Structure of **P**syco

Armin Rigo *(November 2001)*

Psyco_resources:
http://homepages.ulb.ac.be/˜arigo/psyco/.

This document describes the structure of **P**syco, the Python Specializing Compiler.

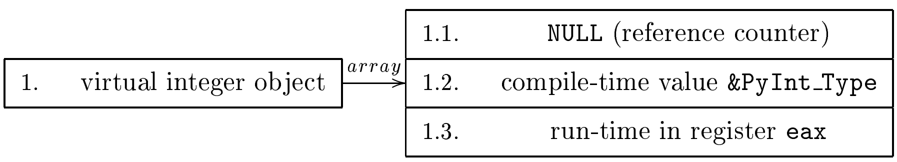
1 The `vinfo_t` data structures

I first describe the most essential data structures that **P**syco_manages: the `PsycoObject` and its `vinfo_t` components.

A `vinfo_t` is a small structure which represents the value and the stage of a particular variable. See `vcompiler.h`. When speaking of a variable I will mean a C-level variable, typically corresponding to a usual C variable of the original Python interpreter or a field in a structure. Do not confuse them with Python variables.

All our variables are untyped and 32-bits; there is no distinction between, say, a `long` or a `PyObject*` variable. (This makes type bugs in **P**syco_a little bit difficult to track. Also, `double`-sized floating-point support will require some tricks.)

To each variable can be associated a substructure, or `array` of variables that this variable points to. Various components of the substructure can be at various stages. For example, a variable holding a `PyObject*` pointing to an integer whose value is currently in the machine register `eax` is described by the following `vinfo_t`s:



(1.1) is `NULL`, meaning that nothing is known about the reference counter. (1) is virtual-time in this example, which is typical of integer objects constructed as the result of arithmetic operations. The array structure matches the structure `PyIntObject`. Most future **P**syco_operations using this structure will discover that the type of the object (1.2) is known to be PyInt_Type and directly read the integer value as described by (1.3), without worrying about the stage of the top-level PyIntObject pointer. On the other hand, if the pointer (1) to the integer object is needed before the object is released, it will be promoted from virtual-time to run-time. This is implemented in pyvirtual.h by calling PyInt_FromLong().

The vinfo_t structures are stored in a PsycoObject, also defined in vcompiler.h, which represents the current state of the compilation. All stage-able variables are stored there, as defined by the macros LOC_xxx in pycompiler.h. The PsycoObject also contains other fields corresponding to "always compile-time" variables.

A few words about reference counters. First, each vinfo_t itself has a reference counter, because the structure of arrays and sub-arrays in the PsycoObject is not a tree, but shares the vinfo_ts. This is typical in LOC_LOCALS_PLUS, which contains both the Python locals and the Python stack; pushing a local to the stack will just add a new reference to the existing vinfo_t at the end of the array of LOC_LOCALS_PLUS.

Do not confuse these reference counters with Python's. **P**syco never attempts to know anything about Python reference counters, but tries to minimize the number of emitted Py_INCREF() and Py_DECREF() instructions: only when the last reference to the vinfo_t is released will a Py_DECREF() instruction be emitted, and only if it is a run-time vinfo_t with the RUNTIME_NO_REF flag not set.

See vcompiler.h for details about the various flags in the various stages.

2 Source files

I will describe the source files by following the control flow. The file psyco.c implements the function proxies described in README.txt in the traditional Python C extension style. Their tp_call slot, invoked when they are called from Python code, is the starting point of **P**syco: it compiles (the beginning of) the "proxified" Python function and executes it. Here is how.

First, psyco_build_frame() is called to make a PsycoObject describing the initial state of the compiler. This includes data such as which Python byte-code object we want to compile and the current instruction pointer (initially at the beginning of the byte-code). The LOC_LOCALS_PLUS array gives the value of all Python variables and the content of the Python stack, as vinfo_t structures. In our case psyco_build_frame() initialize it by assuming that the arguments of the function will be found as PyObject* pointers pushed in the machine stack. (Functions compiled by calls from the *compiled* code might have different argument lists.) The machine code emitted by **P**syco is structured in functions matching Python's, although each function is usually divided in numerous code blocks, each part of the original function corresponding to several differently-compiled code blocks. The run-time call convention used by these emitted functions differs from that of C: the called function saves no register (not even ebp, which has no special meaning) and removes the arguments from the stack at the end.

