Psyco, the Python Specializing Compiler

Armin Rigo *(October-December 2001)*

Psyco resources and the latest version of this page:
http://homepages.ulb.ac.be/~arigo/psyco/.
(This document corresponds to **P**syco 0.3.4.)

This document is an overview of **P**syco, the Python Specializing Compiler. The aim of the **P**syco project is to show that it is possible to execute Python code at speeds approaching that of fully compiled languages. The current prototype operates on i386-compatible processors. Preliminary results show potential speed-ups of a factor between 2 and 100, depending on the kind of code, which means that we can hope for execution speeds closer to fully optimized C than to traditional Python.

More specifically, the goal of **P**syco is to investigate the results one could obtain by automated specialization. Indeed, besides a relatively important but mostly language-independant core, **P**syco is designed as a straightforward rewrite of the existing C interpreter in a higher-level language than C. The core then acts as a specializing compiler on the Python interpreter. So **P**syco can be thought of as a real-world experimentation on the advantage that could have been derived from having written a non-trivial application like the Python interpreter in a higher-level language from the beginning. The "higher-level language" mentionned is currently only conceptual: it is actually C again, but written to be executed at compile-time instead of run-time.

See the file `README.txt` included in the distribution for technical hints, as well as the SourceForge project page at http://sourceforge.net/projects/psyco/.

In more practical terms, **P**syco is a replacement for just the most inner loop of the current CPython implementation of Python, the `eval_frame` function. This means that unlike some previous attempts this not only can be easily integrated into the current implementation, but actually requires it to run. So **P**syco is definitely not the tool you are looking for if all you want is to get rid of the interpreter and distribute fully compiled applications without source.

The name ***P**syco*, or *Python Specializing Compiler*, comes from the fact that the emitted machine code is specialized to specific values of the data, not only to the pseudo-code (a.k.a. bytecode) to interpret.

1 Introduction

An interpreter for a language runs with, as arguments, the pseudo-code to interpret and the value of the variables that this pseudo-code operates on. Following Python's convention, we will call *bytecode* the pseudo-code.

In this point of view, compiling means specializing the interpreter on a specific value of the bytecode; that is, after compilation, we obtain machine code specialized in simulating that specific bytecode. This makes it both faster and less general than the interpreter itself, but these are the only differences.

In an interpreter written in a language like C, there is no difference (at least no one that a computer can tell) between variables holding the bytecode to interpret, and variables holding the actual values on which the bytecode operates. Both the code and the data is at the same level. Thus, to write a compiler from an interpreter, one introduces a differentiation into two levels: *compile-time known* and *run-time known*. We decide that, in the interpreter source code, all variables holding the bytecode are compile-time, and all other variables (the actual Python objects manipulated by the bytecode in this case) are run-time. In other words, we decide that a good way to speed up the interpreter is to *specialize* it to a particular value of the compile-time variables, i.e. to a specific bytecode.

The idea of specializing is fundamental in this document, but not the way a compiler actually chooses *how* to specialize the variable. Indeed, there is in general no best way to choose which variable is compile-time and which one is run-time. Let me give a few examples of differentiations:

- In a usual interpreter, all variables are run-time.
- In a traditional compiler, the source code is compile-time and everything else is run-time (with the exception of a few common optimization, like computing at compile-time arithmetic operations between constants).
- At the other extreme of the usual interpreter, there is the (not quite interesting) case of a compiler that would do all the work itself, and merely emit code that returns the final answer. Of course, such a compiler has to interpret everything itself!

Clearly, none of the above choices works too well for Python.[1] This does not mean that no differentiation can give good results on Python; it merely shows that no *classical* differentiation does, and probably that no *fixed* differentiation does. In **P**syco the level of each variable is not predetermined. Actually, I start by assuming that all variables are run-time, and progressively promoting them

[1]The second choice, the "compiler", corresponds to taking Python sources (or bytecodes), looking at each pseudo-instruction at turn, and dumping the sequel of machine instructions that does the same as what the interpreter would do when encountering this pseudo-instruction. This has the effect of letting each bytecode instruction explode into a large block of machine instructions. Attempts on Python have shown to give about twice the speed of the Python interpreter, at the expense of code size explosion.

to compile-time when for some reason **P**syco guesses it could do a significant optimization if it knew that value at compile-time.

