How to *not* write Virtual Machines for Dynamic Languages

Carl Friedrich Bolz and Armin Rigo

Lehrstuhl Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Düsseldorf, Germany
cfbolz@gmx.de, arigo@tunes.org

Abstract. We argue in this paper that one should not write interpreters for dynamic languages manually but rather use meta-programming techniques and raise the overall level at which they are implemented. We believe this to be ultimately a better investment of efforts than the development of more and more advanced general-purpose object oriented virtual machines (VMs).

Dynamic languages are traditionally implemented by writing a virtual machine centered around an interpreter and/or a built-in compiler and providing the object model and memory management. When a language becomes more popular, the limitations of such an implementation lead to the emergence of alternative implementations that try to solve some of the problems. Another reason for new implementations is the desire to have the language integrate well with existing, well-tuned object-oriented virtual machine like the Java Virtual Machine. In this paper, we describe the mechanisms that lead to an abundance of implementations and explore some of the limitations of standard VMs. We propose a complementary alternative to writing VMs by hand and dealing with low-level details, validated by the PyPy project: flexibly generating virtual machines from a single abstract language "specification", inserting features and low-level details automatically – including good just-in-time compilers tuned to the dynamic language at hand. [1]

1 Introduction

Dynamic languages are traditionally implemented by writing a virtual machine for them in a low-level language like C or in a language that can relatively easily be turned into C. The machine implements an object model supporting the high level dynamic language's objects. It typically provides features like automatic garbage collection. Recent languages like Python, Ruby, Perl and JavaScript have complicated semantics which are most easily mapped to a simple interpreter operating on syntax trees or bytecode; simpler languages like Lisp and Self typically have more efficient implementations based on code generation.

[1] This research was partially supported by the EU funded project: IST 004779 PyPy (PyPy: Implementing Python in Python).

The effort required to build a new virtual machine is relatively large. This is particularly true for languages which are complex and in constant evolution. Language implementation communities from an open-source or academic context have only limited resources. Therefore they cannot afford to have a highly complex implementation and often choose simpler techniques even if that entails lower execution speed. Similarly, fragmentation (for example because of other implementations of the same language) is a problem because it divides available resources. All these points also apply to the implementation of domain-specific languages where it is important to keep the implementation effort small.

For these reasons writing a virtual machine in C is problematic because it forces the language implementer to deal with many low-level details (like garbage collection and threading issues). Limitations of the C implementation lead to alternative implementations which draw resources from the reference implementation. An alternative to writing implementations in C is to build them on top of one of the newer object oriented virtual machines ("OO VM") such as the JVM or the CLR. This is often wanted by the community anyway, since it leads to the ability to re-use the libraries of these platforms. However, if a C implementation existed before the implementation of such a VM is started, this enters into conflict with the goal of having to maintain essentially a single, simple enough implementation for a given programming language: as the language becomes popular, there will be a demand to have it run on various platforms – high-level VMs as well as C-level environments.

In this paper, we will argue that it is possible to benefit from and integrate with OO VMs while keeping the dynamic language implemented with a single, simple source code base. The idea is to write an interpreter for that language in another sufficiently high-level but less dynamic language. This interpreter plays the role of a specification for the dynamic language. With a sufficiently capable translation toolchain we can then generate whole virtual machines from this specification – either wholly custom VMs for C-level operating systems or as a layer on top of various OO VMs. In other words, meta-programming techniques can be used to successfully replace a foreseeable one-VM-fits-all standardization attempt.

The crux of the argument is that VMs for dynamic languages should not be written by hand! The PyPy project is the justification, proving that the approach is feasible in practice. Just as importantly, it also brings new insights and concrete benefits in term of flexibility and performance that go beyond the state of the art.

In section 2 we will explore the way VMs are typically implemented in C and on top of OO VMs and some of the problems of these approaches, using various Python implementations as the main example. In section 3 we will describe our proposed meta-programming approach.

2 Approaches to Dynamic Language Implementation

The observation that limitations of a C-based implementation of a dynamic language leads to the emergence of additional implementations is clear in the case of Python. The reference implementation, *CPython* [18], is a simple recursive interpreter. *Stackless Python* [17] is a fork that adds micro-threading capabilities to Python. One of the reasons for not incorporating it back into CPython was that it was felt that this would make the implementation too complex. Another implementation of the Python language is *Psyco* [13], which adds a JIT-compiler to CPython. Finally, *Jython* is a re-implementation for the Java VM and *IronPython* for the CLR. All of these need to be kept synchronized with the relatively fast evolution of the language.