Next, psyco_compile_code() is called to initiate the compilation. We enter vcompiler.c. First, a word about code buffers. They are managed by codemanager.c. Each code buffer encodes a "snapshot" of the current PsycoObject, a frozen copy of its main data. The snapshot attached to code buffer gives the compiler state at the beginning of the compilation of this code buffer. This is later used by psyco_compatible() to know if we already compiled code for the current state. (More about it in section 5.) So psyco_compile_code() first checks if we have already seen this state, and if not, creates a new code buffer and calls the "compilation entry point" to perform the actual compilation.

In the current version of **P**syco there is only a single compilation entry point left: psyco_pycompiler_mainloop() in pycompiler.c. This function calls main_switch(), the central function: it fetches the Python byte-code to interpret/compile and the current instruction index from the PsycoObject, and switches on the next opcode must like Python's original ceval.c.

The whole loop of main_switch() is quite similar to Python's in ceval.c. The difference is that operations are done on vinfo_ts instead of PyObjects. For simple operations like POP_TOP the translation is quite straightforward. For other operations like PRINT_ITEM we do not optimize anything at compile-time, but simply emit the machine code that will do the job at run-time (often simply by calling a C function whose code I have copied-and-pasted from ceval.c, as for example cimpl_print_item() for PRINT_ITEM).

None of the side-effecting code of ceval.c must be executed by main_switch() itself, as this is only the compilation, not the execution of the Python code. The execution will occur later, after psyco_compiler_code() returned. This occurs when the compilation stops, either because we reached the RETURN_VALUE opcode ending the Python function, or because some other opcode cannot be compiled right now and raises a pseudo-exception. (More about pseudo-exceptions in section 3.) When this occurs, main_switch() returns and psyco_pycompiler_mainloop() performs the usual Python stack unwinding, looking for a loop or exception catching block and performing a function return if none is found. In all cases, when psyco_pycompiler_mainloop() returns, the emitted machine code is completely executable. Then psyco_compile_code() shrinks the allocated memory of the code buffer down to the actual size and returns the code buffer object.

Back in psyco.c, after compilation comes execution. The function psyco_processor_run() of processor.c is a C wrapper to call **P**syco-emitted functions with their particular calling convention.

It is important to note that although we have conceptually separated compilation from execution, they are in practice highly interleaved. One call to psyco_compile_code() returns a complete executable code buffer, but it is almost never the complete function that has been compiled there; only a fragment of it. This code buffer ends with function calls that will indirectly trigger the compilation of the sequel. This is how we can let compilation depend on the actual run-time values computed so far. The function `psyco_finish_call_proxy()` in `processor.c` emits such code which, when executed, calls any previously specified C function. The C function takes an argument which is a pointer to any data it needs to resume compilation (this data was stored at the end of the code buffer itself by `psyco_finish_call_proxy()`) and should return a pointer to the newly compiled code to which we have to jump to go on with execution.

Typically, the called C function will also modify the code that called it so that the next time the execution comes here it directly jumps to the new target.

3 Promotion

Conceptually, "promotion" is easy: when we detect that we could emit better code if we knew the value of a particular variable, we "promote" it from run-time to compile-time, and in the sequel of the compiler we know its value. In practice there is a problem: compilation occurs before execution, so we do not know yet the actual value of the variable. As we cannot go on with compilation, we have to stop it.

This is implemented by raising a pseudo-exception. **P**syco handles Python exceptions in the `LOC_EXCEPTION` field of the `PsycoObject`; as it is also a `vinfo_t`, exceptions are subject to staging just like any other Python variable. In some cases `LOC_EXCEPTION` is used to store a pseudo-exception only. There is a pseudo-exception for each of the particular reasons to unwind the Python stack: returning from the function, a `break` or `continue` statement, etc. (This corresponds to `WHY_xxx` in `ceval.c`. See `WHY_BREAK()`, `WHY_RETURN()` etc. in `pycompiler.c`.)

