Partial evaluation in Python

Partial evaluation in Python

- What is partial evaluation
- When should we apply it?
- Real-world case study
- Partial evaluation library

Part 1

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

What is partial evaluation?

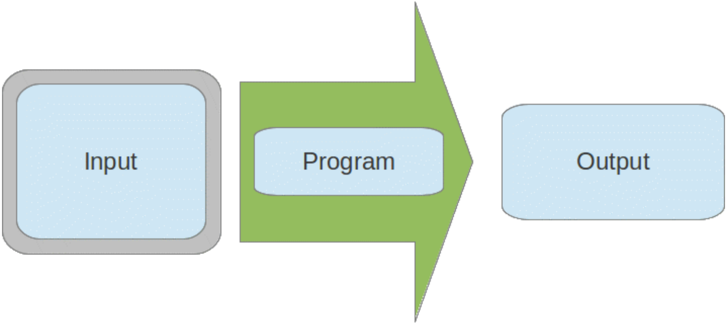
What is partial evaluation?

Partial evaluation is:

- an optimization technique
- that generates specialized code
- using information, available at runtime

Original program

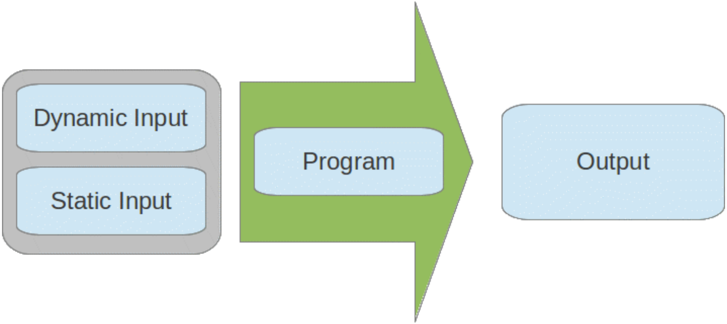
- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library



```
>>> power = lambda x, n: x ** n
>>> power(3, 5)
243
```

Separate inputs

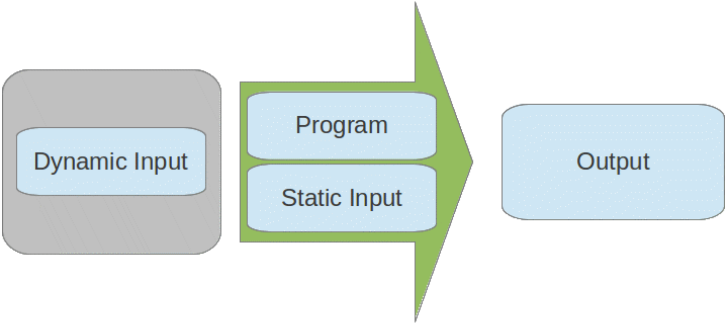
- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library



```
v = [power(x, 3) + 3 * power(x, 47)
     for x in A_LOT_OF_DATA]
```

Specialized program

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library



```
v = [power_3(x) + 3 * power_47(x)
     for x in A_LOT_OF_DATA]
```

An example: power function

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```
def power(x, n):
    if not isinstance(n, int) or n < 0:
        raise ValueError
    elif n == 0:
        return 1
    elif n % 2 == 0:
        v = power(x, n / 2)
        return v * v
    else:
        return x * power(x, n - 1)
```

Specialized: n = 'foo', n = 1

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```
def power_foo(x):
    raise ValueError

def power_1(x):
    return x
```

Specialized: n = 5

```python
def power_5(x):
    _pow_2 = x * x
    _pow_4 = _pow_2 * _pow_2
    return x * _pow_4
```

Specialized: n = 27

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```python
def power_27(x):
    _pow_2 = x * x
    _pow_3 = _pow_2 * x
    _pow_6 = _pow_3 * _pow_3
    _pow_12 = _pow_6 * _pow_6
    _pow_13 = _pow_12 * x
    _pow_26 = _pow_13 * _pow_13
    _pow_27 = _pow_26 * x
    return _pow_27
```

Performance: n = 5

