PyPy: becoming fast

Antonio Cuni
Carl Friedrich Bolz
Samuele Pedroni

EuroPython 2009

June 30 2009

Current status

- 5th generation of the JIT
- the right one (hopefully :-))
- tracing JIT (like Mozilla TraceMonkey)
- up to Nx faster on trivial benchmarks
  - N = 10, 20, 30, 60 depending on the moon phase
- PyPy evil plan: be consistently faster than CPython in the near future

Main ideas (1)

- 80/20 rule
- 80% of the time is spent in 20% of the code
- Optimize only that 20%

Main ideas (2)

- That 20% has to be composed of *loops*
- Recognize **hot** loops
- Optimize hot loops
- Compile to native code
- Execute :-)

Recognize hot loops

Example

```
def fn(n):
    tot = 0
    while n:
        tot += n
        n -= 1
    return tot
```

Bytecode

```
...
LOAD_FAST          1 (tot)
LOAD_FAST          0 (n)
INPLACE_ADD
STORE_FAST         1 (tot)
LOAD_FAST          0 (n)
LOAD_CONST         2 (1)
INPLACE_SUBTRACT
STORE_FAST         0 (n)
JUMP_ABSOLUTE      9

...
```

Recognize hot loops

Example

```
def fn(n):
    tot = 0
    while n:
        tot += n
        n -= 1
    return tot
```

Bytecode

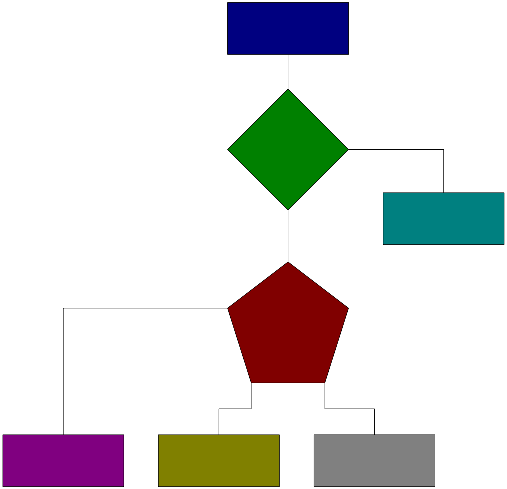
```
...
LOAD_FAST          1 (tot)
LOAD_FAST          0 (n)
INPLACE_ADD
STORE_FAST         1 (tot)
LOAD_FAST          0 (n)
LOAD_CONST         2 (1)
INPLACE_SUBTRACT
STORE_FAST         0 (n)
JUMP_ABSOLUTE      9

...
```

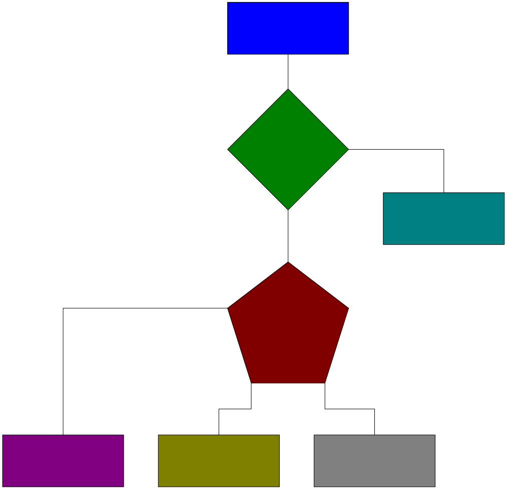
Tracing

- Execute one iteration of the hot loop
- Record the operations, as well as the concrete results
- Linear
- Validity ensured by **guards**
- Recovering logic in case of guard failure