A special class of pseudo-exception is used for promotion. They stop the compiler as if an unhandled exception had been encountered, but do not unwind the Python stack nor advance the "next instruction" pointer. This special behavior is implemented at the end of the `main_switch()` function, which concludes the machine code buffer by emitting a few instructions using `psyco_finish_promotion()` or `psyco_finish_fixed_switch()`. Each of these two functions, implemented in `dispatcher.c`, calls the generic `psyco_finish_call_proxy()` described above.

The two kinds of promotion are:

1. `psyco_nonfixed_promotion`: any run-time value must be promoted to compile-time. Implemented in `psyco_finish_promotion()` by instructing `psyco_finish_call_proxy()` to call `do_promotion()` at run-time. The latter uses an internal dictionary to look-up already-encountered values, and when a new value is found, compiles the corresponding code with the variable now promoted to compile-time.

2. `fixed_switch_t.fixed_promotion`: only the run-time values in a given list are promoted; all other values are considered as "default" and a single

version of the code is ever compiled for this case. As the set of interesting values is known in advance, we have computed in advance some machine code that performs a binary tree search and jumps to various targets depending on the value found. (Perfect hashing would be better is the lists of values were larger.) In `psyco_finish_fixed_switch()`, this code is copied in the current code buffer by `psyco_write_run_time_switch()`. Then we instruct `psyco_finish_call_proxy()` to have `do_fixed_switch()` called at run-time. The latter searches the actual run-time value in the list and compiles the corresponding version (or the "default" version if the value is not found). Unlike `do_promotion()`, which is called each time the execution reaches that point, `do_fixed_switch()` is only called when the corresponding case has not yet been compiled. Indeed, after compilation, the original code buffer is patched by `psyco_fix_switch_case()` to jump directly to the newly compiled code the next time the same value is found.

"Fixed-switch" promotion (2) is more efficient than "promote-everything" (1). It also let us prove that only a small number of versions of the code can ever be compiled from this point. Nevertheless, "promote-everything" is still interesting in some cases. For example, to test the truth value of a Python object, we promote all types, because we can always emit a relatively efficient version whatever the type is. On the other hand, `BINARY_ADD` only promotes known types, and for all other types it just writes a call to `PyNumber_Add`.

Summary: when we need a promotion, we raise a pseudo-exception that finishes the current code buffer. When the execution arrives here, either `do_promotion()` or `do_fixed_switch()` is called and compiles the sequel by calling `psyco_compile_code()` again. This calls `psyco_pycompiler_mainloop()` and `main_switch()` again.

After the promotion, the `PsycoObject` contains new information about the variable which we wanted to promote: it is now compile-time, with its value specified. (Unless we are in the "default" case of a "fixed-switch" promotion; then the variable is still run-time but has an additional tag to tell it is none of the fixed-switch values — see the macro `KNOWN_TO_BE_DEFAULT()`.)

It is important to realize that the execution of the whole opcode that raised the pseudo-exception is restarted. It will succeed this time, as we know the missing value, but things that were already done are done again. This is why most opcode implementations begin with code like `v = TOP()` to fetch the top value from the Python stack, and not `POP(v)`, which removes it and cannot be re-executed. The `POP()`s are still done but at the end only, when we are sure we succeeded.

As an example, I will describe here what occurs for `BINARY_ADD`. The generic code following the `binary_operator:` label in `main_switch()` uses `dispatcher_dtable()` to find and call a C function specialized on the type of the involved Python objects.

`dispatcher_dtable()` is called with the two objects to add, each in a `vinfo_t`. It inspects their `array->items[OB_TYPE]` specifying what is known about the type. The types for which we have written a specialized C function are described in the file `psydefs.h`. (This file was itself produced by a set of Python scripts. I will not give the details of these scripts now; just have a look at `psydefs.h` to see the result.) The tables `dtx_xxx` used by `dispatcher_dtable()` list types as well as what should be done if the type variables are not known (not compile-time). In most cases, for performance, we should promote the type variable to compile-time.