Of course, "significant optimization" is a fuzzy concept. Currently, there are a few simple heuristics, based on the idea that when what comes next depends on an essential way on the value of the variable, it would be a significant optimization to know that value at compile-time. The first thing this applies to is of course the bytecode itself: if we don't know it at compile-time, we are dead. But it also applies, say, to the type pointer inside of Python objects in numerous operations which, like numeric operations, depend essentially on the type. While emitting code, **P**syco manages a list of all the variables and which ones are currently run-time or compile-time.[2]

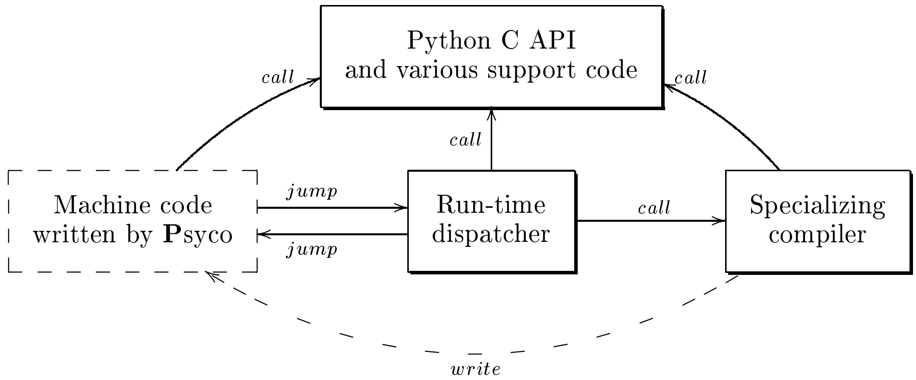
There is actually a third category in **P**syco: virtual-time variables, whose exact value is never physically known, neither at compile-time nor at run-time. They are typical in complex expressions, where the intermediate values should normally be stored in temporary Python objects. Actually building these objects is often not needed; all that is needed is the value that would be put in the fields of the structure if it would have been built. In this situation, the variable pointing to the non-existent structure is virtual-time.

2 Structure of Psyco

Let me first recall that the classical Python implementation has a relatively small "kernel", or "inner loop": the `eval_frame()` bytecode interpreter. This is a C function called whenever we must execute bytecode. The bytecode itself comes from elsewhere (typically, either from the Python source-level compiler or from a `.pyc` file). On the other hand, `eval_frame()` also depends on the rest of the C implementation, because it constantly needs to call other functions to perform the actions required by the bytecode.

Psyco is a substitute for just `eval_frame()`. When called, it will issue its "magic" to optimize speed and reduce the number of outgoing calls (for example, no call to `PyInt_FromLong()` is needed to build temporary integer objects), while still producing a result similar to that of `eval_frame()`. Of course, some of the outgoing calls are essentials and must be done as if we were running the original `eval_frame()` (e.g. to execute a `print` statement).

[2]Could a system use more than two levels of variables, that is, produce code that itself produces code that does the job? This should be explored. I think that currently we lack good heuristics for choosing the levels. This kind of project will probably require us to go to the next step and let the system decide automatically when it should promote variables, by producing code that performs statistical measures at run-time. This is a domain that might require the whole force of projects like Tunes [http://tunes.org].



Psyco consists of three main parts (second row), only the last two of which (in solid frames) are hard-coded in C. The first part, the *machine code*, is dynamically generated. This is the part that is intended to directly substitute for `eval_frame()`. In other words, conceptually, when this machine code is executed, it must perform just like `eval_frame()`.

This machine code contains calls to the C API and a few new supporting functions. But the most interesting point is that the machine code is not completely written in advance — if it were, it would just be the same as `eval_frame()`. Instead, when the machine code prematurely ends, it finishes with a jump to fixed code belonging to the second part of **P**syco, the *run-time dispatcher*.

The role of the run-time dispatcher is to perform appropriate actions to let execution continue. If needed, it calls the compiler to emit new code. Sometimes, it is not needed, because we can discover that appropriate code was already written somewhere else. When found or built, we jump to this new code and execution goes on.

