From textbook interpreters toward JITing VMs
How to use Pypy's toolchain

Léonard de Haro

September 5, 2012

Contents

Introduction

The Pypy project[1] aims at providing a way of producing Virtual Machines with a Just-In-Time compiler without much effort.

Tutorial or examples of how to use this tool exist within the scientific literature or on Pypy related websites, but all of them use Bytecode-based interpreters when the easier and most intuitive way to design an interpreter uses Abstract Syntax Tree (AST).

This work proposes to show how to go from an AST-based textbook interpreter to an "efficient enough" JITing VM. We will expose the work done during the two months internship that led to this and provide a step-by-step procedure to translate the original interpreter. This report might also serve as documentation for beginners in Pypy or in itself as a feedback to the Pypy developers for them to see how easy it really is to use their software.

You can find the files presented in this report here[2].

1 JITing VMs

1.1 The Pypy project

The Pypy project was originally dedicated to implement a Python interpreter in Python, named *pypy*. It later evolved and led to an other powerful tool, the *translation toolchain*. Now these two elements, along with the *RPython* language — a strict subset of Python designed to be statically typed and used in the translation toolchain (TTC)— are part of the Pypy project. The TTC translates a program written in RPython to C and compiles it to produce an executable. When annotations for JITing instructions are added, a second interpreter is produced during translation(although hidden) that will provide the JITing part.

1.2 Different kind of JITing

There are two ways to get a Just-In-Time interpreter, the method-based JITing or the trace-based JITing. The method-based notices methods that are often used in the interpreted code and compile them to machine code, using the latest instead of the interpretation. The trace-based will notice a path often taken by the interpreter, trace its execution and compile it to machine code.

1.3 Pypy's tracing JIT

Pypy uses the latest at a meta level: it's not the execution of the interpreted program that is traced but the execution of the interpreter itself. It detects "hot loops", that is a path in the code taken often enough, traces its interpretation, optimizes the trace (*via* the second interpreter already mentioned), translates it to machine code and uses this code whenever the same path is taken. To ensure it is the exact same path, *guards* are introduced in the trace and a guard failure produces a skip back to the original interpreter. See the paper *Tracing the Meta-Level: PyPy's Tracing JIT Compiler*[3] for more explanations.

[1] www.pypy.org

[2] https://github.com/ldharo/Internship2012

[3] By Bolz et al. http://dl.acm.org/citation.cfm?id=1565824.1565827

2 Working with Pypy's TTC

2.1 RPython

One of the first difficulties you have to face when using Pypy's TTC is RPython. Being a strict subset of Python, you can debug most of it using the classic CPython interpreter, but a valid Python program is not necessarily a valid RPython program. Hence, you will have to work carefully, starting a translation quite often to check the syntax is correct and the tool you use provided. You can find the RPython documentation here[4].

Here is a list of a few things to keep in mind while coding in RPython, especially when used to Python:

- Multi inheritance is not supported
- Overwritten methods should have a compatible signature, arguments and return value
- Exceptions are only raised if asked for with a `try: ... except XXXX:` structure
- Some library methods are not exactly the same than in Python. For example `my_dict.get(name)` will not work but `my_dict.get(name, DefaultValueToReturnIfNotFound)` will.

2.2 Turning an interpreter into a JITing VM

2.2.1 Main hints

Two functions are important — and sufficient — to turn your program into a JITing one:

- `jit_merge_point` indicates that the interpreter can eventually commute to a registered and compiled trace from this point. Usually placed at the beginning of the interpreting loop.
- `can_enter_jit` indicates that the interpreter is jumping back in the code structure, meaning the beginning of a loop that could possibly need to be traced.

2.2.2 Coloration

In order for the meta-JITing interpreter to recognize a loop, you must indicate, among the interpreter's variables, which represent the lexical position in the program and which are related to the state of the computation. The first one should be declared *green*, the latest *red*. For example, if you are working on bytecode, green variables would be the bytecode itself and the position that the program is reading. On AST, this would be the AST itself.

Something that is known to be constant by the translator cannot be colored without raising an error[5].