```
timeit.timeit(
    'for x in xrange(20): tests.power_5(x)',
    'import tests',
    number=5000000)
```

statement	CPython 2.7	PyPy 1.9
power(5, x)	266.4 s	
power_5(x)	32.1 s	
x ** 5	12.9 s	

Performance: n = 5

```
timeit.timeit(
    'for x in xrange(20): tests.power_5(x)',
    'import tests',
    number=5000000)
```

statement	**CPython 2.7**	**PyPy 1.9**
power(5, x)	266.4 s	0.73 s
power_5(x)	32.1 s	0.68 s
x ** 5	12.9 s	0.55 s

Performance: n = 27

```
timeit.timeit(
    'for x in xrange(20): tests.power_27(x)',
    'import tests',
    number=500000)
```

statement	CPython 2.7	PyPy 1.9
power(27, x)	29.8 s	2.26 s
power_27(x)	6.65 s	2.18 s
x ** 27	15.8 s	21.9 s

Analysis: power(x, 27)

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```
def power(x, n):                                      11.19 %
    if not isinstance(n, int) or n < 0:               29.52 %
        raise ValueError
    elif n == 0:
        return 1
    elif n % 2 == 0:                                  20.59 %
        v = power(x, n / 2)                           14.70 %
        return v * v                                   5.80 %
    else:
        return x * power(x, n - 1)                    17.21 %
```

Analysis: power_27(x)

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```
def power_27(x):                                11.66 %
    _pow_2 = x * x                               9.55 %
    _pow_3 = _pow_2 * x                          5.58 %
    _pow_6 = _pow_3 * _pow_3                     6.58 %
    _pow_12 = _pow_6 * _pow_6                    6.95 %
    _pow_13 = _pow_12 * x                        6.54 %
    _pow_26 = _pow_13 * _pow_13                 17.93 %
    _pow_27 = _pow_26 * x                       14.58 %
    return _pow_27
```

Implementation

```
def make_power(n):
    ...
    elif n == 0: return lambda x: 1
    elif n == 1: return lambda x: x
    elif n >= 2:
        source = 'def fun(x):\n'
        source += '\n'.join('  ' + s
            for s in _power_stmts(n))
        fn_def = compile(
            source, '<nofile>', 'exec')
        eval(fn_def)
        return locals()['fun']
```

Implementation

- **Part 1: What is partial evaluation?**
- Part 2: When should we apply it?
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```python
def _power_stmts(n, ret=True):
    if n == 2: stmts = ['_pow_2 = x * x']
    elif n % 2 == 0:
        stmts = _power_stmts(n / 2, ret=False)
        stmts.append(
            '_pow_{n} = _pow_{n2} * _pow_{n2}'\
            .format(n = n, n2 = n / 2))
    else:
        stmts = _power_stmts(n - 1, ret=False)
        stmts.append('_pow_{n} = _pow_{n1} * x'\
             .format(n = n, n1 = n - 1))
    if ret: stmts.append('return _pow_{n}'.format(n=n))
    return stmts
```

To sum it up

- We generate code, specializing on inputs
- Specialized code can run much faster
- But code generation is messy

Part 2

- Part 1: What is partial evaluation?
- **Part 2: When should we apply it?**
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

When should we apply it?

When should we apply it?

- We can separate input into static and dynamic
- There is no single performance bottleneck
- The program is rather large and non-trivial
- So we don't want to rewrite it in C
- But we care about performance enough to introduce some complexity

- Part 1: What is partial evaluation?
- **Part 2: When should we apply it?**
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

Larger context

- „Premature optimization is the root of all evil...“
- Profile
- Think about the algorithm
- Test coverage
- Benchmarks

Possible applications

- Interpreters of:
    - programming languages
    - business-logic rules
    - DB queries
    - spreadsheet formulas
- Template engines

- Part 1: What is partial evaluation?
- **Part 2: When should we apply it?**
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

Pick low-hanging fruit first

- Caching, memoization
- Closures
- Lazy evaluation

Example: closures

```python
def make_fn(static_input):
    value = some_big_computation(static_input)
    def fn(dynamic_input):
        x = value.do_stuff(dynamic_input)
        return x
```

- Part 1: What is partial evaluation?
- **Part 2: When should we apply it?**
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

But

If you need to change control flow, these methods will not help you.

