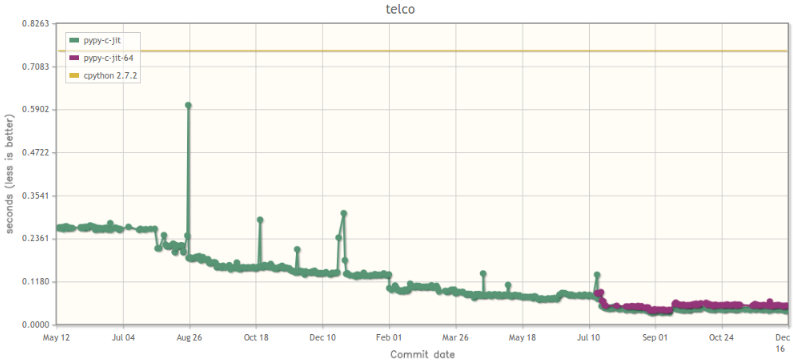
- Most complete language implemented: Python
- regular expression matcher of Python standard library
- A reasonably complete Prolog
- Converge (previous talk)
- lots of experiments (Squeak, Gameboy emulator, JS, start of a PHP, Haskell, ...)

Some Benchmarks for Python

- benchmarks done using PyPy's Python interpreter
- about 30'000 lines of code


chaos
crypto_pyaes
django
go
html5lib
json_bench
pyflate-fast
raytrace-simple
richards
rietveld
slowspitfire
spambayes
sympy_expand
telco
trans_annotate
trans_rtype
twisted_names
twisted_pb
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1
1.2
Ratio (less is better)

Telco Benchmark



telco: A small program which is intended to capture the essence of a telephone company billing application, with a realistic balance between Input/Output activity and application calculations.

Conclusion

- writing good JITs for recent scripting languages is too hard!
- only reasonable if the language is exceptionally simple
- or if somebody has a lot of money
- PyPy is one point in a large design space of meta-solutions
- uses tracing on the level of the interpreter (meta-tracing) to get speed

Conclusion

- writing good JITs for recent scripting languages is too hard!
- only reasonable if the language is exceptionally simple
- or if somebody has a lot of money
- PyPy is one point in a large design space of meta-solutions
- uses tracing on the level of the interpreter (meta-tracing) to get speed
- **In a way, the exact approach is not too important: let's write more meta-tools!**

Thank you! Questions?

- writing good JITs for recent scripting languages is too hard!
- only reasonable if the language is exceptionally simple
- or if somebody has a lot of money
- PyPy is one point in a large design space of meta-solutions
- uses tracing on the level of the interpreter (meta-tracing) to get speed
- **In a way, the exact approach is not too important: let's write more meta-tools!**

Possible Further Slides

- Getting Meta-Tracing to Work
- Language-Specific Runtime Feedback
- Powerful General Optimizations
- Optimizing Reified Frames
- Which Languages Can Meta-Tracing be Used With?
- Using OO VMs as an implementation substrate
- Comparison with Partial Evaluation

Getting Meta-Tracing to Work

- Interpreter author needs add some hints to the interpreter
- one hint to identify the bytecode dispatch loop
- one hint to identify the jump bytecode
- with these in place, meta-tracing works
- but produces non-optimal code

Language-Specific Runtime Feedback

Problems of Naive Meta-Tracing:

- user-level types are normal instances on the implementation level
- thus no runtime feedback of user-level types
- tracer does not know about invariants in the interpreter

Language-Specific Runtime Feedback

Problems of Naive Meta-Tracing:

- user-level types are normal instances on the implementation level
- thus no runtime feedback of user-level types
- tracer does not know about invariants in the interpreter

Solution in PyPy

- introduce more hints that the interpreter-author can use
- hints are annotation in the interpreter
- they give information to the meta-tracer

Language-Specific Runtime Feedback

Problems of Naive Meta-Tracing:

- user-level types are normal instances on the implementation level
- thus no runtime feedback of user-level types
- tracer does not know about invariants in the interpreter

Solution in PyPy

- introduce more hints that the interpreter-author can use
- hints are annotation in the interpreter
- they give information to the meta-tracer
- one to induce runtime feedback of arbitrary information (typically types)
- the second one to influence constant folding

Powerful General Optimizations

- Very powerful general optimizations on traces

Powerful General Optimizations

- Very powerful general optimizations on traces

Heap Optimizations

- escape analysis/allocation removal
- remove short-lived objects
- gets rid of the overhead of boxing primitive types
- also reduces overhead of constant heap accesses

Optimizing Reified Frames

- Common problem in scripting languages
- frames are reified in the language, i.e. can be accessed via reflection
- used to implement the debugger in the language itself
- or for more advanced usecases (backtracking in Smalltalk)
- when using a JIT, quite expensive to keep them up-to-date

Optimizing Reified Frames

- Common problem in scripting languages
- frames are reified in the language, i.e. can be accessed via reflection
- used to implement the debugger in the language itself
- or for more advanced usecases (backtracking in Smalltalk)
- when using a JIT, quite expensive to keep them up-to-date

Solution in PyPy

- General mechanism for updating reified frames lazily
- use deoptimization when frame objects are accessed by the program
- interpreter just needs to mark the frame class

Bonus: Which Languages Can Meta-Tracing be Used With?

- To make meta-tracing useful, there needs to be some kind of runtime variability
- that means it definitely works for all dynamically typed languages
- ... but also for other languages with polymorphism that is not resolvable at compile time
- most languages that have any kind of runtime work

Bonus: Using OO VMs as an implementation substrate

Benefits

- higher level of implementation
- the VM supplies a GC and mostly a JIT
- better interoperability than what the C level provides
- `invokedynamic` should make it possible to get language-specific runtime feedback

Bonus: Using OO VMs as an implementation substrate

Benefits

- higher level of implementation
- the VM supplies a GC and mostly a JIT
- better interoperability than what the C level provides
- `invokedynamic` should make it possible to get language-specific runtime feedback

Problems

- can be hard to map concepts of the scripting language to the host OO VM
- performance is often not improved, and can be very bad, because of this semantic mismatch
- getting good performance needs a huge amount of tweaking
- tools not really prepared to deal with people that care about the shape of the generated assembler

Bonus: Using OO VMs as an implementation substrate

Benefits

- higher level of implementation
- the VM supplies a GC and mostly a JIT
- better interoperability than what the C level provides
- `invokedynamic` should make it possible to get language-specific runtime feedback

Problems

- can be hard to map concepts of the scripting language to the host OO VM
- performance is often not improved, and can be very bad, because of this semantic mismatch
- getting good performance needs a huge amount of tweaking
- tools not really prepared to deal with people that care about the shape of the generated assembler

Bonus: Comparison with Partial Evaluation

Bonus: Comparison with Partial Evaluation

- the only difference between meta-tracing and partial evaluation is that meta-tracing works

Bonus: Comparison with Partial Evaluation

- the only difference between meta-tracing and partial evaluation is that meta-tracing works
- ... mostly kidding

Bonus: Comparison with Partial Evaluation

- the only difference between meta-tracing and partial evaluation is that meta-tracing works
- ... mostly kidding
- very similar from the motivation and ideas
- PE was never scaled up to perform well on large interpreters
- classical PE mostly ahead of time
- PE tried very carefully to select the right paths to inline and optimize
- quite often this fails and inlines too much or too little
- tracing is much more pragmatic: simply look what happens