With the emergence of the CLR and the JVM as interesting language implementation platforms, it is sometimes argued that communities should only develop an implementation of their language for one of these platforms (preferably the argument author's favourite one).

2.1 Assessing the Advantages of Implementing a Language on Top of OO VMs

Implementing a language on top of an existing OO VM is in many ways easier than implementing it in C. Let's take a look at the advantages that are usually cited for basing a language implementation of a dynamic language on a standard object oriented virtual machine, for example the JVM or the CLR.

- *Better interoperability than the C level:* Since the VM offers a standard object model and all the languages implemented on top of it are using it, it is easier to integrate the languages that are running on top of the VM. This allows reuse of libraries between all the implemented languages. This is typically the most important reason to want an implementation on the VM in the first place.
- *Cross-platform portability:* Only the underlying VM has to be ported to various hardware architectures and operating systems. The languages implemented on top of it can then be run without change in various environments.
- *Better tools:* Better IDEs, debuggers and profilers.
- *Better implementation of low-level issues like garbage collection, threading:* Since an OO VM is expected to be widely used and usually backed by a company, it becomes worthwhile and possible to spend a lot of effort tuning its garbage collector, threading model, exception support and other low-level implementation details.
- *Better performance:* Similarly, object-oriented VMs usually come with a highly tuned just-in-time compiler to make them perform well without requiring ahead-of-time compilation to machine language. This, in addition to the previous point, leads to much better performance of the languages running on top of the VM.

- *Ease of implementation:* The implementation of a language on top of an OO VM is easier because it starts at a higher level than C. Usually a high-level language like Java or C# is used for the language implementation, both of which offer the language implementer a much higher level of abstraction than when implementing in C.
- *A single unified implementation base:* The CLR and JVM are trying to position themselves as all-encompassing platforms; if one succeeds, implementations of the dynamic language for other platforms might no longer be required.

The central theme of the benefits of OO VMs is the ability to implement certain hard things only once and share the benefits between all language implementations on top of the OO VM. At a closer look, some of these advantages are only partially true in practice:

- *Better performance:* So far it seems that performance of highly dynamic languages is not actually significantly improved on OO VMs. Jython is around 5 times slower than CPython, for IronPython (which gives up on at least one feature – frame objects – to improve performance) the figures vary but it is mostly within the same order of magnitude as CPython. The most important reason for this is that the VM's JIT compilers are optimized for specific usage patterns that are common in the primary language of the OO VM. To achieve good speeds, the language implementers would have to carefully produce code that matches these usage patterns, which is not a simple task.
- *Better GCs:* While this is obvious in theory, OO VMs tend to have a much higher memory overhead to start with (XXX ref)
- *Cross-platform portability:* While this is true to some extent, the situation with regard to portability is not significantly improved compared to e.g. C/POSIX, which is relatively portable as well. Also, portability sometimes comes at the price of performance, because even if the OO VM is running on a particular hardware architecture it is not clear that the JIT is tuned for this architecture (or working at all), leading to significantly reduced speed.
- *Ease of implementation:* This point is disputable. On the one hand, OO VMs typically allow the language implementer to start at a higher level. On the other hand, they also enforce a specific object and execution model. This means that the concepts of the implemented language need to be mapped to the execution model of the underlying VM, which may or may not be easy, depending very much on the language in question.
  An example where this mapping does not work very well is Prolog. While there exist several implementations of Prolog on top of the JVM [2] [6] and one on .NET [7], they are not particular efficient, especially when compared to good Prolog VMs written in C. This is mostly because the Prolog execution model, which involves backtracking and deep recursion, does not fit the JVM and .NET very well. Therefore the Prolog implementations on top of OO VMs resort to models that are quite unnatural both for the OO VM and for Prolog.

Another important point that makes implementation of languages on top of OO VMs harder is that typically OO VMs don't support meta-programming very well, or do so only at the bytecode level.