Now let us consider the *compiler* proper. It is responsible for emitting native machine code specialized over some variables. It never directly executes the emitted the machine code blocks; it just emits as much code as possible and returns (usually this means it returns to the dispatcher, which will indeed jump to the just-emitted code, but this is of no concern to the compiler). The code emitted by the compiler usually ends in a jump to the dispatcher, as explained above. Writing "as much code as possible" means until we are blocked — generally by the desire to promote a run-time variable to compile-time so that compile-time optimizations are possible. In effect, run-time variables are not known before the machine code actually executes, so all we can do is write a jump to the dispatcher.

Later, when that jump will actually be executed, the dispatcher will know the actual value and be able to call the compiler again to specialize on this particular value. The dispatcher usually manages a look-up table caching the result of previous such compilations: encountering the same value again later will no longer invoke the compiler. In some cases we can even directly fix the original code with the final jump location, thus short-circuiting the dispatcher.

3 Example

Here is an example, showing step-by-step what occurs when **P**syco runs on the following function:[3]

```
def f(n):
    x = i = 0
    while i<n:
        i = i + 1
        x = x + i
    return x
```

First, when **P**syco initializes itself, it starts by invoking the compiler to produce the most generic machine code, specialized on no value at all. Theoretically this should emit a machine code able to run just anything. In practice this is a very short code. Indeed, when the compiler starts, it realizes that it would be a good idea to know at least the bytecode to interpret, so it promotes it from run-time (everything is run-time at this point) to compile-time. This emits code that just reads the actual bytecode pointer and jumps to the dispatcher. In a complete **P**syco system this small code would then be installed as a replacement to `eval_frame()`; it must currently be explicitely called. Here is the emitted code, loading the code object from the frame object and calling a function of the dispatcher (we do not actually *jump* to the dispatcher only because it is written in C, and C functions insist on returning to their caller, so we *call* this function and jump to the address it returned). (This is GNU `as` syntax.)

```
mov     -4(%ebp),%edx             ; get frame pointer
mov     0xc(%edx),%ebx            ; get bytecode (frame->f_code)
push    %ebx                      ; give it to the dispatcher
push    $0x401a05fc               ; internal data for the dispatcher
call    0x4019b5c0 <Psyco_fast_switch>
add     $0x8,%esp                 ; restore the stack
jmp     *%eax                     ; jump to the sequel
```

Now consider what occurs when we call this small code as, say, `f(2117)`. It jumps to the dispatcher, passing it a pointer to the bytecode of `f` (remember, I am not concerned about how to produce bytecode; Python already does this very well). The dispatcher calls the compiler, which knows about `f` now. Passing through assignments of constant values to `i` and `x`, it turns them into compile-time variables: after the assignments, `i` and `x` must point to the constant object `0`. No machine code is emitted for that.

[3] Yes, I know the example could be trivially optimized to `lambda n:n*(n+1)/2`, but let's pretend we don't know it.

Then there is the test `i<n`. Well, `i` is `0`, but `n` is a run-time variable. It would be great to know the type of `n` at this point, so we promote this type to compile-time. More precisely, this is possible because the compiler internally manages a recursive description of what is run-time or compile-time, so that it can say that `n` points to a structure in which the `ob_type` field is compile-time but not the `ob_ival` field holding the actual integer value.

For the promotion, the compiler concludes the machine code with a jump to the dispatcher and exits. The promotion is a little bit different than the previous one, because we know the cases that will interest us: either the type of `n` is one of the types we know how to optimize (currently only integers), or it is something else. This test can be done by the machine code itself.[4] Here it is:

```
mov     -4(%ebp),%ebx           ; get frame
mov     0x14c(%ebx),%esi        ; n (frame->f_fastlocals[0])
mov     0x4(%esi),%edi          ; n->ob_type
mov     %edi,%eax               ; ***
cmp     $0x80c0ae0,%eax         ; is it &PyInt_Type ?
je      0x8151958               ; if yes, jump there
call    0x814969f               ; otherwise call this
                                ; (hack: the call never returns)
```

(The instruction marked "***" could be optimized out, but simply comes from the fact that the compiler tries to work as fast as possible and sometimes a potential optimization would involve rewriting a large block of code, so is avoided.)

