

Writing Interpreter with Pypy Translation Toolchain

Internship midtime report

Léonard de HARO

June 27, 2012

Contents

1	The language and the Interpreter structure	2
1.1	The ifF1WAE language	2
1.2	The Interpreter	2
2	The evolution of interpreters	4
2.1	Recursive	4
2.2	Continuation Passing Style	4
2.3	Trampoline	4
2.4	Iterative	5
3	Difficulties and results	5
3.1	Tests	5
3.2	JITing	6
3.2.1	Find the loop	6
3.2.2	A first optimization	6
3.2.3	Greens and Reds ?	6
3.3	Results	7
3.3.1	First set of tests	7
3.3.2	Benchmarks	7

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 BitBucket⁷ very lately, I saw that there existed an RPython library for parsing only after having implemented my own parser⁸. 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>

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

$$\begin{aligned}
\langle file \rangle &::= \langle Def \rangle^* \langle Prog \rangle \langle Def \rangle^* \\
\langle Prog \rangle &::= \langle ifF1WAE \rangle \\
\langle Def \rangle &::= \{ \langle ' \langle id \rangle \langle id \rangle ' \rangle \langle ' \langle F1WAE \rangle ' \rangle \} \\
\langle ifF1WAE \rangle &::= \langle num \rangle \\
&| \langle ' \langle op \rangle \langle ifF1WAE \rangle \langle ifF1WAE \rangle ' \rangle \\
&| \langle ' \langle \text{with} \rangle \langle ' \langle id \rangle \langle ifF1WAE \rangle ' \rangle \langle ifF1WAE \rangle ' \rangle \\
&| \langle id \rangle \\
&| \langle ' \langle id \rangle \langle ifF1WAE \rangle ' \rangle \\
&| \langle ' \langle \text{if} \rangle \langle ifF1WAE \rangle \langle ifF1WAE \rangle \langle ifF1WAE \rangle ' \rangle \\
&| \langle ' \langle ifF1WAE \rangle ' \rangle \\
\langle op \rangle &::= '+' | '-' | '*' | '/' | '%' | '=' \\
\langle num \rangle &::= ['0' - '9']^+ \\
\langle id \rangle &::= ['_', 'a' - 'z', 'A' - 'Z'] ['_', 'a' - 'z', 'A' - 'Z', '0' - '9']^*
\end{aligned}$$

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 $(\text{with } (x\ e_1)\ e_2)$ 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 e1 e2 e3`) \equiv `if e1 != 0 then e2 else e3` in pseudo- code.
That is, `e1` \equiv `false` if and only if `e1` is evaluated to 0.

2 The evolution of interpreters

The main documentation and examples about Pypy’s translation toolchain use bytecode⁹ interpreters, thus are naturally iterative. Since I work on ASTs, the intuitive way of programming 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` and `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

⁹Andrew Brown’s [tutorial](#), Maciej Fijalkowski’s [example](#) or different papers I read.

¹⁰I 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 $f: x \rightarrow n \times x$ 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 `testnrunsruns`.

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**, **FTtest10000runs2000** 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 `interp` 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.

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

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

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

¹⁵Andrew Brown's **tutorial**

In `JITRPIinterpImperative.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 than 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 where only `funDict` is green, all the loop variables being red.
- Declaring everything red tends to produce similar results than the previous.
- In both cases, 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 `JITRPIinterpretRec.py` and it failed to be efficient. Another brute force party later, I found out that these are supposedly the most efficient colorations:

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

Surprisingly, `funDict` alone in green produces 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

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