Nevertheless, some of the benefits are real and very useful, the most prominent of which being easy interaction with the rest of the VM. Furthermore, there is better tool support and better GCs. Also, for languages where the execution model fits the OO VM well, many of the disadvantages disappear.

2.2 The Cost of Implementation-Proliferation

The described proliferation of language implementations is a large problem for language communities. Although most individual implementations exist for good reasons, the sum of all of them and the need to keep them synchronized with the reference implementations leads to a significant amount of duplicated work and division of effort; this is especially true for open source languages which tend to evolve quickly. At any one point in time some of the implementations will lag behind which makes writing code which can work on all of the implementations harder.

Implementing a language on top of a OO VM has many advantages, so some people propose the solution of standardizing on one particular OO VM to not have to maintain implementations for several of them. While this would, in theory, alleviate the problem it is unlikely to happen. On the one hand, many political issues are involved in such a decision. On the other hand, deciding on a single object and execution model would not be an equally good fit for all languages.

In the next section, we explore a different approach for implementing dynamic languages that we hope is able to solve many of the problems of implementing a language, in particular the problem of an explosion of the number of implementations.

3 Meta-Programming Is Good

The present paper proposes to approach the implementation of dynamic languages from a meta-level: virtual machines for such languages should not be written by hand, but generated automatically "around" an interpreter playing the role of a high-level description of the language. We argue that this approach gives many of the benefits usually expected by an implementer when he decides to target an existing object-oriented virtual machine. It also gives other benefits that we will describe – mostly in term of flexibility. Most importantly, it lets a community write a single source implementation of the language, avoiding the time-consuming task of keeping several of them synchronized. The single source can be used to generate either custom VMs for C-like environments or interpreters running on top of OO VMs. This makes it practical to experiment with large changes to the language and with entirely new languages, such as domain-specific languages, while at any time being able to run the implemented language in a variety of environments, from C/POSIX to the JVM to .NET.

3.1 PyPy architecture

We implemented this idea in the PyPy project [1]. The dynamic language for which we wrote an interpreter is Python. It is a language which, because of its size and rather intricate semantics, is a good target for our approach in the following sense: its previous reimplementations (Jython for the JVM and IronPython for .NET) have each proved to be very time-consuming to maintain. Our implementation is, by construction, easier to maintain and extremely portable (including to C/POSIX, to the JVM and to .NET).

In meta-programming terms, the PyPy architecture is as follows:

- We use a very expressive *object language* (RPython – an analyzable subset of Python) as the language in which the complete Python interpreter is written, together with the implementation of its built-in types. The language is still close to Python, e.g. it is object-oriented, provides rich built-in types and has automatic memory management. In other words, the source code of our complete Python interpreter is mostly free of low-level details – no explicit memory management, no pieces of C or C-level code.
- We use a very expressive metalanguage (namely regular Python) to perform the analysis of RPython code (control flow and data flow construction, type inference, etc.) and its successive transformations.
- This meta-programming component of PyPy is called the *translation framework*, as it translates RPython source code (i.e. the full Python interpreter) into lower-level code. Its purpose is to add aspects to and specialize the interpreter to fit a selectable virtual or hardware runtime environment. This either turns the interpreter into a standalone virtual machine or integrates it into an existing OO VM. The necessary support code – e.g. the garbage collector when targeting C – is itself written in RPython in much the same spirit that the Jikes RVM's GCs are written in Java [3]; as needed, it is translated together with the interpreter to form the final custom VM.

A detailed description of this translation process is beyond the scope of the present paper; it can be found in [15]. The actual Python interpreter of PyPy and the results we achieved by translating it to C, LLVM [10] and .NET are described in [11] [12]. These results show that the approach is practical and gives results whose performance is within the same order of magnitude (within a factor of 2 and improving) of the hand-written, well-tuned CPython, the C reference implementation. These figures do not include the spectacular speed-ups obtained in some cases by the JIT compiler described in section 3.6.

In the sequel, we will focus on the relative advantages and inconveniences of the PyPy approach compared to the approach of hand-writing a language implementation on top of an OO VM.

3.2 A single source

Our approach – a single "meta-written" implementation – naturally leads to language implementations that have various advantages over the "hand-written"

implementations. Firstly, it is a single-source approach – we explicitly seek to solve the problem of proliferation of implementations. In the sequel, we will show that this goal can be achieved without giving up the advantages of hand-written implementations for OO VMs. Moreover, there are additional advantages which, in our opinion, are significant enough to hint that meta-programming, though not widely used in general-purpose programming, is an essential tool in a language implementer's toolbox.