The declaration of green and red variables takes place in the initialization of the `JitDriver` object:

[4] `http://doc.pypy.org/en/latest/coding-guide.html#rpython-definition`

[5] Dictionaries are not known constant even when they might be.

```
jitdriver = JitDriver(greens=[], reds=[])
```

When listing these variable, you must be careful to list them in this order : `INT`, `REFS` and then `FLOATS`. No doing this impeach the translation.

2.2.3 Printing the trace

`JitDriver` has other initialization parameters. The other important one is `get_printable_location` which is supposed to be a function taking the green variables and returning a string. This function allows you to actually see the trace produced in the `out` file when using the following command:

```
PYPYLOG=jit-log-opt:out ./myInterpreter-c myFileToInterpret
```

2.2.4 Controlling the trace

For a "hot loop" to be recognize, all green variables must reach a value already seen. Tracing begins when the loop is seen often enough. Tracing stops when you hit the same value of green variables a second time, but it needs to be on the same stack depth level.

There are a few optional figures that modify the tracing behavior. Some of them haven't been mentioned yet because they are not useful in easy examples like the one described afterward. Here is the list of all the thresholds that control the production of a trace[6]:

- `threshold` : number of times a loop has to run for it to become hot
- `function_threshold` : number of times a function must run for it to become traced from start
- `trace_eagerness` : number of times a guard must fail before we start compiling a bridge
- `trace_limit` : number of recorded operations before we abort tracing
- `inlining` : inlining python functions or not (1/0)
- `long_longevity` : a parameter controlling how long loops will be kept before being freed, an estimate
- `max_retrace_guards` : number of extra guards a retrace can cause
- `max_unroll_loops` : number of extra unrollings a loop can cause

`threshold` and `trace_limit` refer to the most basic use of the JITing facility: an interpreter based on a single loop.

These parameters can be changed using the method `set_param` in `pypy.rlib.jit` . You have to call this function outside the loop.

[6]This information can be retrieved in the Pypy source code file pypy/rlib/jit.py available here : https://bitbucket.org/pypy/pypy

2.2.5 Another hint

Another interesting hint is related to optimizing the trace. It is very well described in the paper *Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages*[7] by Bolz et al. Long story short, if you have a method that has no side effects and that is often called with the same arguments, you can optimize its behavior within the traced loop by declaring it elidable *via* the annotation `@elidable` just before its declaration. This leads to the introduction of guards and a compiled version of the execution. This hint can be helped out with the `promote` annotation. When an object won't change in the following execution[8], it can be useful to let the tracer know that. You do it by promoting the object : `promote(object)`. Properly used, the combination of these two annotation can produce a great optimization of the trace. See the related paper for more detailed explanations.

