Writing Interpreter with Pypy Translation Toolchain
Internship midtime report

Léonard de HARO

June 26, 2012

Contents

Introduction

This work consists of my internship at *PLEIAD* in *University of Chile*, this two-months internship being the conclusion of my first year as a computer science student in french school *École Normale Supérieure*. I work under the supervision of Éric Tanter.

The aim of this internship is to write interpreters using the translation toolchain provided by the *Pypy*'s project on ASTs instead of bytecode and figure out how easy it is to obtain efficient JITing VMs for a given language.

As far, I have designed different interpreters for the same easy language. Source code can be found here. Since I both discovered Python and RPython during this internship, the code might be less efficient than what a skilled programmer could expect, even though I hope it's not too messy.

The following present my work after three weeks and the temporary results I obtained.

1 The language and the Interpreter structure

1.1 The `ifF1WAE` language

As a first language to work on, I choosed to use a slightly modified version of the language `F1WAE` described in . The BNF of this language, called `ifF1WAE` can be seen if Figure 1

⟨*file*⟩ ::= ⟨*Def*⟩* ⟨*Prog*⟩ ⟨*Def*⟩*

⟨*Prog*⟩ ::= ⟨*ifF1WAE*⟩

⟨*Def*⟩ ::= '{' '(' ⟨*id*⟩ ⟨*id*⟩ ')' '(' ⟨*F1WAE*⟩ ')' '}'

⟨*ifF1WAE*⟩ :: = ⟨*num*⟩
| '(' ⟨*op*⟩ ⟨*ifF1WAE*⟩ ⟨*ifF1WAE*⟩ ')'
| '(' 'with' '(' ⟨*id*⟩ ⟨*ifF1WAE*⟩ ')' ⟨*ifF1WAE*⟩ ')'
| ⟨*id*⟩
| '(' ⟨*id*⟩ ⟨*ifF1WAE*⟩')'
| '(' 'if' ⟨*ifF1WAE*⟩ ⟨*ifF1WAE*⟩ ⟨*ifF1WAE*⟩ ')'
| '(' ⟨*ifF1WAE*⟩ ')'

⟨*op*⟩ ::= '+' | '-' | '*' | '/' | '%' | '='

⟨*num*⟩ ::= ['0' - '9']$^+$

⟨*id*⟩ ::= ['_', 'a' - 'z', 'A' - 'Z'] ['_', 'a' - 'z', 'A' - 'Z', '0' - '9']*

Figure 1: `ifF1WAE` Grammar

As one can see, it is a very easy and limited language. Parenthesis are mandatory. Recursion is permitted and easily implemented since any function declaration must be done outside the main procedure, which has to be unique. Functions have precisely one argument and both branches are necessary in the `if` statement. Numbers and identifiers follow the regular expression given in Figure 1.

1.2 The Interpreter

The interpreter is AST-based, thus I had to give myself a Python representation. As expected, I used objects and inheritance to implement the AST structure and use it efficiently. The code can be seen in file `treeClass.py`.

I also implemented my own parser, since I came across very late. Doing so, I had to re-implement a few methods that exist in Python but not in RPython. The parser returns an AST corresponding to the program itself and a dictionary associating each function name to its argument name and the AST representing its body. Corresponding code is in `parser.py`.

The interpreter itself works on three object: an expression to evaluate, the environment (a dictionary associating an identifier to its value) and the dictionary of functions. Given that function declarations have to take place outside the main program, the latest is static. The environment and the expression are of course dynamic. The non-obvious semantic is the following:

- `(` op e_1 e_2 `)` $\equiv$ e_1 op e_2.
- '`=`' represents only the equality between two values.
- e_1 '`%`' e_2 is the rest in the euclidian division of e_1 by e_2.
- evaluation of `(with (`x e_1`)` e_2`)` with a given environment `env` produces the evaluation of e_2 with `env + {`x `:` v`}` where v is the evaluation of e_1 with `env`. This means, the interpreter uses eagerness.
- `(` f e_1`)` is the application of function f to argument e_1. The interpreter first evaluate e_1 (say v) and then evaluate the body of f (stocked in the function dictionary) in the environment `{` x `:` v`}` where x is the name of f's argument.
- `(if` e_1 e_2 e_3 `)` $\equiv$ `if` e_1 `!= 0 then` e_2 `else` e_3 in pseudo- code.
  That is, $e_1 \equiv$ `false` if and only if e_1 is evaluated to 0.

2 The evolution of interpreters

The main documentation and examples about Pypy's translation toolchain uses bytecode interpreters, thus naturally iterative. Since I work on ASTs, the intuitive way of programing the interpreter is recursion. But this might make it harder for the JITing VM to recognize loops. Hence, I have designed four interpreters for the same language, representing the spectrum between a completely recursive version toward a completely imperative one. Doing so, I had to learn and use continuations.

2.1 Recursive

The first version designed was the recursive one, since it's the more natural way to work on ASTs. This version is very intuitive and does not require more object than the three ones I already described (expression to evaluate, environment and dictionary of functions).
Files: `RPinterp.py` and `JITRPinterp.py`.

2.2 Continuation Passing Style

The first step toward an iterative version is to introduce continuations. Since RPython does not support `lambdas` and only accept first-order methods, the only way to provide this is to create classes representing the continuations needed. There is a class `Contk` which is a substitute for an abstract class. I then defined a subclass of `Contk` for each continuation needed. This is strongly inspired (if not almost a direct translation) by. Tail-recursion *via* tail-calls is used (difficulties with that will be acknowledged in section 3.3).
Files: `RPinterpCps.py` and `JITRPinterpCps.py`.