3.3 Writing the interpreter is easier

A first point is that it makes interpreters easy to write, update and generally experiment with. More expressiveness helps at all levels: our Python interpreter is written in RPython as a relatively simple interpreter and is, in some respects, easier to understand than CPython. We are using its high level and flexibility to quickly experiment with features or implementation techniques in ways that would, in a traditional approach, require pervasive changes to the source code. For example, PyPy's Python interpreter can optionally provide lazily computed objects – a 150-lines extension in PyPy that would require global changes in CPython. Further examples can be found in our technical reports; we should notably mention an extension adding a state-of-the-art security model for Python based on data flow tacking [8], and general performance improvements found by extensive experimentation [9], some of which were back-ported to CPython.

3.4 Separation of concerns

At the level of the translation framework, the ability to change or insert new whole-program transformations makes some aspects of the interpreter easier to deal with. By "aspect" we mean, in the original AOP sense, a feature that is added to an object program by a meta-program. The most obvious example in our context is the insertion of a garbage collector (chosen among several available ones) for the target environments that lack it. Another example is the translation of the interpreter into a form of continuation-passing style (CPS), which allows the translated interpreter to provide coroutines even though its source is written in a simple highly recursive style. For more details and other examples of translation-level transformations, see [5].

3.5 The effort of writing a translation toolchain

What are the efforts required to develop a translation toolchain capable of analyzing and transforming the high-level source code and generating lower-level output in various languages?

Although it is able to generate, among other things, a complete, custom VM for C-like environments, we found that the required effort that must be put into the translation toolchain was still much lower than that of writing a good-quality OO VM. A reason is that a translation toolchain operates in a more static way,

which allows it to leverage good C compilers. It is self-supporting: pieces of the implementation can be written in RPython as well and translated along with the rest of the RPython source, and they can all be compiled and optimized by the C compiler. In order to write an OO VM in this style you need to start by assuming an efficient dynamic compiler.

Of course, the translation toolchain, once written, can also be reused to implement other languages, and possibly tailored on a case-by-case basis to fit the specific needs of a language. The process is incremental: we can add more features as needed instead of starting from a maximal up-front design, and gradually improve the quality of the tools, the garbage collectors, the various optimizations, etc.

Writing a good garbage collector remains hard, though. At least, it is easy to experiment with various kind of GCs, so we started by just using the conservative Boehm [4] collector for C and moved up to a range of simple custom collectors – reference counting, mark-and-sweep, etc. Ultimately, though, more advanced GCs will be needed to get the best performance. It seems that RPython, enhanced with support for direct address manipulations, is a good language for writing GCs, so it would be possible for a GC expert to write one for our translation framework. However, this is not the only way to obtain good GCs: existing GCs can also be reused. Good candidates are the GCs written in the Jikes RVM [3]. As they are in Java, it should be relatively straightforward to add a translation step that turns one of them into RPython (or directly into our RPython-level intermediate representation) and integrates it with the rest of the program being translated.

In summary, developing a meta-programming translation toolchain requires work, but it can be done incrementally, it can reuse existing code, and it results in a toolchain that is itself highly reusable and flexible in nature.

3.6 Dynamic compilers

As mentioned above, the performance of the VMs generated by our translation framework are quite acceptable – e.g. the Python VM generated via C code is much faster than Jython running on the best JVMs. Of course, the JIT compilers in these JVMs are essential to achieve even this performance, which further proves the point that writing good OO VMs – especially ones meant to support dynamic languages – is a lot of work.

The deeper problem with the otherwise highly-tuned JIT compilers of the OO VMs is that they are not a very good match for running dynamic languages. It might be possible to tune a general-purpose JIT compiler enough and write the dynamic language implementation accordingly so that most of the bookkeeping work involved in running the dynamic language can be removed – dispatching, boxing, unboxing... However this has not been demonstrated yet.

By far the fastest Python implementation, Psyco [13], contains a hand-written language-specific dynamic compiler. PyPy's translation tool-chain is able to extend the generated VMs with an automatically generated dynamic compiler that uses techniques similar to those of Psyco [14], derived from the interpreter.

