peval Documentation

Release 0.1.0

Bogdan Opanchuk

Nov 20, 2018

Contents

CHAPTER 1

Introduction

This library allows one to perform code specialization at run-time, turning this:

```
@inline
def power(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        v = power(x, n // 2)
        return v * v
    else:
        return x * power(x, n - 1)
```

with `n` set to, say, 27, into this:

```
def power_27(x):
    __peval_return_7 = (x * 1)
    __peval_return_6 = (__peval_return_7 * __peval_return_7)
    __peval_return_5 = (x * __peval_return_6)
    __peval_return_4 = (__peval_return_5 * __peval_return_5)
    __peval_return_3 = (__peval_return_4 * __peval_return_4)
    __peval_return_1 = (x * __peval_return_3)
    __peval_return_2 = (__peval_return_1 * __peval_return_1)
    return (x * __peval_return_2)
```

The resulting code runs 6 times faster under CPython 3.5.1.

The API is identical to that of `functools.partial()`:

```
import peval
power_27 = peval.partial_apply(power, n=27)
```

You must mark the functions that you want inlined (maybe recursively) with *`peval.inline()`*. If want some function to be evaluated during partial evaluation, mark it with *`peval.pure()`* (if it is not, in fact, pure, the results are unpredictable).

CHAPTER 2

Implementation details

The partial evaluation is performed on an AST using a range of optimizations:

- constant propagation
- constant folding
- unreachable code elimination
- function inlining

CHAPTER 3

Restrictions on functions

Currently not all functions can be partially evaluated or inlined. Hopefully, these restrictions will be lifted in the future as the package improves.

The inlined functions:

- cannot be `async` functions;
- cannot be generators;
- cannot have nested definitions (lambdas, functions or classes);
- cannot have closures;
- cannot have decorators and annotations (they will be ignored).

The partially evaluated functions:

- cannot be `async` functions;
- cannot have nested definitions (lambdas, functions or classes);
- cannot have decorators and annotations (they will be ignored).

Also note that the partial evaluated function loses the connection to the globals dictionary of the original function. Therefore, it cannot reassign global variables (the copy is shallow, so mutation of global objects is still possible).

CHAPTER 4

API reference

4.1 Core functions

peval.**partial_eval**(*func*)
Returns a partially evaluated version of `func`, using the values of associated global and closure variables.

peval.**partial_apply**(*func*, **args*, ***kwds*)
Same as *partial_eval()*, but in addition uses the provided values of positional and keyword arguments in the partial evaluation.

4.2 Helper functions

peval.**getsource**(*func*)
Returns the source of a function `func`. Falls back to `inspect.getsource()` for regular functions, but can also return the source of a partially evaluated function.

peval.**specialize_on**(*names*, *maxsize=None*)
A decorator that wraps a function, partially evaluating it with the parameters defined by `names` (can be a string or an iterable of strings) being fixed. The partially evaluated versions are cached based on the values of these parameters using `functools.lru_cache` with the provided `maxsize` (consequently, these values should be hashable).

4.3 Tags

peval.**inline**(*func*)
Marks the function for inlining.

peval.**pure**(*func*)
Marks the function as pure (not having any side effects, except maybe argument mutation).

Python Module Index

p

Index