No way to turn this

- Part 1: What is partial evaluation?
- **Part 2: When should we apply it?**
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```python
def power(x, n):
    if not isinstance(n, int) or n < 0:
        raise ValueError
    elif n == 0:
        return 1
    elif n % 2 == 0:
        v = power(x, n / 2)
        return v * v
    else:
        return x * power(x, n - 1)
```

Into this

- Part 1: What is partial evaluation?
- **Part 2: When should we apply it?**
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

```python
def power_27(x):
    _pow_2 = x * x
    _pow_3 = _pow_2 * x
    _pow_6 = _pow_3 * _pow_3
    _pow_12 = _pow_6 * _pow_6
    _pow_13 = _pow_12 * x
    _pow_26 = _pow_13 * _pow_13
    _pow_27 = _pow_26 * x
    return _pow_27
```

To sum it up

- Part 1: What is partial evaluation?
- **Part 2: When should we apply it?**
- Part 3: Real-world case study
- Part 4: Partial evaluation as a library

- Good fit for interpreters, but also other domains
- Test, benchmark, keep larger context
- Consider caching and closures

Part 3

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- **Part 3: Real-world case study**
- Part 4: Partial evaluation as a library

Real-world case study

Real-world case study

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- **Part 3: Real-world case study**
- Part 4: Partial evaluation as a library

- Problem domain: data collection, OLAP
- Data is stored in a graph DB, in EAV model
- Users define data structure, reports, formulas
- We must optimize formulas evaluation

The problem

- Formulas evaluation is the (big) bottleneck
- All simple and local optimizations are already done
- Time is evenly spread around a large set of functions
- And most of it is spent analysing formulas

Pseudocode

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- **Part 3: Real-world case study**
- Part 4: Partial evaluation as a library

```
def get_value(attr, key):
    if self.key_in_db(key):
        return self.get_db_value(key)
    for f in attr.formulae_before_aggregation():
        res = self.get_formula_value(attr, key, f)
        if res is not None:
            return
    for aggr_formula in attr.aggr_formulae():
        ...
    for f in attr.formulae_after_aggregation():
        ...
```

Pseudocode

```
def get_formula_value(descriptor, key, formula):
    lhs_key = dict(key)
    if formula.left_filters:
        cond_d, _ = formula.left_filters.as_dnf()
        for k, v in cond_d:
            if k in lhs_key:
                ...
    for arg in formula.args:
        arg_key = dict(lhs_key)
        if arg.filters:
            ...
        arg_value = get_value(arg.descriptor, arg_key)
    ...
```

Gets compiled into this

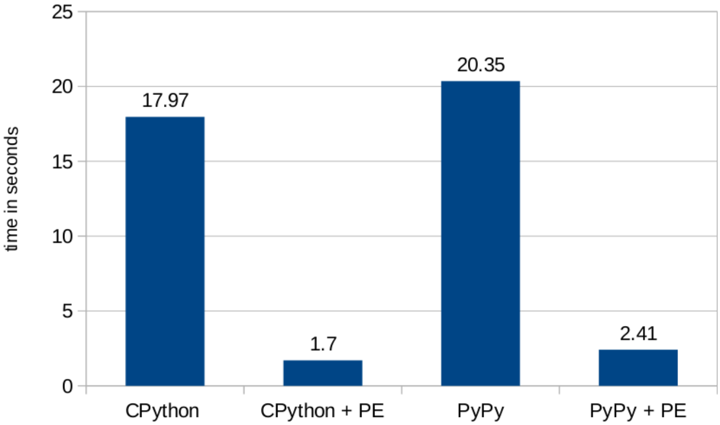
```
def get_value_1(x, y, z):
    ...
    first_iter = True
    while first_iter:
        first_iter = False
        arg_1 = get_value_2(x, z.next())
        if arg_1 is None:
            break
        ...
        res = some_fn(arg_1, arg_2)
    ...
```

Results

- Calculation engine: 2000 LOC
- ``Interpreter'' code that was specialized: 300 LOC
- ``Compiler'' code, that does this specialization: 400 LOC
- Performance: about 10x faster

Performance

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- **Part 3: Real-world case study**
- Part 4: Partial evaluation as a library



Making it maintainable

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- **Part 3: Real-world case study**
- Part 4: Partial evaluation as a library

- Good test coverage
- Benchmarks, regression tests on speed and correctness
- Debugging: output generated code to file, support pdb
- Can run without generating specialized code
- New features are added first without specialization

To sum it up

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- **Part 3: Real-world case study**
- Part 4: Partial evaluation as a library

- Partial evaluation gave 10x speedup
- Removed a huge bottleneck
- Code it still maintainable
- But you have to be careful

Part 4

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

Partial evaluation as a library

Partial evaluation as a library

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

How cool would it be to just write:

