Introducing the PyPy JIT

In part two we will introduce what makes RPython stands out. The automatic generation of a just in time compiler

Just in time compilation

Everyone knows but a quick recap - in certain languages, here Python, the static compilation is a limited approach. One, it's very difficult to prove anything, two a lot of actual information is only available at runtime even with powerful speculations, and three when you compile at runtime, it's much easier to throw away assumptions that have been proven false, so we compile at runtime, usually directly to assembler

Tracing JITs

Tracing JITs are ones where the unit of compilation is a loop instead of a function, but more prominently, they record execution of the interpreter in a special mode (called tracing), then generate an assembler for the trace, after heavy optimizations

tracing meta JITs

The procedure here is as follows - during the compilation of RPython to C, at some carefully chosen stage, the graphs are serialized into so called jitcodes. jitcode is a simple register machine that comes with an interpreter called the metainterp. Metainterp can run the code, but also record the execution of ResOperations. List of resops is then optimized and translated to a given assembler backend

guards and bridges

Each tracing-time if (be it either interpreter-level if, application-level if or something like an indirect call) becomes a guard. Each guard checks if the condition is still met and then jumps out of the assembler code if not. There is complicated logic how to get back into the stage where the processor state is sane as jumping in the middle of a C-compiled function is not a very easy task. If a guard fails enough time, the tracing starts from the position of the guard with everything allocated and the compiled trace is attached into the guard as a new bridge. The process is repeated indefinitely until we run stuff entirely in the assembler code. Note that this means that you can always end up entirely in assembler, regardless of language semantics (with caveats)

simple interpreter example

I'll guide you here through the code and creation of a very small assembler interpreter that let's us experiment with the basics of the JIT

step by step

So let's work it out step by step. First we generate jitcodes. Then the jitcodes are interpreted by the metainterp and generating list of resops. Then it goes through optimization step. There is some light optimization (caching getfields etc.) done already in the tracer, to avoid memory explosion

Cooperating with the JIT generator

As we can see, the difference in performance between optimized and unoptimized interpreter can be tremendous. So let's start walking through the hints I've used and the ones I did not use, but will surely be useful. The hints are few and far apart, but the whole design is done in order to maximize their usefulness

JitDriver

First thing that's obvious is the JIT driver. JIT driver says "this is the main interpreter loop". You can have more than one interpreter loop - the obvious extra ones are something like regexp engine, the less obvious ones are tuple.__eq__ or list.sort which can invoke more python code. can_enter_jit hint enables the JIT to unroll the loop for enough iterations, while finding fix point where the combination of greens is equal

greens & reds

Greens are JIT-time constants, while reds are jit-time variables. Here in our example the greens are bytecode and bytecode position (we want the lookups into that constant-folded away and code specialized for each combination of bytecode & bytecode position), while frame is red - we don't want specialization for each Frame instance

Virtuals and heapcache

The core of optimizations done by PyPy is simple - avoid delay or otherwise deceive allocations and then have other optimizations that enables us to not do them at all. Virtuals are objects which are not allocated at all and kept exploded - but they can survive past bridges. Bookkeeping logic can recreate virtual objects if necessary when a guard fails. Heapcache helps us delay setfields (which can be virtual) as well as cache getfields. We also keep track of what residual calls can and cannot write as well as some aliasing info. Note that this is better than we can do with C, since RPython rules who can write where are quite a bit saner, module well known raw-memory writers like the garbage collector.

interlude - resizable vs non-resizable list

Small interlude - in rpython list which is not-resizable is a bit different than one that's resizable. One is an array, while other has to keep all the extra logic for resizing. As such, it's important to keep some lists non-resizable, like locals or stack in our case.

virtualizables

Virtualizables are probably the most complicated concept in PyPy. I still think they can be done much better, especially for generators and asyncio, but it's a serious research project, for which we could not find funding so far. Virtualizables are meant for frames - they are objects that are not virtual, but can be temporarily or permanently exploded. That means that fields might be out of sync and written directly to the JITFRAME, which is an assembler-level equivalent of stack. It's a GC managed object, so we don't have to sync the frames where the frame survives, but the stack dies. Virtualizables are read at the beginning of assembler and kept in the JITFRAME, so bookkeeping logic can recreate frames if someone calls sys._getframe or does something equally bad. This means locals can always be optimized

```
x = jit.promote(x)
```

Promotion hint means exactly that: please insert a guard_value and then assume it's a constant. It's a specialization hint that we use for example for maps to optimize attribute access in Python.

immutable = True

Immutable hint and _immutable_fields_ = ['a', 'b[*]'] is a hint on class declaration that means this is an immutable object and fields given or arrays (non-resizable) on objects of this class are written only once. It's primarily used for removing field reads and constant-folding the results. A good example is python-level Function object.

@jit.elidable

This function decorator is used for saying "if all the arguments are constant, you can remove the call and replace it with a result doing the call at compile time". There are complicated rules what *exactly* can go there, but all pure functions and functions that are "almost pure", that is they populate some caches and return the answer as if result was always there, are allowed.

exercise

The exercise is "simple" - add a JIT to the existing interpreter. We'll walk through various challenges and try to optimize it. If there is enough time, we might try to add maps to optimize attribute access.