2.3 Trampoline

After the CPS version, the next step is to avoid the continuation stack to grow too much, introducing trampoline. I followed and after having implemented class representation of *bouncing*, I made my interpret function return a new `BounceFun` only when an application is encountered. The `trampoline` function itself can be either tail recursive or hidden in the `main` function *via* a loop.
Files: `RPinterpretCpsTrampoline.py`, `JITRPinterpretCpsTrampoline.py`
and `RPinterpretCpsTrampolineIterative.py`.

2.4 Iterative

Still following , I finally reached a completely iterative version. The concrete implementation of this is inspired by and Prolog's interpreter. It uses four loop variable (the dictionary of function being a constant): `var` (temporary register containing the result of the last evaluation), `expr` (the expression currently evaluated), `env` (the current environment of bindings) and `cont` (the next continuation to apply). I had to redefined the continuations classes to respect the iterative form. The loop goes on as long as there is a continuation to apply and match the current expression, updating the loop variables adequately. When a value or a variable is found, the current continuation is applied. Each continuation remembers the next one, so that the updates can be made properly. Once again, I followed except for applications, due to the difference between his and `ifF1WAE` (mainly : only first-order procedure in the latest).
Files : `RPinterpImperative.py` and `JITRPinterpImperative.py`.

3 Difficulties and results

Writing interpreters with nothing to interpret is quite dumb, and since I wanted to test performances, I needed test files. In section 3.1 I explain them. Then, I present the difficulties I encountered to obtain an efficient JITing VM and present the result on my tests.

3.1 Tests

I won't talk here about the very small tests hand-written I used to ensure myself that my interpreters work properly, semantically speaking. Instead, I'll present my generator of file tests.

Since I want to experiment on the JITing VMs, I had to build tests that could ensure tracing. The answer is in `writeProg.py`. This program takes two arguments n and $runs$. It builds a function f so that $f: x \to n \times x$ and a recursive function `run` so that $run(x)$ calculates $f(3)$ x times. The use of 3 is completely arbitrary, it could have been 42. Finally, the program calls `run` on $runs$. To make sure things take time to be done (and make the JITing visible), `f` is not a basic multiplication but a randomly designed tree (from a basic `Arithmetic Expression` language) where each node is an addition and each leaf is either x or 0. See the code for more precisions on the design of the tree.

The resulting program is written in a file `test`*n*`runs`*runs*`.`.

3.2 JITing

As for the translation with the translator toolchain, I used tutorial to add JITing annotations. I didn't find much documentation about it.

3.2.1 Find the loop

The first "difficulty" when annotating a file to produce the JITing VM, is to find the loop. If in most cases it was trivially indicated by the structure of the program, the Trampoline version could lead to two `jit_merge_point`: within the `interpk` method or in the `trampoline` loop (either recursive or iterative). For now, I've chosen to let the merge point within the interpreter function.

3.2.2 Greens and Reds ?

The main difficulty I encountered was to designate the *greens* and *reds* variable. I found this poorly documented, the only informations that I found being two sentences from the tutorial and a mail from pypy-dev mailing list.

In `JITRPinterp.py`, `JITRPinterpCps.py` and `JITRPinterpretCpsTrampoline.py`, I initially put variable `tree` in green and `env` and `funDict` in red.

In `JITRPinterpImperative.py`, only `expr` was green, `var`, `env` and `funDict` being red.

Since the way I had chosen to color variables produced results slower that the non-JITing version, I decided to brute force it: write every combination possible, translate them all and run them on two test files to see if something emerged from it. I firstly did it on the imperative versions, for it was the only one to support `test10000runs10000` without raising a stack overflow, and because this test took long enough to hope see a clear difference. After a night of translation and an hour of tests on `test10000runs2000`, I could select a few ones that seems to have good results and them designate the "winner" by test those on `test10000runs10000`. Here are the results :

Although I still don't understand these coloration who seems to be the most efficient one, I tried to use the corresponding on `JITRPinterp.py` and it failed to be efficient. Another brute force party later, I found out that this is supposedly the most efficient coloration:

3.3 Results

3.3.1 First set of tests

While coding, I used two tests to see the evolution of performance: `test10runs10` to test the good translation and `test10000runs2000` to test the speed. Every version using tail calls (JITing or not) raise a stack overflow on the later. I then learned that this was normal since RPython does not provide implementation of tail-calls. That's why I didn't test them as much as the version at both extremity of the spectrum. It is also interesting to notice that `RPinterpCpsTrampolineIterative.py` seems to be stuck in an infinite loop (doesn't send a result even on `test10runs10`). That's why there is no JITing version of this one.

Given that recursive and iterative versions both passed the bigger test, I used another file to test their limits: `test10000runs10000`. Both JITing and non-JITing recursive versions encountered stack overflow whereas iterative versions passed this test too.

Both recursive and iterative versions seem at last faster when JITing on the tests they pass among the three mentioned. Let's run more tests.

3.3.2 Benchmarks

I have compared four files: iterative and recursive interpreters both in non-JITing and JITing mode. You can find the test script, the file generated and the results file here

Conclusion and acknowledgment