Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages

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

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

[2]merlinux GmbH, Hildesheim, Germany

[3]Open End, Göteborg, Sweden

ICOOOLPS 2011, July 26, 2011

Good JIT Compilers for Dynamic Languages are Hard

- recent languages like Python, Ruby, JS have complex core semantics
- many corner cases, even hard to interpret correctly

Good JIT Compilers for Dynamic Languages are Hard

- recent languages like Python, Ruby, JS have complex core semantics
- many corner cases, even hard to interpret correctly
- feedback of runtime information to the compiler is necessary
- correct exploitation of this information in the compiler

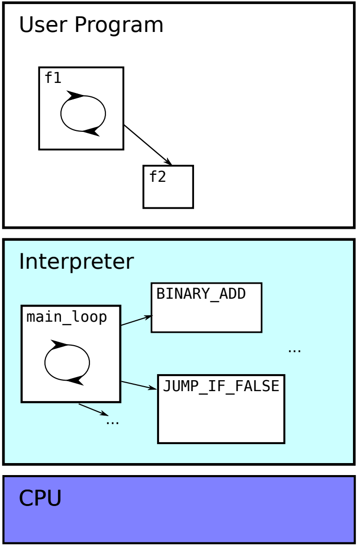
Good JIT Compilers for Dynamic Languages are Hard

- recent languages like Python, Ruby, JS have complex core semantics
- many corner cases, even hard to interpret correctly
- feedback of runtime information to the compiler is necessary
- correct exploitation of this information in the compiler

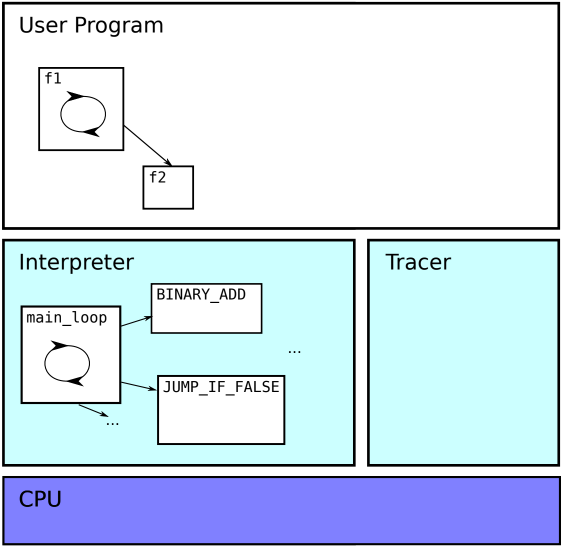
Problems

1. implement all corner-cases of semantics correctly
2. ... and the common cases efficiently
3. feed back and exploit runtime information