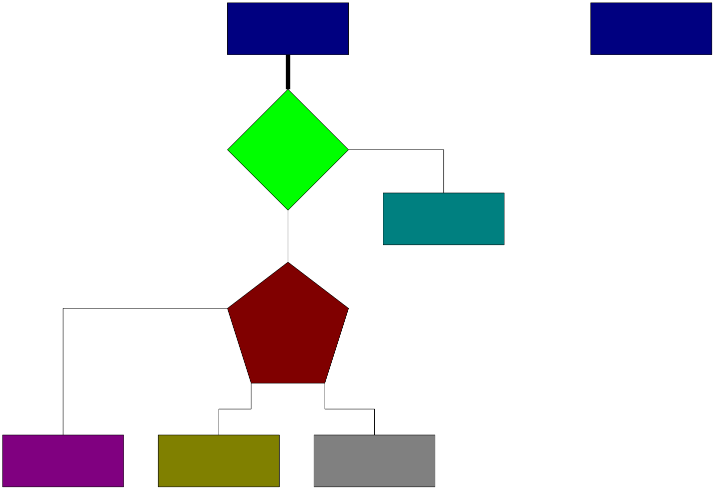
Tracing example



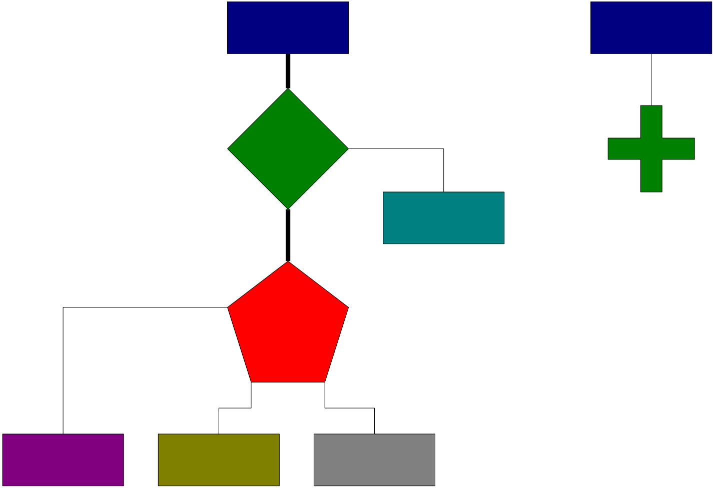
Tracing example



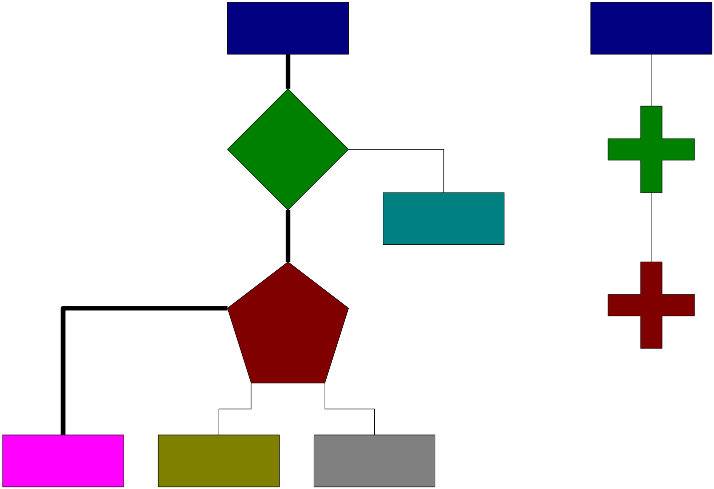
Tracing example



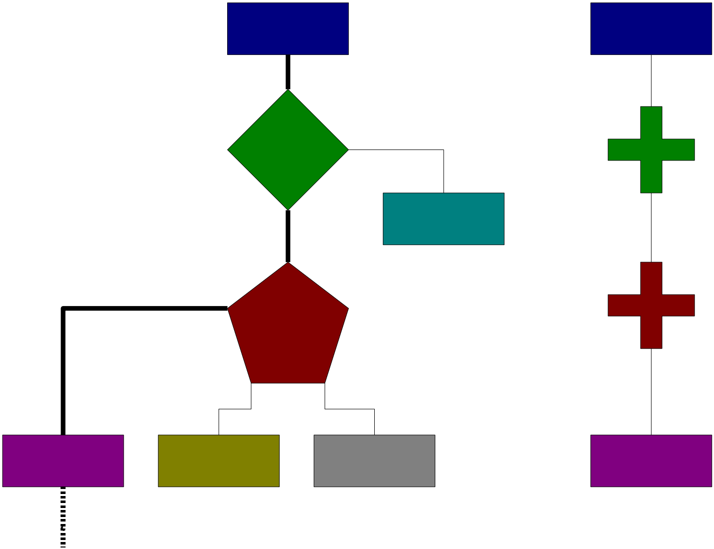
Tracing example



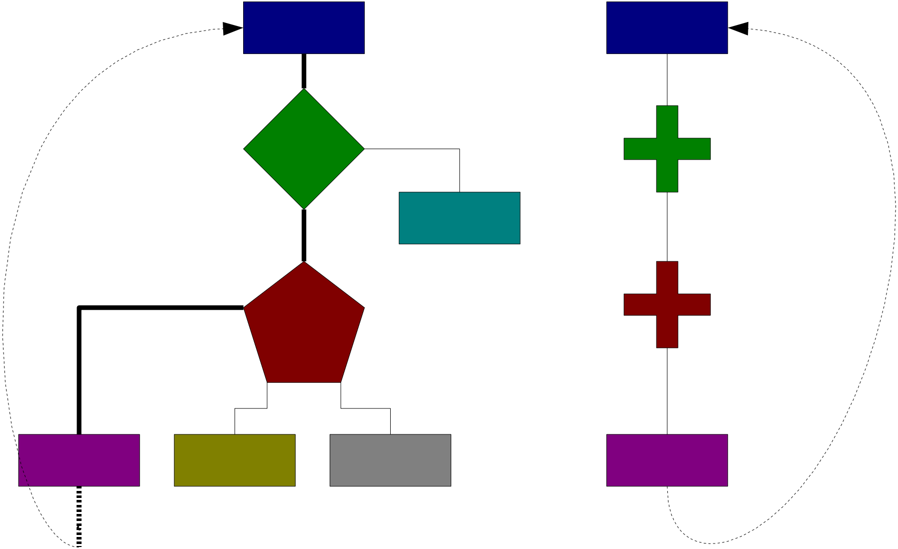
Tracing example



Tracing example



Tracing example



Post-tracing phase

- Generalize or specialize?
- *Generalized* loops can be used more often
- *Specialized* loops are more efficient
- A trace is super-specialized

Perfect specialization

- Generalize the trace...
- ...but not too much
- Most general trace which is specialized enough to be efficient
- e.g.: turn Python `int` into C-level words
- **specialized**: it works only with `int` (and not e.g. `float`)
- **general**: it works with *all* `int` :-)

Optimization phase

- Remove superflous operations
- Constant folding
- Escape analysis: remove unneeded allocations

Code generation

- In theory: the easy part
- Theory != pratice
- The current x86 backend produces suboptimal code
  - but not too bad :-)
- x86-64: not yet, but relatively low effort
- super-experimental CLI/.NET backend
- Contributors welcome :-)

CLI JIT backend

- JIT-over-JIT
  - emit .NET bytecode
  - which is then compiled by .NET's own JIT
- current status: as fast as IronPython on trivial benchmarks
- will be faster than IP in the future

- extremely good results in JIT v2
- it makes a dynamic toy language:
  - as fast as C# for numerical benchmarks
  - faster than C# for some OO benchmarks

CLI JIT backend

- JIT-over-JIT
  - emit .NET bytecode
  - which is then compiled by .NET's own JIT
- current status: as fast as IronPython on trivial benchmarks
- will be faster than IP in the future

- extremely good results in JIT v2
- it makes a dynamic toy language:
  - as fast as C# for numerical benchmarks
  - faster than C# for some OO benchmarks

Contact / Q&A

PyPy: http://codespeak.net/pypy
Blog: http://morepypy.blogspot.com