Lightweight PyTTR

Timpe Hörig
University of Gothenburg
gustimpho@student.gu.se

Abstract

The current implementation of PyTTR has two major draw backs: First, it is based on Python3.4 and thus, cannot utilize the modern python features. Second, objects and types in PyTTR differ from objects and types in python. However, as PyTTR is a designed as a library, writing simple programs in PyTTR require verbose syntax in python's OOP style. In this paper different approaches on solving these drawbacks are discussed and a solution in form of a PyTTR-Programming languages is presented.

1 Introduction

In PyTTR defining a simple type `T` and assigning that type to an Object `'a'` looks like the following:

```
T = Type()
T.judge('a')

print(T.query('a'))
```

This is already verbose but unfortunately required as the interpretation of a PyTTR program differs from python's own interpretation. Unfortunately, as PyTTR programs become more complex so does their syntax. A simple program that defines a basic type `Real`, adding a witness condition to that type and checking if an object `0.5` belongs to that type `Real` looks like the following:

```
Real = BType('Real')
Real.learn_witness_condition(
  lambda n: isinstance(n,float)
  )

print(Real.query(0.5))
```

This paper provides an interpreter for a PyTTR language that uses lightweight syntax and interprets these PyTTR programs using the original but updated PyTTR library. With that the program from above can be rewritten to the following:

```
BType Real
Real <- lambda n: isinstance(n,float)

0.5 ? Real
```

The following sections cover the update of the original PyTTR library and the construction of the lightweight PyTTR language.

2 PyTTR Library

One of the biggest feature added to python is it's type system. While not being necessary to run a python program, the usage of python's type system drastically improves the quality and extensionality of code. Furthermore, it is to expect that updating the libraries source code does not only enhance it's quality but also allows for a less verbose interface. However, updating all of PyTTR's source code needs a full understanding of PyTTR in order to update but not chance the semantics of it. A start of that can be found in `origin/`. As updating the complete library exceeds the scope of this paper while not providing a simple enough syntax, this paper focuses on a lightweight language for PyTTR that provides a simple syntax to the user but interprets the program using the verbose interface provided by the PyTTR library.

3 Lightweight PyTTR

This Lightweight PyTTR interpreter only handles a subset of PyTTR. Extending it is relatively easy as all the structure needed for it is already there. However, as it is a time consuming work and would exceed the scope of this paper, this paper demonstrates the usefulness of a Lightweight PyTTR with Types and BTypes. Leading to the context free grammar for Lightweight PyTTR show in Figure 1. Where `id` can be any alphanumeric combination starting with a letter and `any` any possible python object.

```
<prog>  -> <newl*> <stmt*> <newl*>
<newl*> -> newline <newl*> |
<stmt*> -> <stmt> newline
  <newl*> <stmt*> |
<stmt>  -> <any :> <id> |
  <any ?> <id> | Type <id> |
  BType <id> | <id> <- <func>
<id>    -> ...
<any :> -> ...
<any ?> -> ...
```

Figure 1: BNF of Lightweight PyTTR

3.1 Lexer

The lexer itself can be found in `util/lexer` and is a simple `longest match` lexer for the provided regexes in `lexer`. For each regex a corresponding `Dataclass` acting as tokens is given. Tokens may or may not contain data. As example, the Token for identifiers `TId` contains the value of the identifier as the value of the `Dataclass` `TId`.

3.2 Parser

The parser can be found in `util/parser` and is an implementation of `The Early Parser`. Provide with a `TokenMatcher` for each Token and a context free grammar, it can parse any given list of Tokens. All left to do for the Lightweight PyTTR language is to provide exactly that, a `Tokenmatcher` for every Token defined in `lexer` and the context free grammar from Figure 1. The parser then parses the token list and creates an abstract syntax tree(AST) out of it. The nodes for this AST are given in `py13_pyttr_AST`. Each node is represented by a `Dataclass`, containing all the information that is needed. Every node of this AST is also equipped with a `__str__` method, that allows pretty printing for Lightweight PyTTR programs.

3.3 Immutable Lists

In `util/immutable_list` a simple immutable list structure is given. This is necessary to hash lists, as hashing mutable lists is not safe.

3.4 Interpreter

The heart of this interpreter lays in `interprete`. Given any Lightweight PyTTR AST, this interpreter iterates of all statements (every line in the Lightweight PyTTR program corresponds to one statement) and matches on each of them. Depending on the match the statement is handled. As for example, the `SJudgment(id, t)` statement is handles as follows: First, the corresponding type to the string `t` is extracted from the environment. Then, the object to judge `id` is parsed. Finally, the method `judge` provided by the interface of the PyTTR library is called on the corresponding type of `t` with the object corresponding to `id`. Util now, the output of this interpretation is simply printed to the terminal, but may be returned for further use.

3.5 Scope of the Lightweight PyTTR Subset

In `py13_pyttr` a small subset of PyTTR can be found. This is replaceable by the complete PyTTR library as it provides the same interface for all implemented features. However, this is not done as the current subset is an up-to-date python3.13 subset that is fully typed and thus, provides a much better interface.

3.6 Running It

This interpreter can be ran using the command line by typing the following command:

```
python3.13 run.py path
```

Where `path` is the path of the Lightweight PyTTR program.

4 Future Work

While most of the hard work for the interpreter is done, it remains to extend Lightweight PyTTR to cover all features provided by the PyTTR library as well as updating the library itself.

5 Conclusion

Interacting and writing PyTTR programs with Lightweight PyTTR is much more readable and faster than doing so through the interface of the PyTTR library directly. However, in order to use Lightweight PyTTR properly, it needs to be extended to cover the full interface of PyTTR. As this is separated from updating the PyTTR library to the latest python version, it still would be useful to do so, as interacting with a typed interface is much easier and more robust. Furthermore, updating the PyTTR library may also introduce a simpler and cleaner interface, as it can take full advantage of python's new features.

5.1 References

Robin Cooper Python implementation of TTR