```
import ast_pe
power_5 = ast_pe.specialized_fn(
    power, n=5)
```

Partial evaluation libraries

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

- Were developed for Lisp dialects, and even for C
- Was attempted for Ruby
- I have not found a Python one
- So decided to write it

How does it work

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

- Transforms AST of the target function
- Looks like a little Python AST interpreter
- Uses Python interpreter to evaluate expressions depending only on static input

Abstract Syntax Tree

```python
def sample_fn(x, y, foo='bar'):
    if (foo == 'bar'):
        return x + y
    else:
        return x - y
```

Abstract Syntax Tree

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

```
>> import ast, inspect, meta.asttools
>> meta.asttools.print_ast(
        ast.parse(inspect.getsource(sample_fn)))
```

Abstract Syntax Tree

```
FunctionDef(
    args=arguments(
        args=[Name('x'), Name('y'), Name('foo')],
    defaults=[Str(s='bar')],
    kwarg=None,
    vararg=None),
body=[If(
    body=[Return(value=BinOp(
        left=Name(ctx=Load(), id='x'), op=Add(),
        right=Name(ctx=Load(), id='y')))],
    orelse=[Return(value=BinOp(
        left=Name(ctx=Load(), id='x'), op=Sub(),
        right=Name(ctx=Load(), id='y')))],
    test=...
```

Transforming AST

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

```
class PartialEvaluator(ast.NodeTransformer):
    ...
    def visit_Name(self, node):
        self.generic_visit(node)
        if isinstance(node.ctx, ast.Load) \
                and node.id in self.bindings:
            value = self.bindings[node.id]
            if isinstance(value, numbers.Number):
                return ast.Num(n=value)
            ...
```

But this is not so easy

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

- I/O - files, DB: `f.write(data)`
- Variable assignment:
  `static_input += dynamic_i`
- Variable mutation:
  `static_input.append(dynamic_i)`
- Redefining method:
  `some_obj.method = another_method`
- Redefining operator:
  `def __getitem__(self, key, ...):`

So we have to

- Do dataflow analysis
- Detect mutations and assignments
- Rollback if some assumptions were proven wrong

Assumptions

- You do not mutate/change static input
- Mark pure functions (`@pure` decorator)
- Mark functions you want to be inlined (`@inline` decorator)

To sum it up

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

- It is mostly a prototype now
- But proves it can be done

Contacts

- Part 1: What is partial evaluation?
- Part 2: When should we apply it?
- Part 3: Real-world case study
- **Part 4: Partial evaluation as a library**

- Konstantin Lopuhin
- konstantin.lopuhin@chtd.ru
- https://github.com/chtd/ast_pe
- CHTD - http://chtd.ru
- Twitter: @python_chtd