Pontifícia Universidade Católica de Minas Gerais
Departamento de Ciência da Computação
Disciplina: Compiladores - Trabalho Prático 2 (2024/01)
Professor: Pedro Ramos

1. Introdução

Compilação JIT (*do inglês: Just in Time*) [1] é uma técnica escolhida na computação interpretada pela sua capacidade de melhorar o desempenho de código durante o tempo de execução. Essa abordagem é particularmente valiosa em linguagens como Python, que são interpretadas e tendem a ser mais lentas em comparação a linguagens compiladas. Dois exemplos proeminentes de compiladores JIT para Python são PyPy e Numba.

PyPy [2] é um interpretador Python alternativo que utiliza a compilação JIT para melhorar o desempenho do código Python em tempo de execução. Ele traduz o código Python em bytecode de máquina virtual e otimiza o código conforme necessário durante a execução. Numba [3], por outro lado, é uma biblioteca Python que traduz funções Python em código de máquina otimizado utilizando a infraestrutura LLVM [4]. Numba utiliza JIT para compilar funções Python em tempo de execução, resultando em um código otimizado que é executado significativamente mais rápido que o código padrão.

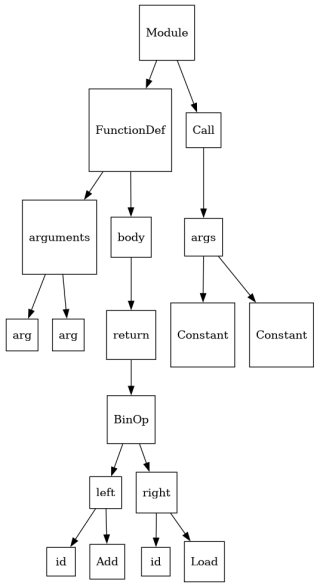
Ambas as ferramentas realizam análises sobre a árvore de sintaxe abstrata do programa enquanto ele executa.

A biblioteca **ast** [5] do python é um ótimo começo para tais análises. Com ela é possível obter um objeto estruturado que representa a definição estática do programa.

```
# main.py
import ast

codigo = """
def foo(a, b):
    return a + b

foo(1, 2)
"""

if __name__ == '__main__':
    tree = ast.parse(codigo)
    print(ast.dump(tree))
```

O código acima, por exemplo, imprime na saída padrão uma versão textual da árvore do programa escrito na variável **codigo**. **A saída, porém, não é muito legível.** A imagem abaixo demonstra uma possível visualização deste *dump* em linguagem .dot (omitindo-se alguns nós da árvore).

```
Module(body=[FunctionDef(name='foo',
args=arguments(posonlyargs=[],
args=[arg(arg='a'), arg(arg='b')],
kwonlyargs=[], kw_defaults=[], defaults=[]),
body=[Return(value=BinOp(left=Name(id='a',
ctx=Load()), op=Add(), right=Name(id='b',
ctx=Load()))))], decorator_list=[]),
Expr(value=Call(func=Name(id='foo',
ctx=Load()), args=[Constant(value=1),
Constant(value=2)], keywords=[]))],
type_ignores=[])
```

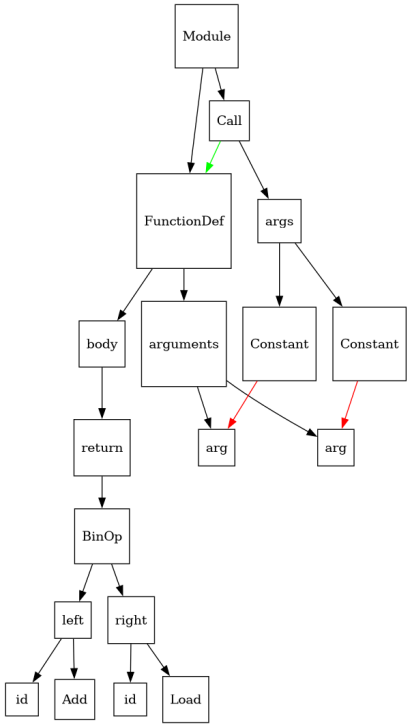


2. Objetivo

É importante notar, porém, que a análise feita pela biblioteca **ast** é puramente intraprocedural, isto é, não relaciona o corpo da definição de uma função com as possíveis chamadas a mesma.

Neste trabalho, você deverá implementar uma análise interprocedural [6] que produza uma representação em grafo de um código python que relacione as chamadas de função com suas respectivas definições.

Por exemplo, uma representação interprocedural do mesmo programa acima está descrita na imagem abaixo.



Na representação acima, há a presença de 3 novas arestas:

- Aresta verde, que liga o nó **CALL** (chamada de função) ao nó **FUNCTIONDEF** (Definição de função), indicando que a chamada é relativa àquela definição.
- Arestas vermelhas, que ligam os nós **CONSTANT** (parâmetros 1 e 2) aos nós **ARGS** da definição da função.

3. O que deve ser entregue

I. Um **programa** em qualquer linguagem que receba como entrada um arquivo de texto contendo um programa em python (.py, por exemplo) e produza, de saída, uma representação em grafo/árvore interprocedural do programa python contido no arquivo. A representação em grafo/árvore não precisa ser na linguagem .dot, pode ser qualquer representação textual, contanto que seja interprocedural, isto é, contanto que conecte chamadas de função às definições das mesmas.

II. Um **relatório** contendo:

   A. A explicação do algoritmo utilizado para produzir o grafo interprocedural;

   B. A resposta para a pergunta: “Qual ou quais estratégias de otimização são possíveis somente em análise interprocedural? Exemplifique”.

4. Por onde começar

Neste repositório existe um script em python que imprime todos os nós da ast e seus respectivos filhos. Para expandir o código basta continuar a implementação já existente de uma classe que herda de ast.NodeTransformer ou ast.NodeVisitor. Essa classe percorre a ast de um programa python. Para aprender quais são os tipos de nós, consulte a documentação da biblioteca.

Boa sorte!

5. Bibliografia

[1] Languages, Compilers, and Runtime Systems, University of Michigan, Computer Science and Engineering, 2018
[2] "PyPy - A fast, compliant alternative implementation of the Python language", disponível em: https://www.pypy.org/
[3] "Numba - A high performance Python compiler", disponível em: https://numba.pydata.org/
[4] "LLVM Project", disponível em: https://llvm.org/
[5] "ast — Abstract Syntax Trees", disponível em: https://docs.python.org/3/library/ast.html
[6] "Lecture Notes: Interprocedural Analysis", Carnegie Mellon University, disponível em
https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/interprocedural.pdf