We see in `psydefs.h` in the table `dt2_operator__add_` that we promote the type of first argument of the addition only if it is `int` or `long`. If it is `int`, then the table `dt2_operator__add__int` tells us that we promote the type of the second argument only again if it is `int` or `long`. Then if it is, say, `int` again, the function `dt2_operator__add__int_int()` is called. This function, implemented in `psydefs.h` as well, can assume that the two arguments are `PyIntObjects` or virtual versions thereof; it reads their `ob_ival` fields to get the integer values and performs the addition, either right now if both arguments are known at compile-time, or by emitting code that will do it at run-time. You can see the use of the special macro `RUN_TIME_CONDITION()`: conceptually it is a check to know whether the addition overflows *at run-time*. How this can be implemented is described in the next section.

4 Respawning

Besides promotions, two techniques are used by the compiler to make choices depending on run-time results not available yet. The first one is implemented by `psyco_coding_pause()` in `vcompiler.c`. This function emits a jump to a small code buffer which, when called, resumes the compilation. This is used to do a "pause" in the compilation: just stop it now, and when later the execution reaches that point, the jump set up by `psyco_coding_pause()` calls `do_resume_coding()` which can go on with compilation. The small code buffer is then released and the jump is modified to point to the newly emitted code, and the net result is quite efficient machine code that no longer jumps to C functions.

`psyco_coding_pause()` can emit conditional jumps as well. This is how `pycompiler.c` sets up code for the `JUMP_IF_TRUE` and `JUMP_IF_FALSE` opcodes. It emits code to do the necessary checks, and calls `psyco_coding_pause()` to conditionally jump to a small coding-resuming buffer. The condition corresponds to one of the branches of the `JUMP_IF_xxx`, so we will jump there at run-time in one of the cases; the other branch is compiled right now, so that if the jump is not taken at run-time execution can just proceed in the same code buffer. If the condition turns out to be satisfied at run-time, the coding pause wakes up and compiles the other branch.

There is yet another technique which is well-suited to branches with one "normal" and one "rather exceptional" case. Such branches are quite common for Python exception, which can be raised by a large number of Python instructions. Coding pauses are (relatively) expensive in memory because they have to store a copy of the `PsycoObject` until we really compile the branch (which may never occur at all, as is common with exception paths). The alternative is "respawning", implemented by the powerful macro `RUN_TIME_CONDITION()` in `dispatcher.h`. This macro conceptually returns `1` or `0` depending on whether the given run-time condition will be true or false. (This shows how stages can conceptually be mixed even in the C code, with macros handling the gory details. It let us write elegant code like `if (condition) then-part; else else-part;` to handle the two cases of a condition that will only be determined at run-time.)

To implement this, the macro returns "false" by default but calls `psyco_prepare_respawn()` to emit a conditional jump. Then the "false" branch is compiled as usual. The "true" branch in the compiler is ignored. But later, if the condition turns out to be true at run-time, the conditional jump is taken. This calls `do_respawn()` in `dispatcher.c`. Then we have to *rebuild* exactly the same compiler state, up to the macro call that tentatively returned "false", and let this macro return "true" this time to compile the other branch.

This could typically be implemented by higher-level language features like coroutines or Python's generators, but no such thing exists in C, and besides we want to avoid the large memory impact of keeping numerous often-unused "restarting" points around. We even do not want to have to store the `PsycoObject`.

This is "respawning": we will come back to a previous well-known point and restart the compilation from there. These well-known points are usually called *safe points*. We have such safe points around: each code buffer's frozen snapshot describes the state of the compiler when it started to write that code buffer. So we "unfreeze" the most recent snapshot and redo the compilation from there. It will re-emit some machine code that has already been emitted, which is not interesting, but it will also rebuild step by step exactly the same state as when the macro `RUN_TIME_CONDITION()` was first called. This time, when it is called again, it detects we are currently respawning: in fact, we just completed the respawning. We call `psyco_respawn_detected()` to trash the re-emitted machine code and start a real code buffer. The macro now returns "true", and the compilation of the missing branch proceeds.