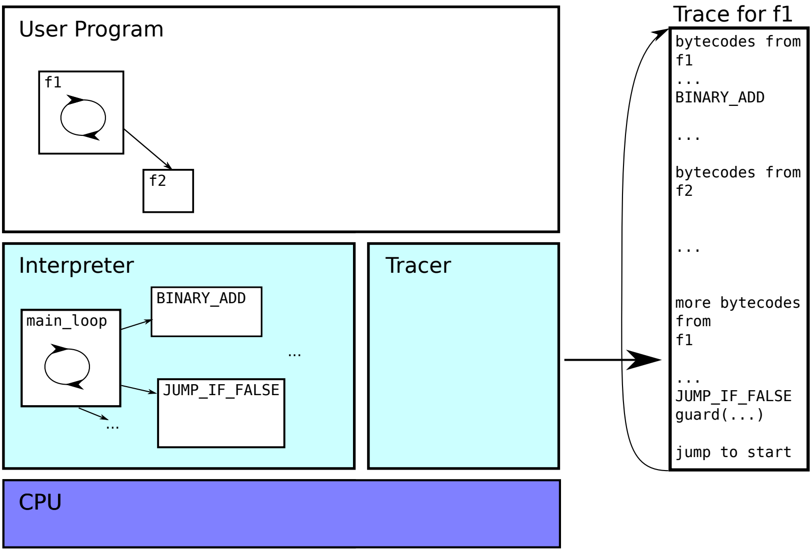
An Interpreter



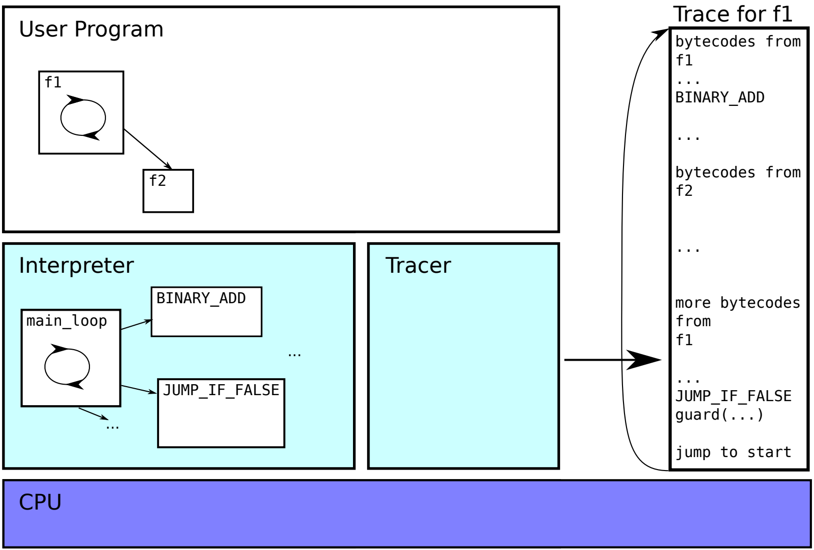
A Tracing JIT



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

Granularity Problems

- if the tracer records bytecode, not enough information is there
- many dynamic languages have bytecodes that contain complex logic
- need to expand the bytecode in the trace into something more explicit
- this duplicates the lanuage semantics in the tracer/optimizer

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`

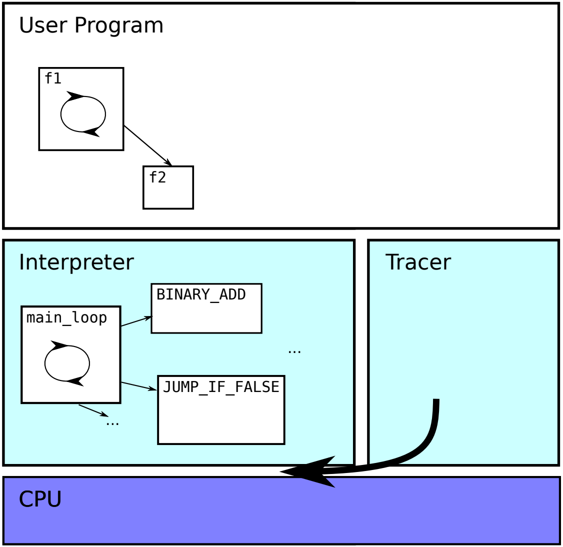
Example: Attribute Reads in Python

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

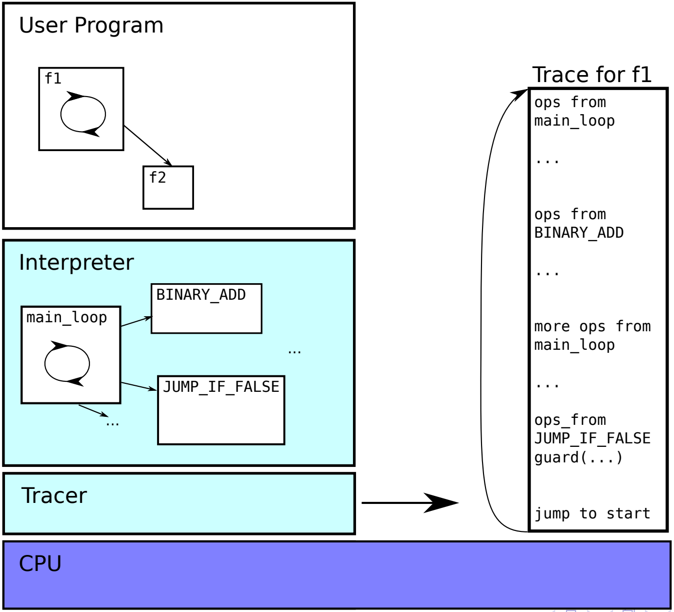
- check for `x.__getattribute__`, if there, call it
- look for the attribute in the object's dictionary, if it's there, return it
- walk up the MRO and look in each class' dictionary for the attribute
- if the attribute is found, call its `__get__` attribute and return the result
- if the attribute is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

all this is one bytecode

Idea of Meta-Tracing



Meta-Tracing



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 dynamic languages

PyPy

A general environment for implementing dynamic languages

Approach

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

PyPy

A general environment for implementing dynamic 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 dynamic 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) <u>in the interpreter</u>
- powerful general optimizations

