PyPy's VM Approach

Authors: Armin Rigo
Samuele Pedroni

PyPy

- Python VM implementation in Python (a well-chosen subset)
- A translation tool-chain
- Open source project (MIT license)

VMs are still hard

It is hard to achieve:

- flexibility
- maintainability
- performance (needs dynamic compilation techniques)

Especially with limited resources.

Python Case

- CPython is a straightforward, portable VM. No dynamic compilation. Performance is limited.
- Some decisions are pervasive: reference counting, single global lock ...
- Extensions:
  - Stackless (heap-bound recursion, coroutines, serializable continuations)

Python Case (ii)

- Extensions ...:
  - Psyco: run-time specializer, interesting results

  ... need to keep track and are hard to maintain. Hard to port Psyco to other architectures.

- The community wants Python to run everywhere: Jython (Java), IronPython (.NET). Lots of effort and duplication.

PyPy's approach

Goal: generate VMs from a single high-level description of the language, in a retargettable way.

- Write an interpreter for a dynamic language (Python) in a high-level language (Python)
- Leave out low-level details, favour simplicity and flexibility
- Define a mapping to low-level targets, generating VMs from the interpreter

Mapping to low-level targets

- Mechanically translate the interpreter to multiple lower-level targets (C-like, Java, .NET...)
- Insert low-level aspects into the code as required by the target (object layout, memory management...)
- Optionally insert new pervasive features not expressed in the source (continuations, specialization abilities...)

Status of the project

Fully compliant interpreter, translatable to C, LLVM and the CLR.

Maintainability: following the (fast-paced) language evolution is very easy.

Flexibility: we were able to reimplement Stackless features without extensive changes to the baseline interpreter ...

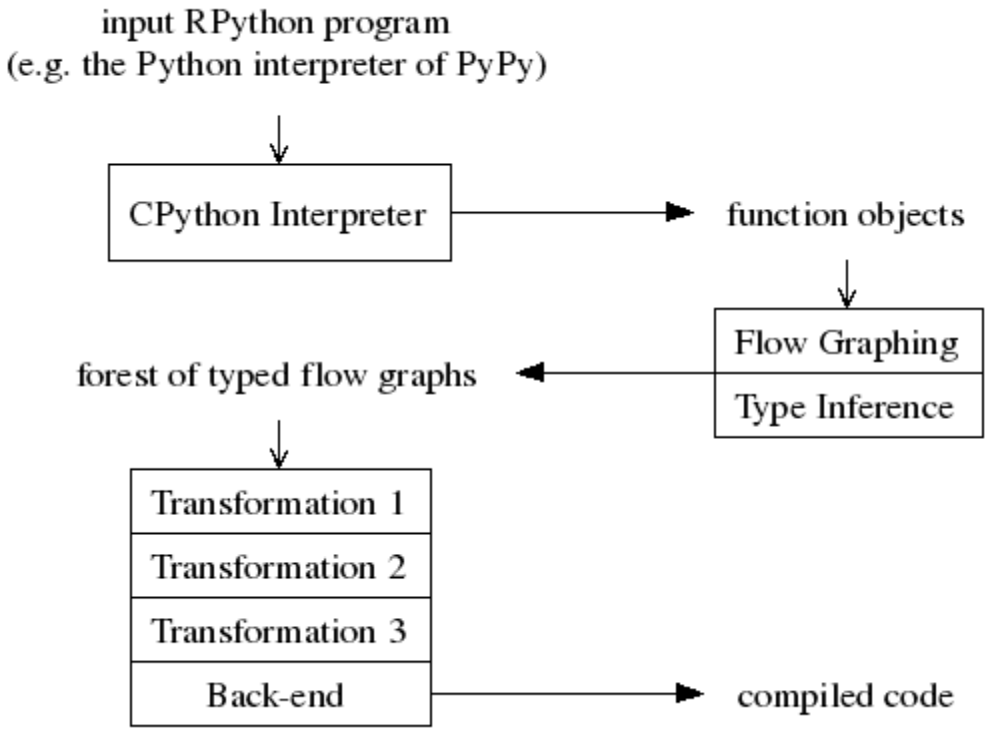
Performance: work in-progress, 2.3 times slower than CPython without dynamic compilation (current goal)

... and many experiments at various levels

Translation approach

- Refine a subset of your favourite language (e.g. Python) amenable to analysis but expressive enough to write interpreters in it.
- Write a translation tool-chain from this subset ("RPython") to multiple targets (C-like, .NET, etc.)
- The translation tool-chain should implement (and be configurable to be) a good mapping from the interpreter to reasonably efficient implementations for the various targets.

