StaticPy Documentation

Release 0.2.1

SnowWalkerJ

Jan 14, 2020

Contents:

CHAPTER 1

Introduction

StaticPy is a Python-to-C++ translater. It is designed in a way that code can be run in both Python and C++ mode. That is, StaticPy valid code should be able to not only be translated to C++ and compiled, but also run by Python interpreter directly.

It supports a subset of Python grammar. It analyzes Python AST and translates it to C++. You can write a C++ template and embed the generated code in your template to form a complete code.

Optionally, it can bind the C++ code back to Python using Pybind11 to grant super speed to Python.

1.1 Compare to alternatives

1.1.1 Compare to Cython

Cython has the advantage of being very flexible. It translates everything static enough to C and leave the rest as it is. By doing that, it allows you to enjoy both the flexibility of Python and fastness of C. StaticPy, however, requires you use only supported grammar(only static-typed variables, for example). The good thing about it is that it can generate pure C++ code without support of Python, so that you can embed the code to a C++ project.

Another difference is that Cython compiles the source code in a static way. It analyzes the source code and translates it to C. An explicit compilation is necessary before you can use the generated module. StaticPy is more like numba in a way that only a decorator around the function and everything is done.

1.1.2 Compare to numba

StaticPy supports C++ better than numba. You can include outside sources and use C++ functions/class directly with StaticPy.

In order to generate pure C++ code, numpy is not supported by StaticPy, which is a huge disadvantage compared to numba.

In addition, the compiling time of StaticPy seems much longer.

CHAPTER 2

Get Started

2.1 Install StaticPy

```
git clone https://github.com/SnowWalkerJ/StaticPy.git
cd StaticPy
pip install -r requirements.txt
python setup.py install
```

2.2 Concepts

C++ is a static-type language. Each variable has a type determined at compile time. StaticPy is not good at deducing the types for you. So if you wish StaticPy to speedup your code, you have to carefully annotate each variable with proper type.

2.3 Using StaticPy

2.3.1 jit

One way to use StaticPy to speed up your code is through the *jit* decorator. You should carefully annotate each variable, parameter and return value.

```
from staticpy import jit

@jit
def frac(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * frac(n - 1)
```

The function will be compiled the first time it is called. If you wish it to compile immediatelly, call *frac.compile()*.

A jit function will be re-compiled under the following circumstances:

- the source code of the function is modified since the last compilation
- a *force-compile* option is turned on
- *obj.compile()* is called

Note that a *jit* function is strict on types. You can't pass an int value to a float parameter or a float value to an int parameter. A manually overloading is needed.

2.3.2 import hook

StaticPy has a import-hook that allows compile a whole module when you import it. Enable it by calling *import staticpy.hook* before you import any other module.

To specify a module that wishes to be compiled by StaticPy, write a signal *# @staticpy* at the first line of your module. This signal lets StaticPy know which modules should be compiled.

```python
# @staticpy
# mod.py
# make sure # @staticopy is at the first line

def frac(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * frac(n - 1)
```

```python
# main.py
import staticpy.hook          # Enables compile-at-import
import mod                    # StaticPy compiles `mod`

assert mod.frac(4) == 24
```

CHAPTER 3

User Guide

3.1 Declare variable

Like in C++, each variable must be declare before it's used. Declare a variable using annotated assignment in StaticPy. Here is an example:

```
variable: int = 0
# or
variable: int
```

The later one declares a variable without intializing it.

3.2 Constants

If you assign a variable wihout declaring its type, the variable is considered a python constant. This constant is like a macro variable in C/C++. The variable itself will never appear in the generated code. It's replaced with the value it's assigned with. For example,

```
zero = 0
myvar: int = zero + 1
```

will be translated into:

```
int myvar = 0 + 1;
```

3.3 Types

3.3.1 Primitives

Basic types are *Bool*, *Int*, *Long*, *Float* and *Double*, which represents *bool*, *int*, *long*, *float* and *double* in C++ respectively. If you use Python type *int* and *float*, they are also mapped to *int* and *float* in C++.

We understand that *int* and *float* in Python actually represents *long* and *double* in C. But we think it is confusing that *Int* means *int* while *int* means *long*.

