How to Make Your Programming Language Run Less Slow, Quickly

Laurence
Tratt

Code for main example:
Carl Friedrich Bolz-Tereick

Software Development Team

2017-05-25

What to expect from this session

What to expect from this session

I’ve designed a new programming language! ☺

What to expect from this session

It runs slow ☹

What to expect from this session

A JIT compiler would help ☺

What to expect from this session

JIT compilers are hard ☹

How are programming languages implemented?

Compiler *Interpreter*

How are programming languages implemented?

Compiler *Interpreter*

GCC / clang

How are programming languages implemented?

Compiler	*Interpreter*
GCC / clang	CPython / CRuby

How are programming languages implemented?

Fast output

Compiler	*Interpreter*
GCC / clang	CPython / CRuby

How are programming languages implemented?

Fast output
Compiler

Slow execution
Interpreter

GCC / clang

CPython / CRuby

How are programming languages implemented?

Fast output
Compiler
Hard to write

Slow execution
Interpreter

GCC / clang

CPython / CRuby

How are programming languages implemented?

Fast output	Slow execution
Compiler	*Interpreter*
Hard to write	Easy to write
GCC / clang	CPython / CRuby

Solution? Virtual Machines

Solution? Virtual Machines[†]

[†] The PL type, not the OS type

Solution? Virtual Machines

Interpreter + JIT compiler

JIT compilers enable adaptive compilation

JIT compilers

JIT compilers enable adaptive compilation

Ferociously complex beasts

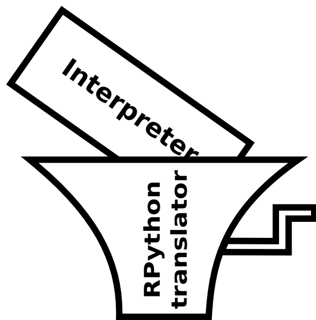
JIT compilers enable adaptive compilation

Ferociously complex beasts

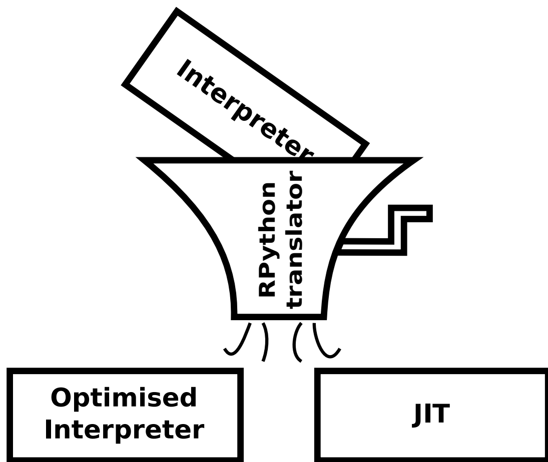
How can they be made easier?