3 `ifF1WAE` : A language with first-order functions

3.1 The language

The `ifF1WAE` language has been designed from the `F1WAE` language in *Programing languages and interpreters*[9] by Shriram Krishnamurthi. It only allows first-order functions. It allows recursion but is only based upon arithmetics. You can see the BNF grammar in the following figure.

```
⟨file⟩ ::= ⟨Def⟩* ⟨Prog⟩ ⟨Def⟩*

⟨Prog⟩ ::= ⟨instr⟩

⟨Def⟩ ::= ‘{’ ‘(’ ⟨id⟩ ⟨id⟩ ‘)’ ⟨instr⟩ ‘}’

⟨instr⟩ :: = ⟨num⟩
  | ‘(’ ⟨op⟩ ⟨instr⟩ ⟨instr⟩ ‘)’
  | ‘(’ ‘with’ ‘(’ ⟨id⟩ ⟨instr⟩ ‘)’ ⟨instr⟩ ‘)’
  | ⟨id⟩
  | ‘(’ ⟨id⟩ ⟨instr⟩‘)’
  | ‘(’ ‘if’ ⟨instr⟩ ⟨instr⟩ ⟨instr⟩ ‘)’
  | ‘(’ ⟨instr⟩ ‘)’

⟨op⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘=’

⟨num⟩ ::= [ ‘0’ - ‘9’ ]+

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

Figure 1: `ifF1WAE` Grammar

[7] http://dl.acm.org/citation.cfm?id=2069181
[8] Hence, *statically constant*
[9] http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/

3.2 The interpreters

Based on *Essential of Programming languages*[10] by Friedman and Wand, different versions of AST-based interpreters have been implemented. All of them share the same hand-written parser[11] and the same tree declaration files. The files are `treeClass.py`[12] and `parser.py`[13]

The parser generates a dictionary of function, linking each name to its code and the name of the arguments. The interpreter function receives as arguments this dictionary, the program to interpret and an environment for variables binding.

Given that there are only first order functions, a call is interpreted with an empty environment filled with a bounding between the argument name and the value the function is called upon.

An elidable function `GetFunc` has been created to optimize lookups in the function dictionary, since it is constant.

The tree and the functions dictionary represents the position within the program, thus are declared green. Everything else within the loop is red. Since the tree is green, all classes representing it must declare their fields *immutable*.

Note on exception handling : we've decided to let the program crash by itself if an error occurs, but we do print the error so that the user know what went wrong.

3.2.1 Recursive version

The recursive version, `InterpretRec.py`[14], is the more basic and naive version of the algorithm.

3.2.2 Continuation Passing Style version

Continuations are introduced, along with tail-recursion. The interpreter called `InterpretCps.py`[15] takes the current continuation as an argument. Unfortunately RPython is not designed to support tail-recursion, so the result is expected to be disastrous.

3.2.3 Trampoline version

Technically speaking this is still a CPS version. We call it `InterpretTramp.py`[16] for trampoline because we introduce the trampoline optimization that avoid the uncontrolled growth of the control stack. We introduce a new object, `Bounce` that encapsulates the tree, the environment and the continuation (the functions

[10] `http://www.eopl3.com/`

[11] Which unfortunately has appeared to be slightly erroneous, but sufficient for the tests.

[12] `https://github.com/ldharo/Internship2012/blob/master/ifF1WAE/firstround/treeClass.py`

[13] `https://github.com/ldharo/Internship2012/blob/master/ifF1WAE/firstround/parser.py`

[14] `https://github.com/ldharo/Internship2012/blob/master/ifF1WAE/firstround/InterpretRec.py`

[15] `https://github.com/ldharo/Internship2012/blob/master/ifF1WAE/firstround/InterpretCps.py`

[16] `https://github.com/ldharo/Internship2012/blob/master/ifF1WAE/firstround/InterpretTramp.py`

dictionary being constant, it does not need to be kept in this object). The new loop does not match the tree but the bouncer.

Problem is, the `Bounce` object contains both green and red variables, so cannot be assigned a color. The solution is to force greens to appear, even if we do not use the bound variables afterward :