3.3.2 Arrays

Arrays are supported as function parameters (but not return types yet). You can declare an array type by something like *Int[:]* or *Float[3]*. This is almost like what you would expect in *Numba* or *Cython*. Array types with provided shapes is appreciated because compiler can take advantage of this information to optimize the generated code.

Multi-dimensional arrays can be declared with *Long[3, 2]*.

Just like in *Numba* and *Cython*, a continuous array is much more efficient than a normal array. So if you are sure an array is continuous, you should annotate it to generate more efficient code. Unlike *Numba* or *Cython*, which annotate a continuous array with *int[::1]*, *StaticPy* use a bool flag at the end of the shape annotation. Use *Int[3, 2, True]* to annotate a continuous type with 3x2 elements. If you feed a non-continuous array to a continuous typed parameter, it may access an invalid memory and cause error or (worsely) return a wrong result without warnings.

3.3.3 Lists and Dicts

List and dict are commonly used containers in Python. They have counterparts in C++ as well. Using them in StaticPy is possible yet not realized. This is one of the most important features we are currently working on.

3.4 Flow Control

Most flow-control statements are supported in StaticPy such as *while*, *if*, *else*. *do-while* is not supported because it doesn't exist in Python. The *elif* is a little tricky, though. It is translated to *else { if () {}}*. This is sementically equivalent in C/C++, but not quite human-friendly to read.

Another commonly used feature is *for*. So far only *for x in range(. . .)* is valid in StaticPy. It is translated into *for (i = start; i < end; i += step) {}*. A general form of for-each is neither supported nor intend to be supported in the short term.

3.5 Standard Library Functions

Very few C++ standard library functions have Python counterparts. We don't intend to port them in StaticPy. However, functions in *cmath* and *iostream* are so commonly used that we consider it inconvinient missing them.

3.5.1 iostream

Commonly used objects *cin*, *cout*, *cerr* and *endl* are implemented in *staticpy.lib.iostream*. You are free to use them in both Python mode and C++ mode. One difference is that you need to call `cout()` to actually get the *cout* object. The same applies to *cin*, *cerr* and *endl*.

What's more, the *cin >> x* usage in C++ relies heavily on the overload of operator>> based on static typing. So the cin object doesn't function properly in Python.

```python
from staticpy.lib.iostream import cout, endl
def myprint(num: float):
    cout() << "The value is " << num << endl()
```

There is an additional function we define for easy and pretty output: *cprint*.

```python
from staticpy.lib.iostream import cprint
def myprint():
    cprint(1.0)                         # 1.0
    cprint(x=1.0)                       # x = 1.0
    cprint("Hello", my_name="Jack")   # Hello, my_name = Jack
```

3.5.2 cmath

Many math functions has implementations in both Python and C++. You can access them in *staticpy.lib.cmath*.

```python
from staticpy.lib.cmath import cos

def mycos(x: float) -> float:
    return cos(x)
```

3.6 External Functions

StaticPy allows you invoke external C++ functions. These functions can't be called in pure Python mode, of course. But they can function properly after compilation. Use *staticpy.util.extern.ExternalFunction* to declare external functions.

```python
from staticpy.util.extern import ExternalFunction
from staticpy import jit

cos = ExternalFunction("cos", "<cmath>", "std")

@jit
def mycos(x: float) -> float:
    # This cos function can not be called in Python mode.
    return cos(x)
```

An external function can also be a function template.

```python
from staticpy.util.extern import ExternalFunction
from staticpy import jit

fmax = ExternalFunction("fmax", "<cmath>", "std")

@jit
def mymax(x: float) -> float:
    return fmax[float](x, 0)      # std::fmax<float>(x, 0.0)
```

CHAPTER 4

Translator

One major goal for StaticPy is to generate pure C++ code. This is done by calling a translator.

```python
import inspect
from staticpy.translator import BaseTranslator

def fn(n: int) -> int:
    s: int = 0
    for i in range(n):
        s += i
    return s

source = inspect.getsource(fn)
translator = BaseTranslator()
block = translator.translate(source)

print("\n".join(block.translate()))
```

This prints out

```cpp
int fn(int n) {
  int s = 0;
  for(int i = 0; i < n; i++) {
    s += i;
  }
  return s;
}
```

CHAPTER 5

Contribute

CHAPTER 6

Indices and tables

- genindex
- modindex
- search