Meta-tracing translation with RPython

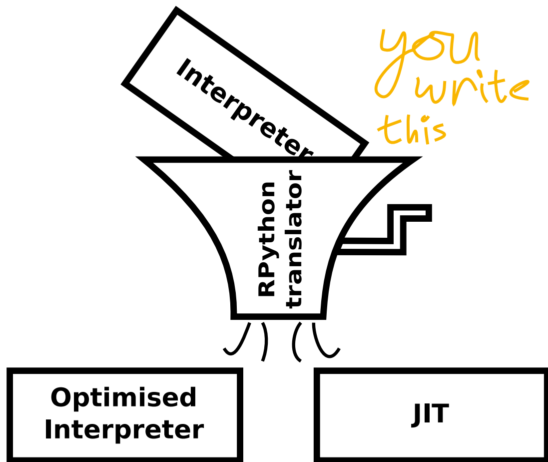
Meta-tracing translation with RPython



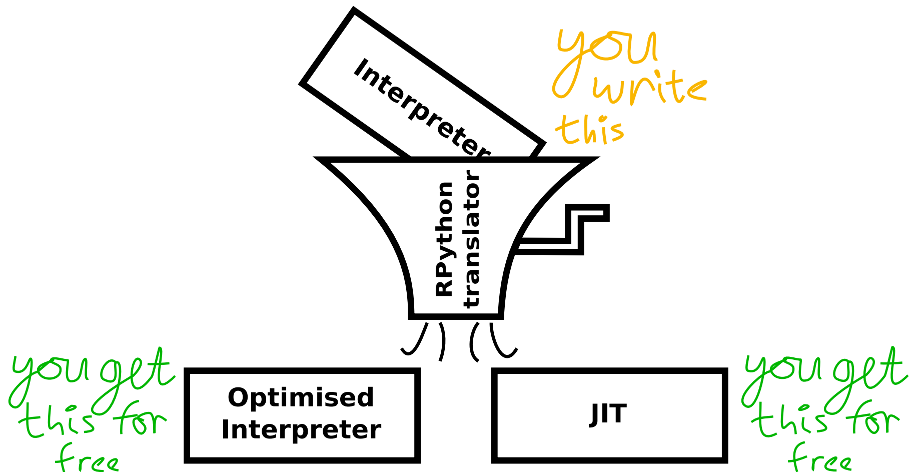
Meta-tracing translation with RPython



Meta-tracing translation with RPython



Meta-tracing translation with RPython



Adding a JIT compiler 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 ...:
        ...
```

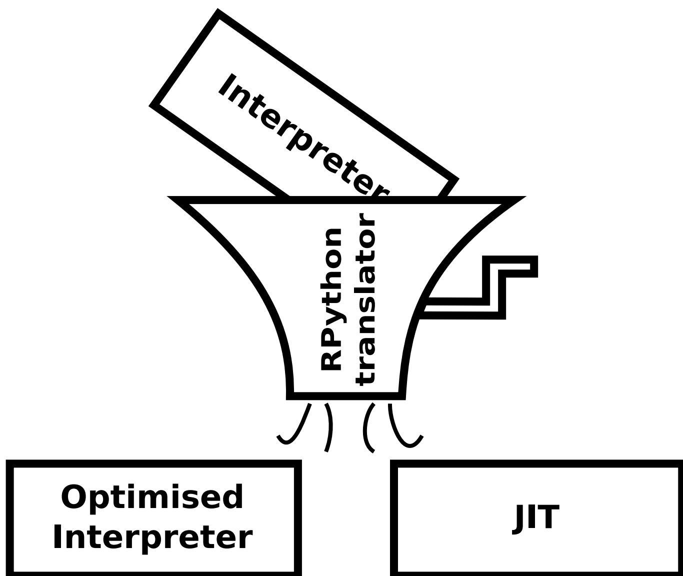
Observation: interpreters are big loops.

Adding a JIT compiler to an RPython interpreter

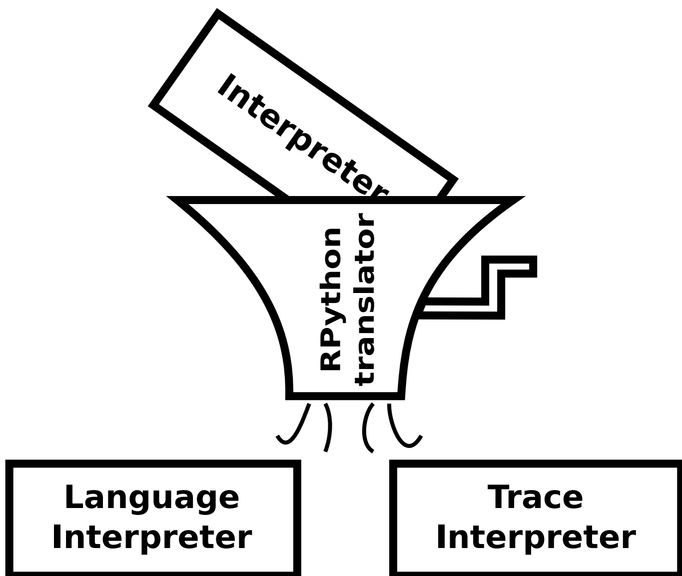
```
...
pc := 0
while 1:
    jit_merge_point(pc)
    instr := load_next_instruction(pc)
    if instr == POP:
        stack.pop()
        pc += 1
    elif instr == BRANCH:
        off = load_branch_jump(pc)
        pc += off
    elif ...:
        ...
```

Observation: interpreters are big loops.

RPython translation



RPython translation



Tracing JITs

User program (lang *FL*)

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

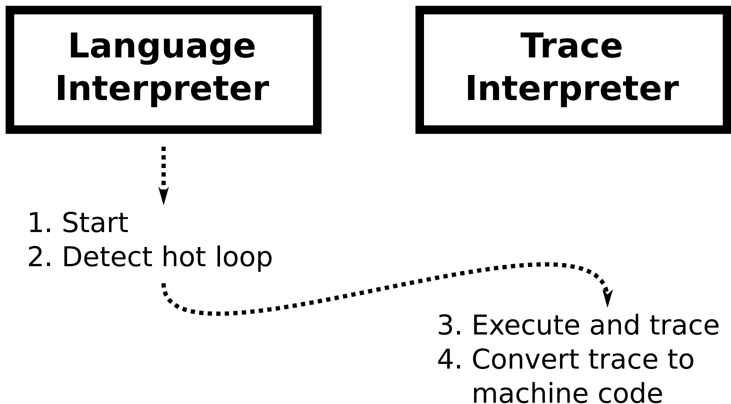
Tracing JITs

User program (lang *FL*)	Trace when x is set to 6
`if x < 0:`	`guard_type(x, int)`
`  x = x + 1`	`guard_not_less_than(x, 0)`
`else:`	`guard_type(x, int)`
`  x = x + 2`	`x = int_add(x, 2)`
`x = x + 3`	`guard_type(x, int)`
	`x = int_add(x, 3)`

Tracing JITs

User program (lang *FL*)	Optimised trace
`if x < 0:`	
`  x = x + 1`	
`else:`	
`  x = x + 2`	`guard_type(x, int)`
`x = x + 3`	`guard_not_less_than(x, 0)`
	`x = int_add(x, 5)`

Meta-tracing VM components



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	User program (lang *FL*)