```
expr = bouncer.expr
env = bouncer.env
```

3.2.4 Iterative version

`InterpretIter.py`[17] is the last version, completely iterative, with continuations as well. To handle this, a new variable has to be introduced : the register that keeps the last evaluated value in memory. It is, of course, red.

3.3 Benchmarks, first round

No default trace-parameters have been modified for now.

3.3.1 Tests

Tests[18] are generated automatically by a Python program `writeProg.py`[19] that takes two parameters: n and $runs$, writes two functions and a program:

- `f` : a randomly-designed tree that operates $n \times x$ where x is its parameters. Only uses $+$.
- `run` : a recursive function that calls `f 3` each time it is called.
- The program is (`run` $runs$)

3.3.2 Results

A test is designated by its coordinates (k;p) where $n = 10^k$ and $runs = 10^p$. Every interpreter is run 11 times on each test, the first run is ignored, results indicate the average time in seconds of the next 10 runs. A yellow cell indicates the production of a trace. A red one a stack overflow and a blue one both. You can find the results in Figure 2.

Produced files are in repertory results[20] and traces[21].

[17] `https://github.com/ldharo/Internship2012/blob/master/ifF1WAE/firstround/InterpretIter.py`

[18] `https://github.com/ldharo/Internship2012/tree/master/ifF1WAE/firstround/tests`

[19] `https://github.com/ldharo/Internship2012/blob/master/ifF1WAE/firstround/writeProg.py`

[20] `https://github.com/ldharo/Internship2012/tree/master/ifF1WAE/firstround/results`

[21] `https://github.com/ldharo/Internship2012/tree/master/ifF1WAE/firstround/traces`

Interpreter	JIT	(1;3)	(1;4)	(1;5)
Recursive	No	0.00	x 🟥	x 🟥
	Yes	0.01 🟨	x 🟦	x 🟦
Cps	No	x 🟥	x 🟥	x 🟥
	Yes	x 🟥	x 🟥	x 🟥
Trampoline	No	0.00	0.05	0.37
	Yes	0.00	0.04 🟨	0.09 🟨
Iterarive	No	0.00	0.04	0.28
	Yes	0.00	0.02 🟨	0.07 🟨

Interpreter	JIT	(2;3)	(2;4)	(2;5)
Recursive	No	0.01	x 🟥	x 🟥
	Yes	0.06 🟨	x 🟦	x 🟦
Cps	No	x 🟥	x 🟥	x 🟥
	Yes	x 🟥	x 🟥	x 🟥
Trampoline	No	0.04	0.41	3.43
	Yes	0.03	0.65	5.84
Iterarive	No	0.03	0.30	2.54
	Yes	0.03	0.57	0.35 🟨

Interpreter	JIT	(3;3)	(3;4)	(3;5)
Recursive	No	0.13	x 🟥	x 🟥
	Yes	0.62 🟨	x 🟦	x 🟦
Cps	No	x 🟥	x 🟥	x 🟥
	Yes	x 🟥	x 🟥	x 🟥
Trampoline	No	0.40	3.65	35.27
	Yes	0.37	3.64	35.34
Iterarive	No	0.30	2.69	25.77
	Yes	0.29	2.85	27.51

Figure 2: Average execution time in seconds. Yellow cells indicate production of a trace. A red cell indicates a stack overflow, a blue one, both.

3.3.3 Comments on the results

The expected result was that Iterative versions would be the fastest along with the Trampoline, with JITing VMs providing a significant extra - speed in these two cases. CPS and recursive version were expected to be slow if ever efficient and not tracing. Here's the list of oddness are their explanation.

- **Fact** : CPS and Recursive version get stack overflow.
  Explanation : This was half expected. Cps and Recursive version were supposed to be weak. The results shows that recursion (and more specifically tail-recursion) is far from being correctly handled by RPython. In fact the disastrous results of the CPS version means their is no way to make an efficient tail-recursive algorithm in RPython. The Recursive results show, particularly the fact that it produce a trace indicates that recursion is supported, but the stack is bounded to a value between 1000 and 10000.

- **Fact** : The default value for `threshold` is slightly above 1000, hence no trace was supposed to be produced for (k;3). What about the recursive

case, then?
Explanation : A first hypothesis would be that the threshold is over-reached by subtrees of the function that can be found identical in different branches of it, making the counter go above 1000. But a quick look at the trace shows that the whole process is being traced, including the test in the function `run`, very much like any other trace. But the interpretation of the test guarantees that this loop is not seen more than 1001 times, wich is still less than the threshold. Hence, it appears that even thought the TTC seems able to recognize implicit loop hidden in recursion, the very use of recursion damages the counting process of the created VM. Besides, the trace produced by a recursive JITing VM seems to slow the interpretation (based on (2;3)), although the data lacks for a strong affirmation here.

- **Fact** : Biggest cases ($k \geq 2$ for Trampoline, $k \geq 3$ for Iterative) do not produce any trace and so lead to the same (when not worse — see Trampoline (2;5)) efficiency than the non-JITing counterpart.
  Explanation : The `trace_limit` is by default set at 6000. By looking the trace the recursive version produced, you can see many loops withs the sum of the number of operations exceeds 6000. It seems that the recursive structure allows the Recursive version to split the trace in many short or medium-size traces, hence not aborted. Trampoline and Iterative seem to prefer to produce one big trace and so hit the `trace_limit`. This assertion is backed by the difference of time between the Iterative JITing and non-JITing versions that could be explained by the attempt to produce a trace, ending up aborting it.

- **Fact** : Why is (2;4) not traced by Trampoline and Iterative ?
  Explanation : Each file has a different tree generated by the writing program. The structure of the tree in (2;4) must be so that it requires more that 6000 operations to deal with it. In fact, a quickly-written program that evaluates depth and the number of leafs[22] of each tree reveals that despite the fact the maximum depth is approximately the same (23 for (2;4) agains 24 for (2;5)), the number of leafs could explain the difference is complexity between the two trees (225 for (2;4) against 185 for (2;5)).

Nevertheless, some things went just as expected :

- Iterative proved itself to be the fastest and more robust version of the interpreter.

