Writing Interpreter with Pypy Translation Toolchain
Internship midtime report

Léonard de HARO

June 27, 2012

Contents

Introduction

This work consists of my internship at *PLEIAD*[1] in *Universidad de Chile*[2], this two-months internship being the conclusion of my first year as a computer science student in french

[1] http://www.pleiad.cl/
[2] http://www.uchile.cl/

school *École Normale Supérieure*[3]. 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 project*[4] 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 on GitHub[5] 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 chose to use a slightly modified version of the language `F1WAE` described in Part II of Shriram Krishnamurthi's book *Programming Languages: Application and Interpretation*[6]. The BNF of this language, called `ifF1WAE` can be seen in Figure 1.

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`.

Since I came across Maciej Fijalkowski's example on BItBucket[7] very lately, I saw that their existed an RPython library for parsing only after having implemented my own parser[8] . When I did it, 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

[3] `http://www.ens.fr/`

[4] `http://www.pypy.org/`

[5] `https://github.com/zebign1/RPython-internship/tree/master/midtimeReport`

[6] The book *Programming Languages: Application and Interpretation* can be freely download from the author's website or, in case the site is unavailable, from here.

[7] `https://bitbucket.org/pypy/example-interpreter/changesets`

[8] I couldn't use any of the existing tool I found since they most probably wouldn't be supported by RPyhton. Another option would have been to use serialization, but neither `Pickle` nor `CPickle` seem to be supported by RPython.

⟨*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

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 (`expr`), the environment (`env`)(a dictionary associating an identifier to its value) and the dictionary of functions (`funDict`). 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 use bytecode[9] interpreters, thus are 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.

All interpreters share the same `treeClass` an `parser` modules, so that difference of performance are exclusively due to their different forms.

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: `RPinterpretRec.py` and `JITRPinterpretRec.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 Shriram Krishnamurthi's interpreter that can be found Chapter 6 of his book. Tail-recursion *via* tail-calls is used (difficulties with that will be acknowledged in section 3.3), and the function interpreting the AST now takes a fourth argument, the continuation (`k`)
Files: `RPinterpretCps.py` and `JITRPinterpretCps.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 Daniel P. Friedman and Mitchell Wand's book *Essential of Programming Language*[10], section 5.2. and after having implemented class representation of *bouncing*, I made my interpret function return a new `BounceFun` only when an

[9] Andrew Brown's tutorial, Maciej Fijalkowski's example or different papers I read.
[10] I used the Third Edition of the book which is not available on the internet.

application is encountered. The `trampoline` function itself can be either tail recursive or hidden in the `main` function *via* a loop.
Files: RPinterpretTrampolineRec.py, JITRPinterpretTrampolineRec.py, RPinterpretTrampolineIter.py and JITRPinterpretTrampolineIter.py.

2.4 Iterative

Still following *EOPL* but adapting it, I finally reached a completely iterative version (section 5.3 of the book). The concrete implementation of this is inspired by Carl Bolz's Prolog interpreter[11]. 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 matches 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.
Files : RPinterpretIter.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*.

[11] https://bitbucket.org/cfbolz/pyrolog

While coding, I used two tests to see the evolution of performance: `test10runs10` to test the good translation, `test10000runs2000` to test the speed and `test10000runs10000` to test the resistance of the stack. You can find those file in the **FirstTests** repertory respectively renamed `FTtest10runs10`, `FTtest100000runs2000` and `FTtest10000runs10000`.

3.2 JITing

As for the translation with the translator toolchain, I used Andrew Brown's tutorial[12] [13] to add JITing annotations. I didn't find much more 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 jiting point within the interpreter function. I've also chosen to use `can_enter_jit`, even if it's not used by Andrew Brown, to ensure the expected behaviour.

3.2.2 A first optimization

Since `funDict` is calculated during parsing and never changed, I declared the `GetFunc` function (doing the same as `funDict[name]`) a pure function in every JITing interpreter.

3.2.3 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 :

> You should declare the variables representing "the program" as green, and the variables representing "the state" as red.[14]

> [...] this boils down to what's constant for the execution of a particular instruction, and what's not. These are called "green" and "red" variables, respectively.[15]

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

[12] Part 1 : http://morepypy.blogspot.co.uk/2011/04/tutorial-writing-interpreter-with-pypy.html

[13] Part 2 : http://morepypy.blogspot.co.uk/2011/04/tutorial-part-2-adding-jit.html

[14] Antonio Coni, on the pypy-dev mailing list (pypy-dev@python.org)

[15] Andrew Brown's tutorial

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

Since the way I had chosen to color variables (mostly based on the first approach) 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. Only four colorations produced results faster or equivalent to the non-JITing version on `test10000runs2000` and two of them emerged from the run on `FTtest10000runs10000`.