This is achieved by a pragmatic application of partial evaluation techniques guided by a few hints added to the source of the interpreter. In other words, it is possible to produce a reasonably good language-specific JIT compiler and insert it into a VM, alongside with the necessary support code and the rest of the regular interpreter.

This result was one of the major goals and motivations for the whole approach. By construction, the JIT stays synchronized with its VM and with the language when it evolves, and any code written in the dynamic language runs correctly under the JIT. Some very simple Python examples run more than 100 times faster. At the time of this writing this is still rather experimental, and the techniques involved are well beyond the scope of the present paper. The reader is referred to [16] for more information.

4 Conclusion

Here are the two central points that we have asserted in the present paper:

- *Do not write dynamic language implementations "by hand".* Writing them more abstractly, at a higher level, has primarily only advantages, among them the avoidance of a proliferation of diverging implementations. Writing interpreters both flexibly and efficiently is difficult and meta-programming is a good way to achieve it. Moreover, this is not incompatible with targeting and benefiting from existing high-quality object-oriented virtual machines like those of the Java and .NET.
- *Do not write VMs "by hand".* Writing language-specific virtual machines is a time-consuming task for medium to large languages. Unless large amounts of resources can be invested, the resulting VMs are bound to have limitations which lead to the emergence of many implementations, a fact that is taxing precisely for a community with limited resources. (This is of course even more true for general-purpose VMs.)

As a better alternative, we advocate a more general usage of meta-programming:

- *Let's write more meta-programming translation toolchains.* Aside from the advantages described in section 3, a translation toolchain need not be standardized for inter-operability but can be tailored to the needs of each project. Diversity is good; there is no need to attempt to standardize on a single OO VM.

The approach we outlined is actually just one in a very large, mostly unexplored design space; it is likely that some of the choices made in PyPy will turn out to be suboptimal. We are hoping that other toolchains will emerge over time, exploring other aspects and proposing other solutions. By their "meta" nature, these multiple approaches should be easier to bridge together than, say, multiple OO VMs with different object and runtime models. We believe that further research in this area might open the door to better solutions for interoperability in

general – e.g. high-level bridges instead of (virtual-)machine-level ones, enabled by cross-translation.

We believe this to be ultimately a better investment of efforts than the development of more advanced general-purpose OO VMs.

References

1. PyPy. An Implementation of Python in Python. http://codespeak.net/pypy.
2. M. Banbara. *Design and Implementation of Linear Logic Programming Languages*. PhD thesis, The Graduate School of Science and Technology of Kobe University, September 2002.
3. S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *ICSE 2004, 26th International Conference on Software Engineering*, 2004.
4. H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.
5. C. F. Bolz and A. Rigo. Support for massive parallelism, optimisation results, practical usages and approaches for translation aspects. Technical Report D07.1, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
6. M. Calejo. InterProlog: Towards a declarative embedding of logic programming in Java. In *JELIA*, pages 714–717, 2004.
7. J. Cook. P#: A concurrent Prolog for the .net framework. In *Software: Practice and Experience 34(9)*, pages 815–845. John Wiley & Sons, Ltd, 2004.
8. A. Cuni, S. Pedroni, A. Chrigström, H. Krekel, G. Wesdorp, and C. F. Bolz. High-level backends and interpreter feature prototypes. Technical Report D12.1, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
9. M. Hudson, A. Rigo, and C. F. Bolz. Core object optimization results. Technical Report D06.1, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
10. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
11. PyPy Team. PyPy - Architecture Overview. Web page. http://codespeak.net/pypy/dist/pypy/doc/archi-tecture.html.
12. PyPy Team. PyPy - Translation. Web page, in-progress. http://codespeak.net/pypy/dist/pypy/doc/trans-lation.html.
13. A. Rigo. Psyco. http://psyco.sourceforge.net/.
14. A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.
15. A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium*, 2006. To appear, accepted for publication.
16. A. Rigo and S. Pedroni. Jit compiler architecture. Technical Report D08.2, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
17. C. Tismer. Stackless python. http://www.stackless.com/.
18. G. van Rossum et al. CPython 2.5.1, April 2007. http://www.python.org/download/releases/2.5.1/.