Runtime Feedback

Problems of Naive Meta-Tracing:

- no runtime feedback of user-level types
- tracer does not know about invariants in the interpreter

Runtime Feedback

Problems of Naive Meta-Tracing:

- no runtime feedback of user-level types
- tracer does not know about invariants in the interpreter

Proposed Solutions

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

Runtime Feedback

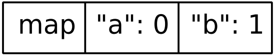
Problems of Naive Meta-Tracing:

- no runtime feedback of user-level types
- tracer does not know about invariants in the interpreter

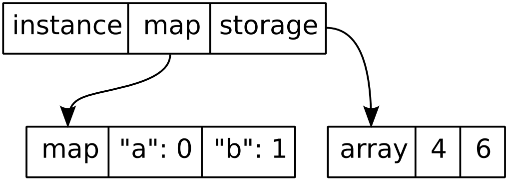
Proposed Solutions

- introduce *hints* that the interpreter-author can use
- hints are annotation in the interpreter
- they give information to the meta-tracer
- two hints presented here
- one to induce runtime feedback of arbitrary information
- the second one to influence constant folding

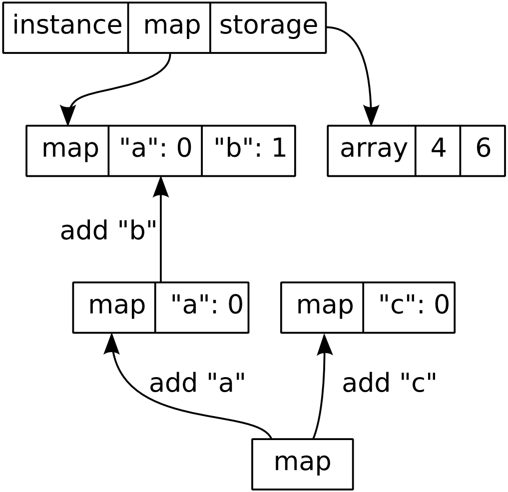
Example: Instances with Maps



Example: Instances with Maps



Example: Instances with Maps



Map Implementation

```
class Map(object):
    def __init__(self, indexes):
        self.indexes = indexes
        ...

    def getindex(self, name):
        return self.indexes.get(name, -1)

    def add_attribute(self, name):
        ...

EMPTY_MAP = Map({})
```

```python
class Instance(object):
    def __init__(self):
        self.map = EMPTY_MAP
        self.storage = []

    def getfield(self, name):
        index = self.map.getindex(name)
        if index != -1:
            return self.storage[index]
        return None

    def write_attribute(self, name, value):
        ...
```

Trace for Code `inst.a + inst.b`

```
# inst₁.getfield("a")
map₁ = inst₁.map
index₁ = Map.getindex(map₁, "a")
guard(index₁ != -1)
storage₁ = inst₁.storage
result₁ = storage₁[index₁]
```

Trace for Code `inst.a + inst.b`

```
# inst₁.getfield("a")
map₁ = inst₁.map
index₁ = Map.getindex(map₁, "a")
guard(index₁ != -1)
storage₁ = inst₁.storage
result₁ = storage₁[index₁]

# inst₁.getfield("b")
map₂ = inst₁.map
index₂ = Map.getindex(map₂, "b")
guard(index₂ != -1)
storage₂ = inst₁.storage
result₂ = storage₂[index₂]

v₁ = result₁ + result₂
```

Runtime Feedback Controlled by the Interpreter Author

- give the interpreter author a way to feed back runtime values into the trace
- written as `promote(x)`
- captures the argument's runtime value during tracing
- should be used only for variables that take few values

Tiny Example

```
def f1(x, y):
    promote(x)
    z = x * 2 + 1
    return z + y
```

guard(x_1 == 4)
v_1 = x_1 * 2
z_1 = v_1 + 1
v_2 = z_1 + y_1
return(v_2)

Foldable Operations Defined by the Interpreter Author

- let the interpreter author define foldable functions
- those functions typically don't look foldable
- otherwise there is no need for an annotation
- done via a function decorator `@elidable`

