How to not write Virtual Machines for Dynamic Languages

Carl Friedrich Bolz and Armin Rigo

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

3rd Workshop on Dynamic Languages and Applications,
July 31st 2007

Scope

This talk is about:

- implementing dynamic languages
  (with a focus on complicated ones)
- in a context of limited resources
  (academic, open source, or domain-specific)

Scope

This talk is about:

- implementing dynamic languages
  (with a focus on complicated ones)
- in a context of limited resources
  (academic, open source, or domain-specific)

Our points

- Do not write virtual machines “by hand”
- Instead, write interpreters in high-level languages
- Meta-programming is your friend

Common Approaches to Language Implementation

Using C/C++ (potentially disguised as another language)

- CPython
- Ruby
- Spidermonkey (Mozilla's JavaScript VM)
- but also: Scheme48, Squeak

Common Approaches to Language Implementation

Using C/C++ (potentially disguised as another language)

- CPython
- Ruby
- Spidermonkey (Mozilla's JavaScript VM)
- but also: Scheme48, Squeak

Building on top of a general-purpose OO VM

- Jython, IronPython
- JRuby, IronRuby
- various Prolog, Lisp, even Smalltalk implementations

Implementing VMs in C

When writing a VM in C it is hard to reconcile:

- flexibility, maintainability
- simplicity
- performance

Implementing VMs in C

When writing a VM in C it is hard to reconcile:

- flexibility, maintainability
- simplicity
- performance

Python Case

- CPython is a very simple bytecode VM, performance not great
- Psyco is a just-in-time-specializer, very complex, hard to maintain, but good performance
- Stackless is a fork of CPython adding microthreads. It was never incorporated into CPython for complexity reasons

Fixing of Early Design Decisions

- when starting a VM in C, many design decisions need to be made upfront
- examples: memory management technique, threading model
- such decisions are manifested throughout the VM source
- very hard to change later

Fixing of Early Design Decisions

- when starting a VM in C, many design decisions need to be made upfront
- examples: memory management technique, threading model
- such decisions are manifested throughout the VM source
- very hard to change later

Python Case

- CPython uses reference counting, increfs and decrefs everywhere
- CPython uses OS threads with one global lock, hard to change to lightweight threads or finer locking

Compilers are a bad encoding of Semantics

- to reach good performance levels, dynamic compilation is often needed
- a compiler (obviously) needs to encode language semantics
- this encoding is often obscure and hard to change

Compilers are a bad encoding of Semantics

- to reach good performance levels, dynamic compilation is often needed
- a compiler (obviously) needs to encode language semantics
- this encoding is often obscure and hard to change

Python Case

- Psyco is a dynamic compiler for Python
- synchronizing with CPython's rapid development is a lot of effort
- many of CPython's new features not supported well

Implementing Languages on Top of OO VMs

- users wish to have easy interoperation with the general-purpose OO VMs used by the industry (JVM, CLR)
- therefore re-implementations of the language on the OO VMs are started
- more implementations!
- implementing on top of an OO VM has its own set of benefits of problems

Implementing Languages on Top of OO VMs

- users wish to have easy interoperation with the general-purpose OO VMs used by the industry (JVM, CLR)
- therefore re-implementations of the language on the OO VMs are started
- more implementations!
- implementing on top of an OO VM has its own set of benefits of problems

Python Case

- Jython is a Python-to-Java-bytecode compiler
- IronPython is a Python-to-CLR-bytecode compiler
- both are slightly incompatible with the newest CPython version (especially Jython)

Benefits of implementing on top of OO VMs

- higher level of implementation
- the VM supplies a GC and mostly a JIT
- better interoperability than what the C level provides

Benefits of implementing on top of OO VMs

- higher level of implementation
- the VM supplies a GC and mostly a JIT
- better interoperability than what the C level provides

Python Case

- both Jython and IronPython integrate well with their host OO VM
- Jython has free threading

The problems of OO VMs

- most immediate problem: it can be hard to map concepts of the dynamic language to the host OO VM
- performance is often not improved, and can be very bad, because of the semantic mismatch between the dynamic language and the host VM

The problems of OO VMs

- most immediate problem: it can be hard to map concepts of the dynamic language to the host OO VM
- performance is often not improved, and can be very bad, because of the semantic mismatch between the dynamic language and the host VM