A few gory details (counters etc.) are needed to cope with the fact that `RUN_TIME_CONDITION()` could be used a lot of times between two safe points; we must keep a track of all these calls and let successive calls to `RUN_TIME_CONDITION()` return just the same answers as before, up to the call for which we are respawning. (The implementation is more lightweight than it seems from this description.)

5 Dispatching

"Dispatching" is the set of techniques used to store a trace of the compiled code buffers and find if one already matches the current compiler state. This is currently implemented very basically: all code buffers are put in a Python list, psyco_ge_mainloop.fatlist. State look-up is done in psyco_compatible() in dispatcher.c by comparing the compiler state PsycoObject with each code buffer's frozen snapshot one by one.

This clearly needs to be optimized. To see why the solution is not as obvious as using a dict and hashing the snapshots, let me describe what is exactly meant for a snapshot to match a PsycoObject.

A match can be perfect (psyco_compatible() returns COMPATIBLE in this case). It means that the old code buffer is suitable for reuse, and no compilation is needed.

It does not mean that the two structures of vinfo_ts are exactly the same. The acceptable differences are those which do not prevent the old code buffer from being reused, namely:

1. we can loose information when jumping to the existing buffer, i.e. the snapshot may contain NULLs where the PsycoObject does not;

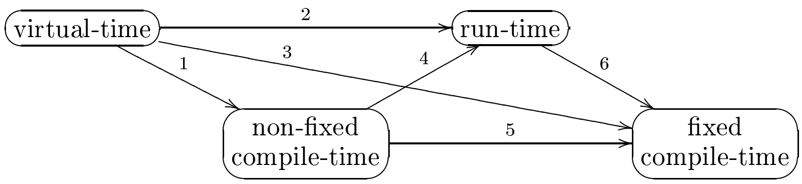
2. a run-time value does not have to be in exactly the same register or machine stack slot; the function psyco_unify() can emit code to move the run-time values from their PsycoObject position to the position specified in the snapshot;

3. there is also the issue of the shared vinfo_ts' graph structure; for more details see the comments in compatible_array().

If no perfect match is found, psyco_compatible() might also return a *partial match*. This is the case where a snapshot matches the PsycoObject excepted for a few vinfo_ts which are *compile-time* in both the snapshot and the PsycoObject but with a different known value. In this case, compilation will be done with the value un-promoted to run-time. This is necessary; otherwise, the same byte-code would be compiled and compiled again with just a different compile-time value in some variables. The situation is quite common in loops: when compiling (say) for i in range(n), i is known to be 0 in the first iteration, then it is known to be 1, then known to be 2, and so on. Each iteration can be very efficiently compiled, but compilation will be as long as the loop! So we detect that the only difference between two iterations is in the value of i, and un-promote i to run-time. This will compile a version of the loop iteration that works for any value of i, and at the next iteration a perfect match will be found.

During compilation, values can be promoted from compile-time to run-time and back. To prevent endless loops, we tag variables promoted from run-time to compile-time; these variables are in what is conceptually another stage: "fixed compile-time". They are never un-promoted to run-time. This is reasonable, because if they were explicitely promoted to compile-time this must be because real optimizations could then be done. (This is the reason why the known value is not stored in the vinfo_t itself but in a small structure, source_known_t, to which the vinfo_t has a pointer. When copies of the vinfo_t are made, they all share the same `source_known_t`. The eventual "fixed" tag that may be added later in the `source_known_t` will then be visible from all other `vinfo_t`s, and most importantly from the ones stored in the snapshot.)

More formally, taking into account virtual-time variables, here are all the possible promotions:



There is clearly no loop. Examples of each promotion type:

1. `pyvirtual.c`: `compute_int()` for a known integer value
2. `pyvirtual.c`: `compute_int()` for a run-time integer value
3. no example (it is just theoretically possible)
4. `dispatcher.c`: `psyco_unfix()`
5. `pycompiler.c`: `dispatcher_dtable()`
6. `dispatcher.c`: `psyco_finish_{promotion,fixed_switch}()`

6 Conclusion

OK, that's it for now. Questions, comments and requests for more info are welcome.

Armin Rigo.