- When a trace is produced, the result can be faster by a factor of 4 (1;5) up to 8 (2;5), and maybe more in other cases. Using JITing on examples that run too fast is not really useful (seen (1;4)).

3.4 Benchmarks, second round

3.4.1 Modifications

First of all, the `trace_limit` is increased to 25000 — enough for the interesting tests. Recursive and CPS versions are dropped. To control the expected length

[22]The tree being randomly designed, it can be that a branch is supposed to have 0 nodes, resulting in a leaf with the number 0. The number of times this happen changes the number of leafs and the complexity of a tree.

Interpreter	JIT	(6;3)	(6;4)	(6;5)
Trampoline	No	0.01	0.14	1.39
	Yes	0.01	0.12	0.32
Iterative	No	0.02	0,12	1.06
	Yes	0.01	0.06	0.22

Interpreter	JIT	(8;3)	(8;4)	(8;5)
Trampoline	No	0.05	0.53	5.27
	Yes	0.05	0.62	1.98
Iterative	No	0.06	0.41	3.96
	Yes	0.04	0.29	0.87

Interpreter	JIT	(10;3)	(10;4)	(10;5)
Trampoline	No	0.23	2.10	20.74
	Yes	0.20	3.32	34.03
Iterative	No	0.18	1.56	15.49
	Yes	0.16	2.88	29.55

Figure 3: Average execution time in seconds. Yellow cells indicate the production of a trace.

of the trace, the randomness of the test function structure is forgotten as well. We now deal with complete binary tree of depth *depth* where inner nodes are + operations and leafs the argument x. The result of the function is unchanged ($f(x) = x \times n$) with $n = 2^{depth}$.

We also introduce a new way of implementing the environment, based on the idea showed in the *Runtime Feedback* paper. We then compare this design we built-in dictionaries.

Files used or generated by this are to be found here[23]

3.4.2 Results

The new coordinates are (k;p) where p's signification is unchanged and $depth = k$. You can find the results in Figure 3.

3.4.3 Comments on the results

Again, we can see how efficient a tracing interpreter is. We can also on (10;p) cases that extending the trace capacity could lead to relatively important slowdown in the VM's performance: aborting tracing later means more time spending recording for nothing. The iterative version definitely proves itself more efficient.

Another interesting thing to notice is that when looking at the number of operation recorded by traces, they approximately follow linearly the extension of the tree in the user's program. Which explains why the case (10;p) does not produce a trace the length of the recorded loop in (8;p) being slightly under 5000. Surprisingly enough, when we rised `trace_limit` up to 40000 in the JITing Iterative version, a trace was produced for both (10;4) and (10;5) with a little less than 19000 operations and a running time of approximately 1.9s and 5s. Unfortunately we couldn't run proper tests to assert these.

[23] https://github.com/ldharo/Internship2012/tree/master/ifF1WAE/secondround

It is also remarkable how traces in case (k;4) and (k;5) are identical. In fact it makes sense given the structure of the test files and the fact that the tracing JIT does not, unless told to, remember the value of the identifier, so despite the difference in *runs*, the path taken by the interpreter in the `if` condition of the recursive function is the same and so is the trace.

4 `RCFAE` : a language with higher-order functions

`ifF1WAE` only accepted first order functions. The next step is naturally to implement higher order functions. That's what RCFAE does. This language can be found in *PLAI*.

4.1 The language

Figure 4 shows the BNF of this language. The equivalence between `{ with { y e} g }` and `{ {fun { y } g } e}` allows us not to implement the with, although that would only be syntactic sugar. The `rec` instruction allows recursive declaration, as expected, in a way that `{ rec { f e } g }` implies that `f` is the name of the function and `e` a function declaration.

⟨*rcfae*⟩ :: = ⟨*num*⟩
| ‘{’ ⟨*op*⟩ ⟨*rcfae*⟩ ⟨*rcfae*⟩ ‘}’
| ‘{’ ‘fun’ ‘{’ ⟨*id*⟩ ‘}’ ⟨*instr*⟩ ‘}’
| ⟨*id*⟩
| ‘{’ ⟨*rcfae*⟩ ⟨*rcfae*⟩‘}’
| ‘(’ ‘if0’ ⟨*rcfae*⟩ ⟨*rcfae*⟩ ⟨*rcfae*⟩ ‘}’
| ‘{’ rec ‘{’ ⟨*id*⟩ ⟨*rcfae*⟩ ‘}’ ⟨*rcfae*⟩ ‘}’

⟨*op*⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘=’

⟨*num*⟩ ::= [‘0’ - ‘9’]$^+$

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

Figure 4: `RCFAE` Grammar

4.2 The interpreters

4.2.1 The parser

To do the parser of this language and avoid a bad surprise, we used the module `ebnfparse` from `pypy/rlib/parsing`. Then we had to translate the resulting tree (structure is in `pypy/rlib/parsing/tree.py`) to our own structure. This is not hard and well documented[24]. You can also check this tutorial[25]. The counterpart of using this module is that you won't be able to run your parser

[24] http://doc.pypy.org/en/latest/rlib.html#parsing
[25] https://bitbucket.org/pypy/example-interpreter/overview

on top of CPython and maybe not Pypy either depending on your installation. See `grammar.txt`[26] and `parser.py`[27].

4.2.2 Structure used

This time, there is no static dictionary, so, given the previous results from `ifF1WAE`, it is useless to use a dictionary for the environment. Even more, it is wrong because dictionaries are mutable whereas we specifically don't want a mutable structure here, given that a function has to remember the environment in which it has been declared.

So we had to change the way our environment was designed. We used *PLAI* again to build a `Env` class that provided the appropriate behavior, including the boot-straping part of declaring a recursive function.

4.2.3 The Interpreter itself

We did not provide recursive nor Cps version of the interpreter this time, according to results obtained previously. There are only an iterative (`interpretIter.py`[28]) and a trampoline (`interpretTramp.py`[29]) version. Nevertheless, to reach these version, we found very useful to start with the naive and intuitive recursive version and adapting it step by step. They both have the instruction