In this case it is an integer, so the conditional jump is followed. It jumps to the dispatcher, calling the compiler again. (When the compiler returns, the target of the conditional jump above will be changed to point directly to the new code block.)

Now that we know that `n` is an integer, we must still compare it with `0`. We just write the corresponding machine instruction. According to Python, the result of the comparison should be stored in a Python integer object, either `0` or `1`. Instead of building a new object, we mark the variable holding it as *virtual-time*, i.e. we know that it could be actually computed if needed, just by calling `PyInt_FromLong()`, depending on the outcome of the comparison — which, remember, is still marked as being in the processor's flags register at this time.

The next instruction in the bytecode is a conditional jump (for the `while`). **P**syco sees that the condition is in a virtual-time value that could be computed from the processor flags. The long path to the jump would be to build a value, 0 or 1, that represents the outcome of the previous comparison, and then test this value. Instead of doing that we can (in this quite common case) emit just a conditional jump instruction testing the correct flags.

[4]What to write to the machine code and what is statically present in the dispatcher must be decided sometimes arbitrarily. In this case we put the tests in the machine code so that we can later modify this code to jump directly to the other blocks of code corresponding to each case.

If you come back to the Python source of `f` you will see that `n` only appears here, and all other variables are compile-time. Normally, this means that everything can be computed by the compiler but the conditional jump on the value of `n`. Without precaution the compiler would produce this kind of code:

```
cmp     $0,%eax
jle     xxxxxxxx       ; produced at the beginning
cmp     $1,%eax
jle     xxxxxxxx       ; produced after one loop
cmp     $2,%eax
jle     xxxxxxxx       ; produced after two loops
cmp     $3,%eax
jle     xxxxxxxx       ; and so on...
...
```

The problem here is that the compiler must go on and on with emitting code because the outcome of the looping condition `i<n` is not known (the value of `n` is run-time) but everything else is compile-time so should be done by the compiler. The targets of each of the above conditional jumps would point to code that simply emits the good answer, as computed this far by the compiler. But of course not only does the compiler never stops (there are actually limits on the size of code the compiler may emit before it is forced to pause); worse, this is not efficient at all. The compiler is actually doing exactly the same thing as an interpreter, with a lot of additional fuss.

The solution here is to realize that compile-time values must be forced into run-time ones. In two words, there are computations that seem to be doable at compile-time, but would simply but so much faster if done at run-time — namely the ones involving loops. This is implemented by detecting that the only difference between the current "state of variables" and an already-seen state is in the value of some compile-time variables, and moreover that so far these values were not essential for the compilation (e.g. we never switched on the values of `i` or `x` — only on their types). In our case this situation occurs after one complete iteration of the loop: we are back in the same state as before the first iteration but the values of `i` and `x` are now `1`. So we "un-promote" the value of one of them from compile-time to run-time. To do so we must emit a machine instruction that loads the current value, i.e. `1`, into some register and go on with compilation by considering that this value is no longer compile-time-known, but only stored in that register. This time, as the next iteration finishes, the variable will still be run-time, and we will reach exactly the same state — no more difference in the compile-time-known value. When the compiler reaches an already-seen state it codes a jump back to the point corresponding to the last time the same state appeared.

This leads to the production of the following code (notice that un-promotion actually occurs after two iterations only):

```
mov     0x8(%esi),%edi      ; get the value of n
```

```
cmp     $0x0,%edi          ; compare it with 0
nop
jle     0x81279c4          ; if n<=0 stop
cmp     $0x1,%edi          ; compare it with 1
nop
jle     0x8157fc4          ; if n<=1 stop
mov     $0x2,%eax          ; unpromote i; at this point n==3
cmp     %edi,%eax          ; compare i and n
nop
jge     0x8158ee4          ; if i>=n stop
push    %eax
add     $0x1,%eax          ; i=i+1
jo      0x401a05ac <PyOverflow_IntegerAdd_proxy>
mov     $0x3,%ecx          ; n=n+i for n==3
add     %eax,%ecx          ; now n is run-time
jo      0x401a05ac <PyOverflow_IntegerAdd_proxy>
@Loop:
cmp     %edi,%eax          ; compare i and n
nop
jge     0x815a0b0          ; if i>=n stop
push    %eax
add     $0x1,%eax          ; i=i+1
jo      0x401a05ac <PyOverflow_IntegerAdd_proxy>
push    %ecx
add     %eax,%ecx          ; n=n+i
jo      0x401a05ac <PyOverflow_IntegerAdd_proxy>
sub     $0xfffffff8,%esp   ; cancel the two 'push'
jmp     @Loop
```