- The fastest version is the one were only `funDict` is green, all the loop variables being red.
- Declaring everything red tend do produce similar result than the previous.
- In both case, they run almost twice as fast as the non-JITing version on `FTtests10000runs10000`
- Adding `var` to green variable in the previous two cases produced a result being exactly the average of the non-JITing and the "without-`var`-in-green" counterpart.

These results confirm Andrew Brown's statement. The fact that `var` is modified but not as often as `cont` or `expr` could explain the latest result.

I tried to use the corresponding on `JITRPinterpretRec.py` and it failed to be efficient. Another brute force party later, I found out that these are supposedly the most efficient coloration:

- `env` in green, `funDict` and `expr` in red
- only `expr` in red

Surprisingly, `funDict` alone in green produce a stack overflow on `FTtest10000runs2000`. And this time, `env` in green is mandatory to have efficient results. None of the previous statements seems to stand to explain this.

3.3 Results

3.3.1 First set of tests

What first appeared when I made the first tests (with `FTs` files), is that JITing version had a lot of stack overflow. So much that none would pass `FTtest10000runs2000` when `RPinterpretRec.py` would. The answer to that is to be found in the strange handling of errors by RPython-translated files [16] and my very-defensive programming. At first, every function that I wrote ensured to have valid arguments and raised errors when not. I even

[16] Documented here.

chose to test belonging to a dictionary with an access attempt instead of going through the keys. After realizing that, I skipped to a non-defensive programing-style: the program continues to test validity of arguments, but instead of raising errors, prints what's going on and let the execution crashes in a way or in another.

The other important result of the first test, is the fact that none of the version using tail-call is capable of running `FTtest10000runs2000`, whereas even the interpreted version of the recursive interpreter was able to do so (given enough time). They all produced stack overflow. This is explained by the fact that RPython does not support tail-call efficiently, as I was explained later. By the way, for an unknown reason, iterative trampoline versions seem to be stuck in an infinite loop when their recursive counterpart produces result in satisfying time for `FTtest10runs10`.

3.3.2 Proper tests

I haven't yet been through a proper test phase, mostly because of lack of time. Anyway, there is a script ready in `tests` repository. Although this script uses symbolic links I created for translation purpose and to use `pypy`, with very little modification it can be run on any Unix-based machine (at least, I think so, I'm not an expert in systems but I don't think I have used anything fancy).

The testing process is the following:

- Four files are tested, corresponding to both non-JITing and JITing versions of recursive and iterative interpreters. Cps and trampoline versions are left apart due to their tail-call structure.
- In recursive JITing version, only `env` is green whereas in iterative version, only `funDict` is green. This is due to previously mentioned experiments.
- n and *runs* vary in the set of power of 10 from 10^1 up to 10^4
- Each test is run 10 times.

Conclusion and acknowledgment

Building these took me some time, mostly time spent debugging the source code due to RPyhon limitations and time spent learning how Pypy works. Of course, this didn't took me the whole three weeks, but I don't count learning Python and continuation as time spent or the interpreter itself.

I acknowledge the fact that my tests are not a proof and that they might perfectly be some limit cases. Still, I believe they give a good overview of the capability of different interpreters. I also humbly recognize that some of the above remarks and weakness of the programs might be entirely due to some personal misunderstanding or programing mistakes.

From what I've done so far, it seems that JITing and interpreting is much more efficient on explicitly iterative interpreters. I hope, and believe, running the tests will confirm this. If they do, a first answer to "how easy it is to produce an efficient JITing VM for a language" would be "write a naive recursive-AST-based interpreter and progressively turn it to an iterative one". Even if intermediary steps might decrease the efficiency of the program, I think they represent a good way from recursive semi-efficient interpreters toward efficient iterative ones. I also believe that trying to improve Tail-call based interpreters would represent an amount of work superior to the potential gain: a skilled programmer being potentially able to skip these steps.

Of course, this are not definitive nor scientific conclusions but intuitive and "little-experienced" based conclusions and a lot of work is still to be done to confirm them.