```
set_param(jitdriver, "trace_limit", 25000)
```

4.3 Benchmarks

4.3.1 The tests

We used the same structure than for the second round of benchmarks of `ifF1WAE`, the syntax being obviously adapted.

Because we ran out of time, we could not provide a benchmarks as important as for `ifF1WAE`, so we only provide representative cases.

4.3.2 The results

Here are the results of the tests we had time to launch.

4.3.3 Observations

First of all, we can notice that this is faster than `ifF1WAE` and that tracing is easier (no threshold has been modified, so we're in the same configuration as the first round of benchmarks). Then, we can flatter ourselves noticing that we finally have an efficient way of implementing environments. We didn't have time to see through the trace, but maybe it could be worth it to gain more speed.

[26]https://github.com/ldharo/Internship2012/blob/master/RCFAE/benchmarks/grammar.txt
[27]https://github.com/ldharo/Internship2012/blob/master/RCFAE/benchmarks/parser.py
[28]https://github.com/ldharo/Internship2012/blob/master/RCFAE/benchmarks/interpretIter.py
[29]https://github.com/ldharo/Internship2012/blob/master/RCFAE/benchmarks/interpretTramp.py

Interpreter	JIT	(6;4)	(6;5)	(8;4)	(8;5)
Trampoline	No	0.18	1.77	0.69	6.83
	Yes	0.13	0.22	0.56	1.41
Iterative	No	0.16	1.47	0.58	5.63
	Yes	0.04	0.07	0.26	0.39

Figure 5: A few results for `RCFAE`. Average execution time given in seconds. A yellow cell indicates the production of a trace.

5 A guide from a textbook interpreter to an efficient JITing VM

One of the central interest of this work was to provide a somehow automatic way to go from an AST-based interpreter to an efficient JITing machine. Here is a skeleton of what it would be, based on our experience.

5.1 The parser

The module `ebnfparse` is quite powerful to parse a file which BNF is given. We strongly advise to use it. Even if it means debugging trying to launch a translation every time (bugs appear in the first minute of translation so you don't have to wait the whole translation)

5.2 The tree

Even though parsing with `ebnfparse` returns you an AST, it is preferable to use your own structure. This way, you'll have to think of the strange return format of `ebnfparse` only once.

Remember that the AST will be declared green, hence you have to declare the fields *immutable*.

5.3 Environment structure

If your language is functional, you only need to wonder whether you need to keep static information in memory. In that case, use an optimized dictionary. If you need non-mutable structure, you can use something like the environment we used.

If you language is Object Oriented and dynamic, you could find the hints described in the *Runtime Feedback* paper useful.

In any case, think carefully of the constraint on you environment, and adapt yourself (eventually searching for inspiration in existing VMs).

5.4 Writing the interpreter

If the interpreter is already provided in iterative form, then don't hesitate, just translate from the book. We've seen that this was the most efficient way. Otherwise, we recommend that you start coding the recursive version. Then modify it step by step to CPS, introduce Trampoline and if you seek the most efficient way, go to completely iterative. Once again, it depends on your needs, the trampoline version can be sufficient (the Prolog VM written by Carl Friedrich Bolz[30] is in trampoline form).

This exercise needs a good comprehension of what CPS is, but going slowly and carefully, it is quite easy.

Remember that you have to provide an explicit loop, even if you take advantage of the Object Oriented structure of Python to avoid `isinstance` and dispatch the interpreting code as methods of the tree classes.

Do not forget the `entry_point` method, as showed in this tutorial[31].

5.5 Adding JITing annotation

Once you have your final interpreter, before adding JITing annotation, translate it without JITing attributes. Run it on tests and check that it runs properly.

Now you can add the JIT hints. We advise to proceed it that order:

- Declare the `JitPolicy`