```
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
```

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

Meta-tracing JITs

FL Interpreter	Initial trace

```
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
```

```
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)
```

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)
```

Optimisation #2 & #3

List folded trace	Dict 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)	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)

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	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, 2)`
`v36 = add(v23, 3)`
`dict_set(v2, "x", v36)`
	`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	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, 2)`
`v36 = add(v23, 3)`
`dict_set(v2, "x", v36)`
	`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.

Task

Task

Write a meta-tracing interpreter

Task

Write a meta-tracing interpreter

We provide a skeleton of a stack-based language – you fill it in

RPython

- A statically typeable subset of Python: think Python with Java's type system.

RPython

- A statically typeable subset of Python: think Python with Java’s type system.
- Types (mostly) implicit, discovered by type inference. Variables need a single type.

RPython

- A statically typeable subset of Python: think Python with Java's type system.
- Types (mostly) implicit, discovered by type inference. Variables need a single type.
- Every RPython program is valid Python (but not necessarily vice versa).

RPython

- A statically typeable subset of Python: think Python with Java’s type system.
- Types (mostly) implicit, discovered by type inference. Variables need a single type.
- Every RPython program is valid Python (but not necessarily vice versa).
- Misses some features (e.g. multiple inheritance) semi-randomly.

First steps

- Download PyPy sources `https://www.dropbox.com/s/k4u9t1xtopn6n6u/pypy2-v5.7.1-src.tar.bz2` and decompress them to `/path/to/pypy`.

First steps

- Download PyPy sources `https://www.dropbox.com/s/k4u9t1xtopn6n6u/pypy2-v5.7.1-src.tar.bz2` and decompress them to `/path/to/pypy`.
- Download TLA `https://www.dropbox.com/s/svbcv83rhoc0ks6/tla_skeleton.tgz`

First steps

- Download PyPy sources `https://www.dropbox.com/s/k4u9t1xtopn6n6u/pypy2-v5.7.1-src.tar.bz2` and decompress them to `/path/to/pypy`.
- Download TLA `https://www.dropbox.com/s/svbcv83rhoc0ks6/tla_skeleton.tgz`
- Run TLA's tests: `python /path/to/pypy/pytest.py -x` and then fix them one by one.

First steps

- Download PyPy sources `https://www.dropbox.com/s/k4u9t1xtopn6n6u/pypy2-v5.7.1-src.tar.bz2` and decompress them to `/path/to/pypy`.
- Download TLA `https://www.dropbox.com/s/svbcv83rhoc0ks6/tla_skeleton.tgz`
- Run TLA's tests: `python /path/to/pypy/pytest.py -x` and then fix them one by one.
- Type check things occasionally with:
  `python /path/to/pypy/rpython/bin/rpython -O2 targettla.py`
  [If you have a PyPy binary, replace `python /path/to/...` with `pypy /path/to...`.]
- One thing to note: to get an instruction from `bytecode` (which is a string), use `ord` e.g. `instr = ord(bytecode[pc])`.

A full JIT

- When all tests pass add a merge point at the start of the interpreter loop (yours may vary depending on variable names and structure):

```
jitdriver.jit_merge_point(pc=pc, \
  bytecode=self.bytecode, self=self)
```

- Then you can generate a JIT:

```
python /path/to/pypy/rpython/bin/rpython -Ojit targettla.py
```

  will (eventually) produce a binary `targettla-c`.

- ‘Compile’ TLA programs with:

```
PYTHONPATH=/path/to/pypy python tla_assembler.py \
  add_10.tla.py
```

  which will produce a `add_10.tla` file that you can run:

```
./targettla-c add_10.tla
```

A full JIT

- You *know* that some things are partly or wholly static: telling the JIT optimiser about these helps it do a better job.
- You can log the trace and get useful summary statistics to `/path/to/trace` with:

```
PYPYLOG=jit-log-opt,jit-summary:/path/to/trace \
  ./targettla-c add_10.tla
```

- Use features like `_immutable_fields_`, virtualisables, elidables, and fixed size lists (e.g. `[None] * 1024`) to gradually optimise the trace. See e.g. `https://rpython.readthedocs.io/en/latest/jit/virtualizable.html` Pay particular attention to `cond_call` which are calls to C functions: these are black boxes which the trace optimiser has to treat with extreme caution (i.e. these calls prevent the trace optimiser doing a good job). Optimising away such calls can be a significant win.

Thanks for listening