Foldable Operations Defined by the Interpreter Author

- let the interpreter author define foldable functions
- those functions typically don't look foldable
- otherwise there is no need for an annotation
- done via a function decorator `@elidable`
- decorated functions should be pure
- or have idempotent side effects (such as a function that memoizes)
- trace optimizer will remove calls to such functions with constant arguments

Adding Hints to Maps

```
class Map(object):
    def __init__(self, indexes):
        self.indexes = indexes
        ...

    @elidable
    def getindex(self, name):
        return self.indexes.get(name, -1)

    def add_attribute(self, name):
        ...

EMPTY_MAP = Map({})
```

Adding Hints to Maps

```
class Instance(object):
    def __init__(self):
        self.map = EMPTY_MAP
        self.storage = []

    def getfield(self, name):
        promote(self.map)
        index = self.map.getindex(name)
        if index != -1:
            return self.storage[index]
        return None

    def write_attribute(self, name, value):
        ...
```

```
# inst_1.getfield("a")
map_1 = inst_1.map
guard(map_1 == 0xb74af4a8)
index_1 = Map.getindex(map_1, "a")
guard(index_1 != -1)
storage_1 = inst_1.storage
result_1 = storage_1[index_1}]

# inst_1.getfield("b")
map_2 = inst_1.map
guard(map_2 == 0xb74af4a8)
index_2 = Map.getindex(map_2, "b")
guard(index_2 != -1)
storage_2 = inst_1.storage
result_2 = storage_2[index_2]

v_1 = result_1 + result_2
```

Final Trace

$$
\begin{aligned}
&map_1 = inst_1\texttt{.map}\\
&\texttt{guard(}map_1\ \texttt{==}\ \texttt{0xb74af4a8)}\\
&storage_1 = inst_1\texttt{.storage}\\
&result_1 = storage_1\texttt{[0]}\\
&result_2 = storage_2\texttt{[1]}\\
&v_1 = result_1 + result_2
\end{aligned}
$$

Uses of These Hints

`promote` lets one specialize on various things:

- user-level types
- shapes of instances
- the current state of a classes' methods
- ...

Uses of These Hints

`promote` lets one specialize on various things:

- user-level types
- shapes of instances
- the current state of a classes' methods
- ...

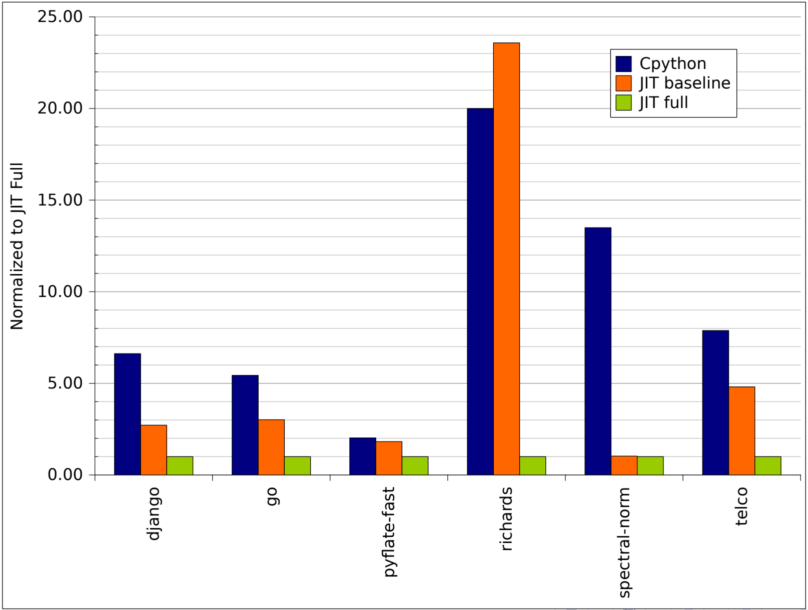
uses of `@elidable`

- define immutable fields by decorating a getter
- declare arbitrary invariants

Some Benchmarks

- benchmarks done using PyPy's Python interpreter
- about 30'000 lines of code
- 20 calls to `promote`
- 10 applications of `@elidable`

Some Benchmarks



Conclusion

- meta-tracing can make the efficient implementation of complex dynamic languages easier
- only requires to write a correct interpreter
- two kinds of hints to be added by the interpreter author allow arbitrary runtime feedback and its exploitation
- the hints are expressive enough to re-implement classical optimizations such as maps
- usage of the hints leads to good speedups for object-oriented code in PyPy's Python interpreter

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