  ```
  def jitpolicy(driver):
      from pypy.jit.codewriter.policy import JitPolicy
      return JitPolicy()
  ```

- Create the `JitDriver`, identify green and red variables, code the pretty-printing function if it is not done yet

- Add `jit_merge_point` and `can_enter_jit` where you have to (at the beginning of the loop for the first one, just before jumping back in the user program for the second)

- If you have a static dictionary, optimize its lookups

- Optimize the rest with `elidable` and `promote` and any other hint you might have learn

Finally, try to trace a few examples, check whether there are still some hints you could add and determine the proper tracing parameters for the class of program you want your VM to be designed for.

6 A few remarks and self-criticism

6.1 About Maps

Based on a few evocations here and there (including in mail discussions and comments on other VMs), we tried to optimize the environment representation with Maps. We first re-used the code shown in *Runtime feedback*, adapting the `Instance` structure into an Environment. We used this representation on `ifF1WAE`. It seemed to provide a slight speed up compared to optimized dictionary on small examples, but whenever we tried it with bigger examples of Second Round or with a `trace_limit` extended, the cost would become prohibitive (more than 15 times slower than the non-JITing version!). In fact, the

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

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

```
class Map(object):

    def __init__(self):
        self.values = {}

    @elidable
    def getvalue(self, name):
        return self.values.get(name, ErrorV("Free variable : %s" % name))


    @elidable
    def add_attribute(self, name, value):
        newmap = Map()
        newmap.values.update(self.values)
        newmap.values[name] = value
        return newmap
```

Figure 6: Source code of maps for `RCFAE`

structure in the paper is supposed to represent object whose fields change rarely whereas a environment is likely to change very often.
Since this structure was, as the dictionaries, mutable, (the problem with that is illustrated in `envFailure.py`[32]) we came up with the following implementation for `RCFAE`:

We use such Maps to represent the environment. The environment is promoted whenever one of its method are called, so that the class methods consider it as a constant during their execution :

