GROUP 13
CS3020
6 October 2015

PyPy Report

A compiler-theoretic report

INTRODUCTION

PyPy is a Python interpreter and a Just-In-Time(JIT) compiler. PyPy focuses on speed, efficiency and compatibility with the original Python interpreter. PyPy started out being a Python interpreter written in the Python language itself. Current PyPy versions are translated from RPython to C code and then compiled. The JIT compiler is capable of translating Python code into machine code at runtime. PyPy emphasises on clean separation between language specification and implementation aspects.

NEED FOR PYPY

Objective of the PyPy project is to create a new implementation of Python that

i. Has a smaller memory footprint; and

ii. Runs applications faster than the current interpreter; and

iii. Is well suited to applications making significant use of lightweight threads and continuations.

A possible side-effect of implementing python in python is that python may become impressively modular -- leading to its use in small and embedded platforms. However, this was only a possible side-benefit, and was not the primary focus of the project.

HIGH LEVEL VIEW

The PyPy interpreter itself is written in a restricted subset of Python, called RPython. RPython puts some constraints on the Python language like making a variable’s type be inferred at compile time. The PyPy tool-chain analyses RPython code and translates it into C code, which is then compiled to produce a native interpreter. Finally, the JIT generator builds a JIT compiler into the interpreter, given a few annotations in the interpreter source code.

To achieve the objectives that it set out for the PyPy project:

1. Rewrote the 'front-end' of the Python interpreter, specifically the language parser, lexical analyser and compiler byte-code generator, in python itself.

2. The byte-code interpreter is a program that reads the byte-code stream as data and processes it to do ‘real work'. In order to meet objective (iii), a new VM that employs structure and mechanisms seen in the ‘stackless' implementation was implemented.

IMPLEMENTATION DETAILS

Overview

The PyPy parser includes a tokenizer and a recursive descent parser. (lexing and parsing are clubbed together under the parsing phase)

The parser is a simple LL(1) parser that is similar to CPython’s.

The PyPy Lexer(lexer.py)

Tokenizer

The tokenizer is implemented as a single function (*generate_tokens*) that builds a list of tokens. The tokens are then fed to the parser.(The function *match_encoding_declaration* returns the declared encoding(None if there wasn’t any specified). The *generate_tokens* function takes a single line as an argument and produces a 5-tuple: token-type, token-string, a tuple(row, column) specifying the row and column where the token ends in the source, and the line on which the token was found. In case a discrepancy with the column is found, an Indentation Error with the eponymous string ‘Unindent does not match any outer indentation level’ is given

The PyPy Parser(parser.py)

Building the Python grammar

The python grammar is built at startup from the original Python specification(as in the original paper by VanRossum). The grammar builder first represents the grammar as rules corresponding to a set of Nondeterministic Finite Automatons (NFAs). It then converts them to a set of Deterministic Finite Automatons (DFAs). Finally, the grammar builder assigns each DFA state a number and packs them into a list for the parser to use. The final product is an instance of the *Grammar* class.

Parser implementation

The workhorse of the parser is the *add_token* method of the *Parser* class. It tries to find a transition from the current state to another state based on the token it receives as a argument. If it can't find a transition, it checks if the current state is accepting. If it's not, a *ParseError* is raised. When parsing is done without error, the parser has built a tree of *Node*.

Parsing Python

The glue code between the tokenizer and the parser as well as extra Python specific code is in pyparse.py. The *parse_source* method takes a string of Python code and returns the parse tree.

Compiler

The next step in generating Python bytecode is converting the parse tree into an Abstract Syntax Tree (AST).

Building AST

Python's AST is described in Python.asdl. From this definition, asdl_py.py generates ast.py, which RPython classes for the compiler as well as bindings to application level code for the AST.

astbuilder.py is responsible for converting parse trees into AST. It walks down the parse tree building nodes as it goes. The result is a top level *mod* node.

Symbol analysis

Before writing bytecode, a symbol table is built in symtable.py. It determines if every name in the source is local, implicitly global (no global declaration), explicitly global (there's a global declaration of the name in the scope), a cell (the name in used in nested scopes), or free (it's used in a nested function).

REMARKS

Nowadays high level languages allow to write simple code with greater flexibility. Creating applications faster makes people choosing dynamic languages, where you don't need to specify types and waste time to fighting with them (all the boilerplate code about interfaces just to satisfy compilation process). Someone can argue that this behaviour produces buggy code. For those I would say, after Guido van Rossum: "who produces code without test"?

Now, Python is very popular. But it's position is threatened by a competition. Python has a great ecosystem with huge amount of software and community. But it lacks efficient and state of the art runtime, which competition already have. There are a lot of ways to write a fast code with python. The most popular and unfortunately still propagated is to write the hot parts of application in a low level language and then use CPython C API.

All the python shining efficient tools requires a lot of complicated c code, which block other contributors to come in. Now we would like to write fast and beauty python code.

PyPy provides much better architecture for optimization and further language development. PyPy already comes with the solution for most of the Pythons issues:

- State of the art runtime and design
- Speed - PyPy built-in JIT shines. Sometime (actually rarely) can beatt even C!
- glue code - easy handling c libraries with cffi, which is even faster than ctypes in CPython!

PyPy already has support for multiple platforms (x86, 64_x86, ARM)

In short, the future of Python is PyPy and consequently the future of programming, as well.

REFERENCES

http://doc.pypy.org/

https://bitbucket.org/pypy

VanRossum's blog

Robert Zaremba's blog

http://pypy.org/features.html

http://morepypy.blogspot.in/

Wikipedia