Translation overview



Type Inference

- based on abstract interpretation
- fix-point forward propagation
- extensible

Targets as Type Systems

- RPython types (lists, strings, dicts, instances and classes...) may be too high-level for the target (e.g. in C, structs and pointers)
- approach: reflect the essential aspects of a target as a custom type system into RPython (e.g. C-like types)

```
STR = GcStruct('rpy_string', ('hash', Signed),
                             ('chars', Array(Char)))
```

Targets as Type Systems (ii)

- implement a simulation of the types in normal Python, allowing code like to run:

```
def ll_char_mul(char, length):
    newstr = malloc(STR, length)
    newstr.hash = 0
    for i in range(length):
        newstr.chars[i] = char
    return newstr
```

Targets as Type Systems (iii)

- extend the type inferencer to understand usages of these types
- use the type system to express how regular, high-level RPython types should be represented at the level of the target
- write implementation "helper" code (e.g. ll_char_mul) which is again RPython and can be type inferenced and translated

Translation Aspects

Features not present in the source can be added during translation:

- memory management (Boehm, or reference counting by transforming all control flow graphs, or our own GCs - themselves written within the same framework as the RPython "helper" code)

Translation Aspects (ii)

- continuation capture, implemented by saving the low-level frames' local variables into the heap and back
- work in progress: turning an interpreter into a compiler is a translation aspect too (based on binding-time analysis and partial evaluation, extended to use the techniques of Psyco)

Translation Summary

The translation tool-chain has proved effective:

- low-level details and pervasive decision can be left out of the interpreter
- it can targets at the same time: C, LLVM, the CLR and is open for further backends (JVM in progress)
- it can and has been used in the context of other research projects and spin-off ideas (e.g. a JavaScript backend, compilation of other RPython programs...)

Website etc.

- http://codespeak.net/pypy
- IST EU co-funded project in FP6 (7 partners)
- Thanks

Run-time Specialization

Previous experience: Psyco

- a "just-in-time specializer" which can transparently accelerate user code
- a C hand-written "generating extension", in the terminology of partial evaluation
- similar to conventional JITs with the additional ability to suspend compilation at any point, and wait for actual run-time information (e.g. type of an object): **promotion**.

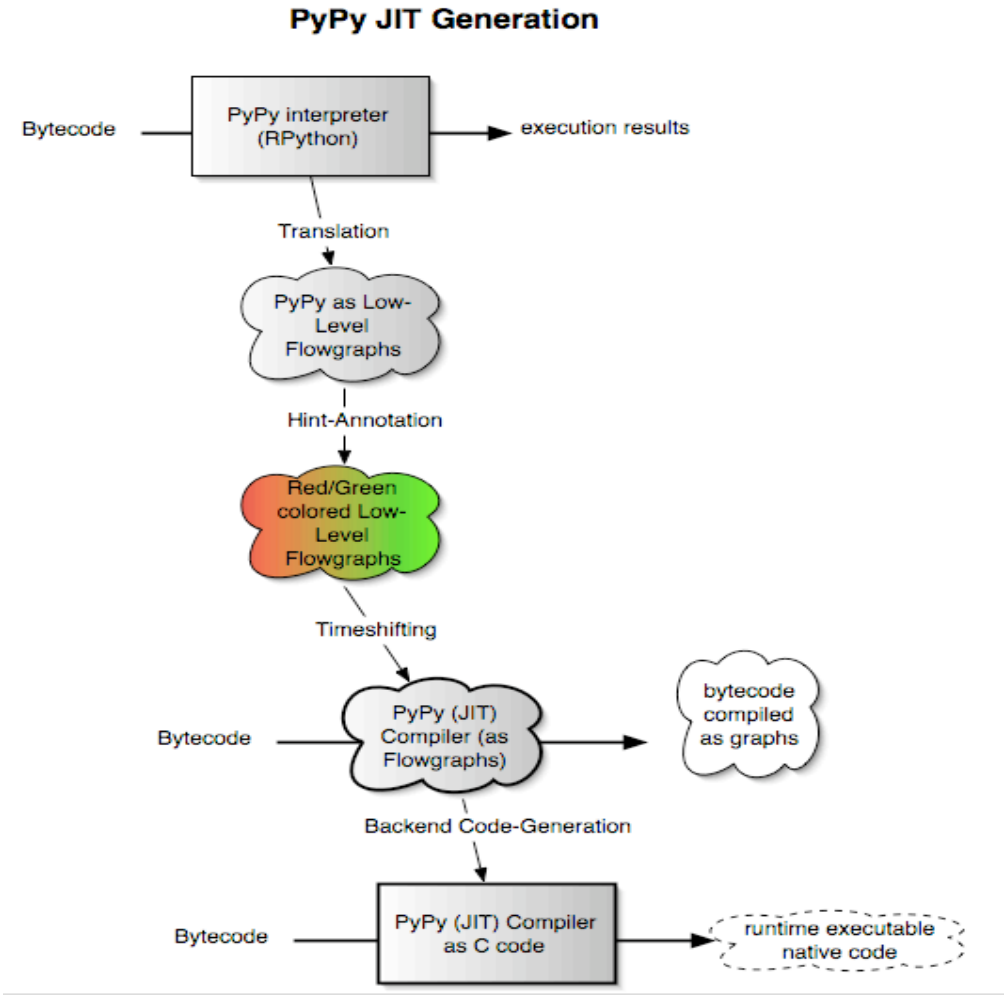
A Specializer as an Aspect

General idea (the devil is in the details):

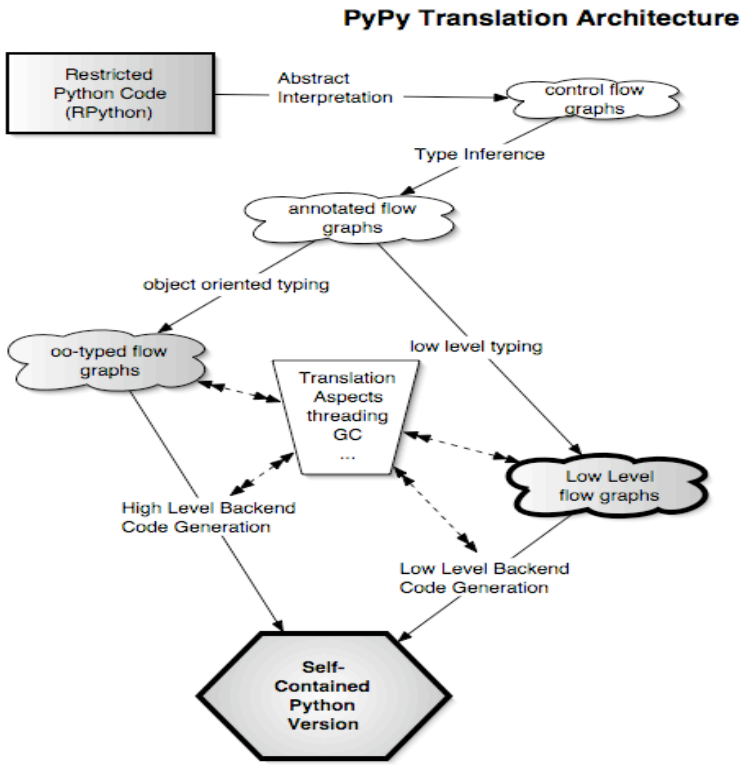
- Transform the flowgraphs of the interpreter into a compiler, using the type inference framework to do binding-time analysis (runtime/ compile-time) based on a few hints.
- Special hints to insert and control promotion.
- We think that promotion is the key to making it practical for large interpreters and complex semantics.

This is what we are working on right now.

JIT Generation Diagram



Translation Diagram



Self-hosted JITs

- they work: Jikes VM
- the language semantics need to be captured into a good compiler
- good means the resulting VM should be fast enough
- target hardware CPUs
- lots of effort still, and hard to reuse for another language

Target platform VMs (JVM, CLR)

- semantics mismatch (e.g. lookup) can result in speed penalty or unnatural code
- how to obliviously layer dynamic compilation on top of a JIT is effectively an open problem
- urge to tweak the underlying VM
- coding in Java, C#: not expressive enough, same risks of inflexibility, hard to revert pervasive decisions

Open Virtual Machines

Reconfigurable at run time to run specific languages.

- Open research area.

- Large design space.
- What are the best primitives?
- Likely same trade-offs in more acute form: need sharp tools.

GC Pressure

RPython is still a garbage collected language.

Large allocation rate from interpreter objects (boxes, frames) but easily temporary objects too.

Good allocation removal optimisations and memory management very much needed.