Python Case

- Jython about 5 times slower than CPython
- IronPython is about as fast as CPython (but some stack introspection features missing)
- Python has very different semantics for method calls than Java

Implementation Proliferation

- restrictions of the original implementation lead to re-implementations, forks
- all implementations need to be synchronized with language evolution
- lots of duplicate effort, compatibility problems

Implementation Proliferation

- restrictions of the original implementation lead to re-implementations, forks
- all implementations need to be synchronized with language evolution
- lots of duplicate effort, compatibility problems

Python Case

- several serious implementations: CPython, Stackless, Psyco, Jython, IronPython, PyPy
- various incompatibilities

PyPy's Approach to VM Construction

Goal: achieve flexibility, simplicity and performance together

- Approach: auto-generate VMs from high-level descriptions of the language
- ... using meta-programming techniques and *aspects*
- high-level description: an interpreter written in a high-level language
- ... which we translate (i.e. compile) to a VM running in various target environments, like C/Posix, CLR, JVM

PyPy

- PyPy = Python interpreter written in RPython + translation toolchain for RPython

PyPy

- PyPy = Python interpreter written in RPython + translation toolchain for RPython

What is RPython

- RPython is a subset of Python
- subset chosen in such a way that type-inference can be performed
- still a high-level language (unlike SLang or PreScheme)
- ...really a subset, can't give a small example of code that doesn't just look like Python :-)

Auto-generating VMs

- we need a custom *translation toolchain* to compile the interpreter to a full VM
- many aspects of the final VM are orthogonal from the interpreter source: they are inserted during translation
- translation aspect $\cong$ monads, with more ad-hoc control

Auto-generating VMs

- we need a custom *translation toolchain* to compile the interpreter to a full VM
- many aspects of the final VM are orthogonal from the interpreter source: they are inserted during translation
- translation aspect $\cong$ monads, with more ad-hoc control

Examples

- Garbage Collection strategy
- Threading models (e.g. coroutines with CPS...)
- non-trivial translation aspect: auto-generating a dynamic compiler from the interpreter

Good Points of the Approach

Simplicity:

- dynamic languages can be implemented in a high level language
- separation of language semantics from low-level details
- a single-source-fits-all interpreter
  - runs everywhere with the same semantics
  - no outdated implementations, no ties to any standard platform
  - less duplication of efforts

Good Points of the Approach

Simplicity:

- dynamic languages can be implemented in a high level language
- separation of language semantics from low-level details
- a single-source-fits-all interpreter
  - runs everywhere with the same semantics
  - no outdated implementations, no ties to any standard platform
  - less duplication of efforts

PyPy

arguably the most readable Python implementation so far

Good Points of the Approach

Flexibility at all levels:

- when writing the interpreter (high-level languages rule!)
- when adapting the translation toolchain as necessary
- to break abstraction barriers when necessary

Good Points of the Approach

Flexibility at all levels:

- when writing the interpreter (high-level languages rule!)
- when adapting the translation toolchain as necessary
- to break abstraction barriers when necessary

Example

- boxed integer objects, represented as tagged pointers
- manual system-level RPython code

Good Points of the Approach

Performance:

- “reasonable” performance
- can generate a dynamic compiler from the interpreter (work in progress, 60x faster on very simple Python code)

Good Points of the Approach

Performance:

- “reasonable” performance
- can generate a dynamic compiler from the interpreter (work in progress, 60x faster on very simple Python code)

JIT compiler generator

- almost orthogonal from the interpreter source – applicable to many languages, follows language evolution “for free”
- based on Partial Evaluation techniques
- benefits from a high-level interpreter
- generating a dynamic compiler is easier than generating a static one!

Drawbacks / Open Issues / Further Work

- writing the translation toolchain in the first place takes lots of effort (but it can be reused)
- writing a good GC is still necessary. But: maybe we can reuse existing good GCs (e.g. from the Jikes RVM)?
- dynamic compiler generation seems to work, but needs more efforts.

Conclusion / Meta-Points

- VMs shouldn't be written by hand
- high-level languages are suitable to implement dynamic languages
- doing so has many benefits
- PyPy's concrete approach is not so important
- let's write more meta-programming toolchains!

Questions?

PyPy

http://codespeak.net/pypy/