This example contains a lot of unnecessary `push` instructions. This is because no value needs to be saved between one iteration and the next, but the compiler is not smart enough to know that. Also, in this case, the values of `i` and `n` happen to be in the same register at the beginning and at the end of the loop; in general, `jmp` instructions like the final one are preceeded by some shuffle between registers and memory to put the values exactly where they are expected by the code we are jumping to.

Finally, when the loop exits by one of the conditional jumps, **P**syco compiles the following code promoting the variable `x` (as a pointer to a `PyIntObject` structure) from virtual-time to run-time:

```
push    %eax             ; saved for later, just in case
push    %ecx             ; idem
push    %ecx             ; the integer value
call    0x8088720 <PyInt_FromLong>
sub     $-16,%esp        ; remove temporary values from the stack
ret                      ; good bye
```

This finally constructs the integer object in memory, and returns it. Until now all the manipulated integer objects were purely virtual-time pointers.

A final note: if you try afterwards to call the same function `f` with other arguments, the same code will be reused. In fact, the compiler is very fast, so we might run into code explosion problems; currently, no emitted code is ever removed from the memory. This is clearly not a satisfying solution, and need to be addressed in some future version.

4 The compiler

Let us take a closer look at the compiler part of **P**syco, the most interesting one; it is the only part that fundamentally depends on the fact that we are interpreting Python and no other language. Looking at it (file `python_eval.h`) you will find a strong similarity with the code of `eval_frame()` itself. This is because the compiler is best viewed as just the original interpreter in a *two-staged dynamically separated* implementation.

In other words the compiler is obtained as follows:

1. Take the original source code of `eval_frame()`.

2. Replace every variable there by a structure in which you can say either that the variable is compile-time, in which case you give its value as usual, or run-time, in which case there is no value (but there is information on where the value will be at run-time, e.g. in which processor register or where in memory).

3. Before every single operation it does, insert a test that checks whether all the involved variables are compile-time; if they are, proceed with the operation; if they are not, the operation outcome will only be known at run-time, so emit the machine code that will do it at run-time.

4. Tag operations that you consider should always be done at compile-time. Switches seem to be a good choice: if we know the value on which we switch, the emitted code will be quite compact, but if we do not we have no better solution than considering all possible cases. So before all switches, promote the necessary run-time values to compile-time. (Remember, this means returning from the compiler, which will later be called to go on when the value is known.)

5. Optimize all this stuff by finding values that will necessarily end as compile-time, like the Python bytecode, but also all the stack manipulation (because the bytecode is enough to know where objects are pushed or popped from the stack, even if we don't know what these objects actually are).

Notice that this recipe could be implemented as a computer algorithm. This means it is a priori possible to write a meta-program which, given the source code (in a suitably restricted language) of *any* interpreter, would produce a fully functional specializing compiler to be inserted into **P**syco.

The most interesting point lie in the remark that after all there is no need for the source code to be that of an interpreter. There are a number of other things that would benefit from specialization. [XXX references. Look for the Fabius compiler. Difference: in Fabius the two stages are statically chosen, variables do not pass from compile-time to run-time or vice-versa. Advantage: speed (only one or two C instructions needed to emit a machine code instruction, no test needed). Disadvantage: static stages]

5 For more information

The file `README.txt` included with the distribution gives a practical overview of **P**syco and its current state.

Other resources:
http://homepages.ulb.ac.be/~arigo/psyco/.

Armin Rigo.