```
promote(env)
register = env.getvalue(tree.name)
```

or

```
promote(env)
env = env.add_attribute(tree.funName, dummy)
```

This was the occasion to verify that it was indeed the better way to optimize. We created a test file with coordinates (1;5) according to how we built tests in the first round of benchmarks in `ifF1WAE` and compared traces, modifying the presence of two hints: `elidable` and `promote`. You can see the result of the experiment in Figure 7. The files are here[33]

It clearly appears that using `elidable` and `promote` annotations produce, as expected, the best optimized trace, i.e. the one with less operations and so the fastest to be executed. But it was still not fast enough, and the same thing occurred with `RCFAE` than with the other language.

[32] https://github.com/ldharo/Internship2012/blob/master/RCFAE/envFailure.py
[33] https://github.com/ldharo/Internship2012/tree/master/RCFAE/compareTraces

`elidable`	`promote`	`Loop 0`	`Loop 1`	`Loop 2`
No	No	546	x	x
Yes	No	288	39	x
No	Yes	565	232	228
Yes	Yes	170	80	x

Figure 7: Number of operations per loop in different traces.

6.2 Conclusive thoughts

According to what has been done during this internship, a natural conclusion would be the following:

> The Pypy Project provides a proven powerful to design JITing VMs: the Translation Toolchain. Working efficiently both on bytecode-based or AST-based interpreters, this tool allows its user to add a meta-tracing JITing interpreter to the VM, which, when properly designed gives very good performance and beats by far the non-JITing version.
>
> Nevertheless, when one might expect a VM's efficiency to be related to the language itself, the VMs that result from the TTC's utilization seem to guaranty efficiency on a certain class of program instead of the whole language. Even if dynamically modifying the tracing parameters could be considered, it would remain the user's responsibility to provide adequate figures ; this process does not seem acceptable in the prospect of an efficient VM designing. Besides, this also concern the very design of the VM, choosing a structure over another for, say, the environment, could be a good choice in term of coding time but be completely inadequate as soon as interpreted files become tricky.

A lot of things lack for the conclusions drawn previously to be really accurate. First of all, no object oriented language has been implemented and even if the division between first order and higher order remains and that the reference paper *Runtime Feedback* gives a pretty good idea of how to implement a VM for such a language, there might be some difficulties that we did not yet think of.

The second weakness of the experiment is the very restrictive aspect of the languages used. both of them only work on arithmetic expressions and do not give a great latitude for experimenting. Their is no way to be certain that the discussion on the length of the user program remains in a more general case. Thinking this way, what appear to be a weakness of Pypy's TTC — the fact that parameters of the tracing must be changed depending on the efficiency objective and the program interpreted — might be an oddity related to these particular languages. No data can assert it or not. In fact, both C.F. Bolz's Prolog VM Pyrolog and Laurence Tratt's Converge VM[34] use default values of the parameters.

A third weakness would be the programer skills in Python. In fact, Python and RPython had to be learned especially for this work, and many subtleties

[34]https://github.com/ltratt/converge/tree/master/vm

might be lacking. For example, it appeared that the way we designed environment for `RCFAE` was catastrophic when ported to `ifF1WAE`: no tracing ever occurs and JITing VMs using this feature are always slower than their non-JITing counterparts. It is probably be due to a poor understanding of how Pypy's or Python's memory management works.

Anyway, trying to design a JITing VM with the TTC provided by the Pypy's project is a good way of learning about interpreters, continuations and language designing.

Acknowledgment

This work took place under the supervision of Éric Tanter form DCC in Universidad de Chile in Santiago, may he be thanked for his welcoming.

This work would have been poorer without the help of Pypy's developers who helped me when I was stuck with coding issues, and especially Carl Friedrich Bolz to have answered my questions and helped me understand how things worked.

References

[1] Shriram Krishnamurthi. *Programing Languages: Application and Interpreters.* 2003

[2] Daniel P. Friedman and Mitchell Ward. *Essential of Programing Languages*, Third Edition, MIT Press, 2008

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski and Armin Rigo. *Tracing the Meta-Level: PyPy's Tracing JIT Compiler.*

[4] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni and Armin Rigo. *Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages*