Back to the future in one week — implementing a Smalltalk VM in PyPy ⋆

Carl Friedrich Bolz[2], Adrian Kuhn[1], Adrian Lienhard[1], Nicholas D. Matsakis[3], Oscar Nierstrasz[1], Lukas Renggli[1], Armin Rigo, and Toon Verwaest[1]

[1] Software Composition Group
University of Bern, Switzerland
[2] Softwaretechnik und Programmiersprachen
Heinrich-Heine-Universität Düsseldorf
[3] ETH Zurich, Switzerland

Abstract. We report on our experiences with the Spy project, including implementation details and benchmark results. Spy is a re-implementation of the Squeak (*i.e.,* Smalltalk-80) VM using the PyPy toolchain. The PyPy project allows code written in RPython, a subset of Python, to be translated to a multitude of different backends and architectures. During the translation, many aspects of the implementation can be independently tuned, such as the garbage collection algorithm or threading implementation. In this way, a whole host of interpreters can be derived from one abstract interpreter definition. Spy aims to bring these benefits to Squeak, allowing for greater portability and, eventually, improved performance. The current Spy codebase is able to run a small set of benchmarks that demonstrate performance superior to many similar Smalltalk VMs, but which still run slower than in Squeak itself. Spy was built from scratch over the course of a week during a joint Squeak-PyPy Sprint in Bern last autumn.

1 Introduction

In this paper we present a preliminary report on the Spy project. Spy is an implementation of the Squeak[4] variant of Smalltalk built using the PyPy toolchain[9]. The goals of the Spy project are to allow the popular Squeak platform to be easily ported to high-level runtimes, such as the Java Virtual Machine (JVM) and Common Language Runtime (CLR), as well as to eventually improve Squeak's performance through the use of PyPy's Just-in-time (JIT) compiler generation techniques. The Spy project also serves to highlight some of the distinctive features in PyPy's approach to building virtual machines, especially when it is compared to Squeak.

Squeak is an open source, full-featured implementation of Smalltalk. One of its distinctive features is that the virtual machine itself is written in Slang[4], a restricted subset of Smalltalk. Slang is designed to be easily translated into C,

⋆ In *Self-Sustaining Systems*, LNCS, vol. 5142, Springer, 2008, pp. 123-139.

meaning that the core VM can be translated into C and then compiled with a standard C compiler. This allows Squeak to enjoy reasonably fast execution and high portability, while preserving the ability to read and understand the VM source code without leaving Smalltalk.

The PyPy project[4] is a toolchain for building interpreters[9]. It allows interpreters to be written in RPython, a restricted form of Python, and then translated to a more efficient lower-level language for execution. PyPy is able to translate RPython programs to many different backends, ranging from C source code, to JavaScript (for execution in the browser), to bytecodes for the JVM or CLR, although not all of these backends are as full-featured as the others. In addition to simple translation, PyPy can introduce optimizations along the way, and can even generate a just-in-time compiler semi-automatically from the interpreter source. These features are described in depth in other publications[1, 6].

At first glance, it may seem that the role of Slang in Squeak and RPython in Spy/PyPy are very similar. Both are restricted forms of a dynamic language used to code the core of the interpreter. However, the similarity is only skin deep. Slang, the restricted form of Smalltalk used by Squeak, is designed to be easily translated to C. Slang only contains constructs that can be directly mapped to C. RPython, on the other hand, is much closer to the full Python language and includes such features as garbage collection, classes with virtual functions, and exceptions. Having such facilities available frees the programmer to focus on the language design issues and ignore the mundane, low-level details of writing a VM.

The main contributions of this paper are

- We report on our experiences using the PyPy toolchain to realize Spy, a Smalltalk-80 VM.
- We present implementation details and discuss design decisions of the realized VM.
- We compare benchmarks of Spy with those of similar Smalltalk VMs.

The remainder of this paper is structured as follows: In Section 2 we present a brief overview of the PyPy project. In Section 3 we explain how the PyPy approach has been adapted in the Spy project to the implementation of a Squeak VM. Related work is presented in Section 4. Section 5 presents the results of various benchmarks to validate the effectiveness of the Spy implementation. We provide remarks on future work in Section 6 and eventually present in Section 7 our conclusions. Additionally, the source code of the benchmarks is given in Appendix A, and download and build instructions for both PyPy and Spy are given in Appendix B.

2 PyPy in a Nutshell

Although the initial goal of the PyPy project was to implement a next-generation interpreter for Python, the project has gradually evolved into a general-purpose

[4] http://codespeak.net/pypy/

tool that can be used for any number of languages. In addition to Python and Smalltalk, (partial) interpreters have been developed for Prolog, Scheme and JavaScript.

The goal of the PyPy project is to create an environment that makes it easy to experiment with different virtual machine designs, but without sacrificing efficiency. This is achieved by separating the semantics of the language being implemented, such as Python or Smalltalk, from low-level aspects of its implementation, such as memory management or the threading model. A complete interpreter is constructed at build time by weaving together the interpreter definition and each low-level aspect into a complete and efficient whole[9].

The project currently includes a wide variety of backends that support translations from RPython into C/Posix, LLVM[5], CLI/.NET, Java bytecodes, and JavaScript, although only the first three are fully functional.

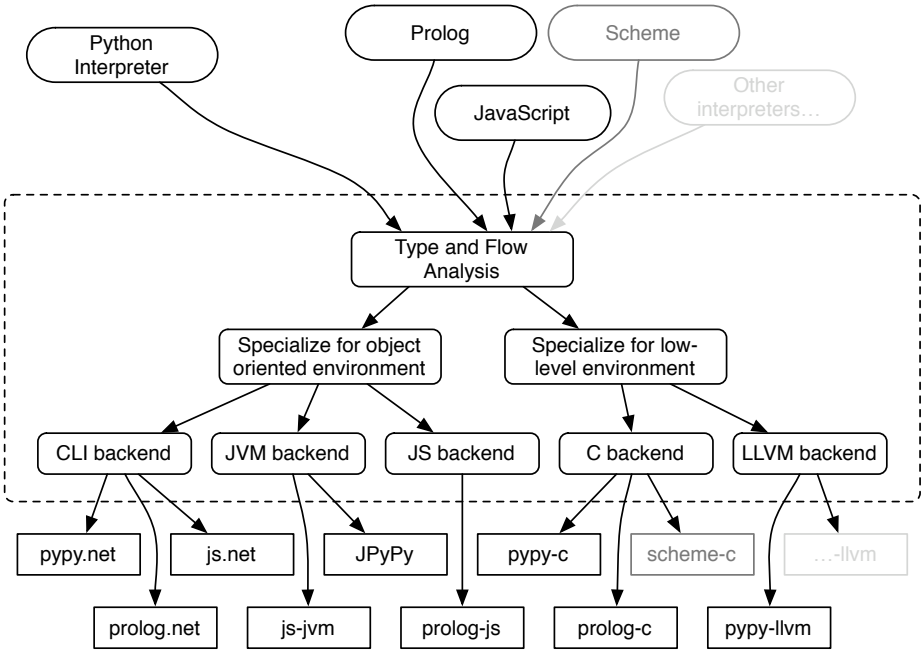

Fig. 1. The PyPy toolchain specializes high-level interpreters for different languages into compilers or VMs for different platforms.

The translation process works by using abstract interpretation to convert the RPython programs into flow graph form. The graphs are then used for whole-program type inference, which assigns a static type to all values used in the program. The ability to perform type inference on the input programs is the

key requirement for the PyPy toolchain. This means that RPython is defined, rather imprecisely, to be the subset of Python which our tools can statically check. In practice, RPython forbids runtime reflection and any type-inconsistent usage of variables (*e.g.,* assigning both integers and object references to the same variables). Despite these restrictions, RPython is still quite expressive, supporting exceptions, single inheritance with explicitly declared mixins (instead of Python's full multiple inheritance), dynamic dispatch, first class function and class values, and runtime `isinstance` and type checks.

Once the flow graphs have been built and typed, they can be transformed by any number of translation aspects[1] which implement low-level details, such as garbage collection or a variant of the CPS-transformation. These translation aspects give tremendous flexibility in controlling the behavior and performance of the final interpreter and also illustrate one of the advantages of specifying the interpreter in a higher-level language like RPython. Because RPython does not specify low-level details such as the garbage collection strategy, the toolchain is free to implement them however it sees fit. In contrast, changing the traditional, C-based Python interpreter so as not to use reference counting would require pervasive changes throughout the entire codebase.

The promise of PyPy and RPython is that it should be possible to develop a single interpreter source which can be used via different choices of translation aspects and backends, to create a whole family of interpreters on a wide variety of platforms, as illustrated by Figure 1. This avoids the problem that many languages face, *i.e.,* to keep the interpreter definition in sync across all platforms on which it is supported, and to allow all versions to benefit from new features and optimizations instantly. As an example, consider the Jython project[5], which defines a Python interpreter on the JVM. Because Jython and CPython do not share the same source, Jython lags several versions behind its C counterpart, making it increasingly challenging to use with modern Python programs. PyPy essentially offers a model-driven approach [10] to programming language implementation — it transforms platform-independent models (*i.e.,* high-level interpreters) into implementations for multiple platforms.

Another advantage of this approach is that since RPython is a proper subset of Python, an RPython program can be fully tested and easily debugged by letting it run on a Python interpreter. The program is only translated to a different language if it is reasonably bug-free. This increases testability, eases development, and decreases turnaround times.

3 Spy Implementation

Similar to most Smalltalk VMs, Spy consists of four main parts: a bytecode interpreter, a set of primitives, an image loader, and an object model.

[5] http://jython.org/

3.1 Interpreter, Primitives, and Image Loading

The core components of a Smalltalk VM are the bytecode interpreter, primitive methods, and the image loader. For the most part Spy does not deviate from the traditional Smalltalk VM design [3], though in some cases we made minor alterations. For example, Spy is not based on an object table, *i.e.,* objects reference each other directly without a level of indirection. This is similar to Squeak's approach as described in Section 4.

Bytecodes in Smalltalk are generally used to implement control flow and message sends, and to introduce constant values into the computation. Spy's bytecode interpreter takes a traditional form, consisting of a table of function pointers, which is indexed by the current bytecode on every iteration.

As a performance optimization, during the translation process to C, we are able to take advantage of the fact that the function table is immutable. This allows us to alter the dispatch loop so that it uses a local `switch` to translate bytecodes to method calls, rather than having an indirection via the global opcode table (see Figure 2). This will not only localize lookups but it will also use direct instead of indirect calls. Which will then allow for further optimizations such as inlining of the actual code related to the bytecodes.

```
Before:
table = [method_for_opcode_0, method_for_opcode_1, ...]
while 1:
    byte = get_next_byte()
    method = table[byte]
    method()

After:
while 1:
    byte = get_next_byte()
    switch on byte:
        case 0:
            method_for_opcode_0()
        case 1:
            method_for_opcode_1()
        ...
```

Fig. 2. Translation of the dispatch loop from a bytecode table to a local switch.

Compared to other virtual machines, Smalltalk contains relatively few bytecodes. For example, there are no bytecodes for low-level operations such as doing arithmetic. Instead, these operations are implemented as *primitive methods*, which are methods that are implemented in the core virtual machine, either for efficiency's sake or because they encode a fundamental operation which isn't possible to express in the language itself, such as integer addition.

Primitive methods are invoked as the result of normal message sends. When an object receives a message in Smalltalk, the first thing that happens is the lookup of the corresponding method implementation. The resulting method object contains the bytecodes to execute and, optionally, a primitive method identifier, which is just a small integer. If a primitive method identifier is supplied, the VM uses the integer to index into its primitive method table to find a built-in function to execute.

As in Squeak, primitive methods in Spy are implemented as a series of functions placed into a table. In Spy, however, we are able to take advantage of several RPython features to make their implementation less tedious and error-prone. The first feature are exceptions: in the Squeak VM, when a primitive method wants to signal failure, it does so by setting a field, `primitiveFailed`, of the global interpreter object to true. This means that all following code must be guarded to ensure that it does not execute once the `primitiveFailed` field is set to true. In RPython, however, we can use a Python exception to signal a failure condition, resulting in less cluttered code.

The second RPython feature, which proved to be very important is its capacity for meta-programming. Because primitive methods execute directly on the VM structures, they often contain a certain amount of repetitive code that loads method arguments from the stack, inspects their types, and finally pushes any result back onto the stack. Using Python annotations, however, we are able to attach a declarative tag to each primitive method stating the number of stack arguments it expects, any preprocessing they require, and whether or not a result is pushed back on the stack after execution. This not only makes the primitives shorter, it helps to avoid errors. In particular, we were able to use these annotations to specify when an argument represents an array index: since array indices are 1-based in Smalltalk, the preprocessor is not only able to confirm that the index is an integer, but can automatically subtract 1 to convert it to a 0-based RPython array index, leading to much cleaner code.

Figure 3 shows the definition of the primitive method for computing square roots. The `@expose_primitive` annotation on the first line declares both the primitive code, which is the symbolic constant `FLOAT_SQUARE_ROOT`, and the fact that the function expects only one argument from the stack, which should be a floating point value. Note that the object on the stack is actually a wrapped floating point value, but the preprocessor automatically inserts code to unwrap it and extract the RPython floating point value within. This unwrapped value is passed to the implementation function. Within the body of the function, there is a test that ensures that the argument is positive which raises an exception (`PrimitiveFailedError`) should that not be the case. Otherwise, the square root is computed using the standard RPython function `math.sqrt`, wrapped in a Smalltalk object, and returned. Note that the return value will be automatically pushed on the stack.

For comparison, Figure 4 shows the the same primitive method in Slang. The key difference to RPython is that Slang does not provide object-oriented language features. Slang is, roughly spoken, C code disguised as Smalltalk syntax.

```
@expose_primitive(FLOAT_SQUARE_ROOT, unwrap_spec=[float])
def func(interp, f):
    if f < 0.0:
        raise PrimitiveFailedError
    w_res = utility.wrap_float(math.sqrt(f))
    return w_res
```

Fig. 3. The definition of the primitive square root operation in RPython. The code uses high-level features, such as method decorators, exceptions, and object-orientation.

For example, to indicate failure, a global field of the interpreter is used rather than throwing an exception. Pushing and popping has to be done manually. But in particular the call to `#cCode:inSmalltalk:` breaks abstractions and testability: as a first argument it is given a fragment of C code, as a second argument a Smalltalk closure. When translating the VM down to C, the code fragment is literally copied into the generated source code. When debugging the VM within another Smalltalk image, the closure is evaluated. As both are not causally connected, it might even happen that a bug in the C code does not appear when debugging the VM and vice versa!

```
primitiveSquareRoot
    | rcvr |
    self var: #rcvr type: 'double '.
    rcvr := self popFloat.
    self success: rcvr >= 0.0.
    successFlag
        ifTrue: [self pushFloat:
            (self cCode: 'sqrt(rcvr)' inSmalltalk: [rcvr sqrt])]
        ifFalse: [self unPop: 1]
```

Fig. 4. The definition of the primitive square root operation in Slang. The code is, roughly spoken, C code disguised as Smalltalk syntax. Object-oriented features are not used, *e.g.,* failure is signalled with flag rather than an exception.

Image loading is one area where Spy differs significantly from Squeak. Traditionally, a Squeak image is simply a dump of the core memory into a file. Loading an image can be done by simply memory-mapping the image file, followed by some minimal pointer and integer adjustments. This technique works well when you can guarantee that the memory layout between virtual machines is compatible. Unfortunately, the memory layout for a RPython program is not specified and sometimes even outside of the control of PyPy's toolchain, if the translation target is a high-level VM such as the JVM or .NET's CLR. Since we wanted to retain compatibility with Squeak's image formats, we implemented an

image loader that proceeds by parsing the Squeak image file formats, decoding the object headers, and constructing equivalent objects in our own VM.

3.2 Object Model

The Squeak implementation uses three different addressing schemes for its objects: bytes, words, and pointers. Each object structure begins with a format word that describes which kind of object it is. This determines how the raw bytes that make up an object in memory are interpreted: a "bytes" object treats them as an array of bytes, which is useful for classes like strings. "Words" objects store words, and are useful for vectors of integers. Finally, "pointers" objects contain pointers, and are used for almost all other kinds of objects. In addition, as is common in many VMs, small integers are represented as tagged pointers.

The Spy model is somewhat more complex. In addition to bytes, words, and pointers objects, we have special classes for representing compiled methods, method and block contexts (stack frames), small integers, and floating point values. Please refer to Figure 5 for the full class hierarchy.

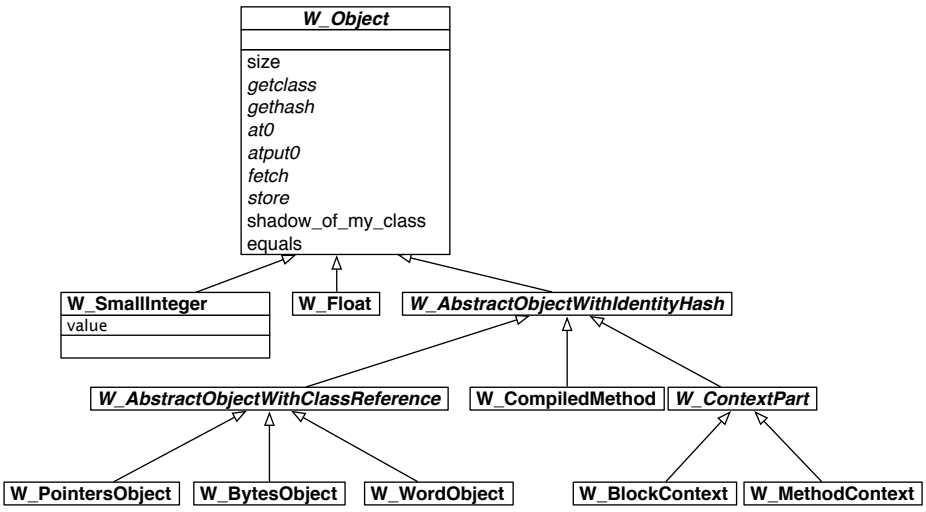

Fig. 5. Different kinds of objects in the Spy implementation.

All these classes are subclasses of an abstract class representing a Smalltalk object (`W_Object`). Therefore they all implement the same interface, which makes them equivalent from the Smalltalk point of view, and any of them can be used anyplace that a standard Smalltalk object is expected. The concrete implementation, however, differs from class to class. For the classes that are close to the VM

internals (such as compiled methods, method and block contexts), the implementation is made as convenient for the VM as possible, whereas the implementation of generic Smalltalk objects is less complex.

As illustrated by Figure 5, small integers are implemented by a normal class that has exactly one field, which contains the value of the integer itself. This deviates from the original design in the Squeak VM where they are represented by tagged pointers. To compare results of both styles, we could easily mimic the behavior of the Squeak VM by plugging an extra transformation into the toolchain. With the transformation turned on, the resulting C source generated by the toolchain would actually use tagged pointers as representation of small integers. This is by itself already another example where the RPython code abstracts over low-level details. We can assume a consistent model everywhere and do not need to check for tagged pointers throughout the source code, while resulting in the exact same behavior. A small bit of experimentation seemed to indicate that using tagged pointers for small integers actually worsens performance. The necessity of checking whether a pointer is a heap pointer or a small integer around every method call offsets all the benefits of the smaller memory footprint that comes with tagged small integers. It is important to note, however, how tedious it would be to experimentally introduce or remove tagged pointers with a traditional, low-level interpreter.

The class hierarchy illustrated by Figure 5 is internal to the VM, it is not related to Squeak's class hierarchy. All these classes are internally used for wrapper objects, hence the `W` prefix, and do not denote the high-level class of objects. Which high-level class an object has is completely under control of Squeak itself, it is stored in the `W_Object.shadow_of_my_class` field. Thus, the VM's class hierarchy can be used to run any version of Squeak. Both Squeak 2.0 and Squeak 3.9 images run with the current implementation of Spy.

3.3 Shadow Objects

As noted in the previous section, Squeak does not distinguish between objects based on the role that they play in the system, but only based on the kind of data that they contain (bytes, words, or objects). For example, a class object is simply an object that is *used as a class* by some other object. It is not necessarily an instance of a particular class, though the layout of the object in memory must be compatible with what the VM expects[6]. This implies that, at image loading time, it is impossible to distinguish which objects are, or will be later, used as classes, and so we cannot use a special subclass of `W_Object` to represent them.

Unfortunately, being forced to use a generic data layout for such special kinds of objects as classes can be very inefficient. The memory layout of Smalltalk data

[6] To be used as a class, a Squeak object must have at least three instance variables, of which the first must refer to its superclass, the second must refer to a method dictionary, and the third must contain a magic number encoding the format of instances. Any object that meets these criteria, and implements primitive #70 (primitiveNew), can be used to create instances of itself.

structures were chosen with an eye towards reducing memory consumption, and not for ease of access. Spy could be made far more efficient if it could use native RPython data structures instead. For example, each class has a method dictionary that is normally stored as a native Smalltalk `Dictionary` instance. If this method dictionary could be converted by the VM into a native RPython dictionary, then Spy could take advantage of the highly optimized RPython dictionary implementation.

To resolve this dilemma, Spy allows every Smalltalk object to have an associated "shadow" object. These shadow objects are not exposed to the Smalltalk world. They are used by the VM as internal representation and can hold arbitrary information about the actual object. If an object has a shadow object attached, the shadow is notified whenever the state of the actual object changes, to keep both views of the object synchronized. One way of looking at shadows is as a general cache mechanism. However, the approach is far more powerful, since arbitrary meta-level operations can be triggered when the update notifications are received[7].

In the current implementation, the shadows are used to attach nicely decoded information about classes to all objects which are used as classes. This allows any object to be used as a class, even ones that are not instances of the Smalltalk class `Class`. The shadow object is created and attached to the class the first time the VM sees the object being used as a class. It stores all required information about the class in a convenient, easily accessible data structure (as opposed to the obscure bit format used at the Smalltalk level). The class shadow contains the format and size of instances, an RPython-level dictionary containing the compiled methods, and the name of the class (if it has one). All of this information is kept in sync with the "real" Smalltalk object that the shadow is associated with.

At the moment[8] classes are the only objects that have shadows attached to them. It seems likely that we will change some of the objects that are now implemented with special RPython classes to use shadows as well later. For example, Squeak allows arbitrary objects to be used as method and block contexts, but the current Spy implementation does not. This could be resolved by using shadow objects to contain any extra information associated with objects that are used as a context.

3.4 State of the Implementation

The VM parts described in the previous subsections add up to the current implementation of Spy. This implementation is able to load Squeak images (tested

[7] Shadow objects are not related to the concept of mirrors. Mirrors are a mechanism to introduce reflection on demand. Shadow objects are an implementation artifact of our design allowing us to benefit from native RPython data structures. They have nothing to do with reflection per se, though they could be used for this purpose too.

[8] This paper refers to revision 49630 of Spy on codespeak's SVN repository, for more information please refer to the download instructions in Appendix B.

with the Squeak 2.0 mini image and more recent Squeak 3.9 images) and execute all bytecodes and a subset of primitives. The most important missing primitives are the graphical primitives. We do already support enough for the VM to be able to run the `tinyBenchmarks`. Furthermore we are still lacking support for threading and image saving.

4 Related work

In this section we present existing Smalltalk VMs, and discuss how their implementation relates to Spy. The VM of the Squeak dialect of Smalltalk follows closely the specification given in the Smalltalk-80 Blue Book[3]. The Blue Book specifies an object memory format, the bytecodes, the primitives, and the interpreter loop of a Smalltalk VM.

4.1 Squeak VM

The main difference of Squeak's VM [4] compared to the Smalltalk-80 specification of the Blue Book [3] is the object memory format. The object memory specified in the blue book is based on an object table. An object table introduces a level of indirection for object references. In contrast, Squeak implements object references as direct pointers, that is, an object reference is just the address of that object in memory. Today, this is the common approach taken by most virtual machines.

Squeak's object memory layout consists of a header for the class pointer, hash bits, GC flags, size *etc.* and a fixed number of fields. There are four kinds of object formats. Objects with named instance variables, indexed object fields, indexed word size or byte size fields. Everything, including interpreter-internal data such as execution contexts, processes, classes, and methods, is represented as a normal object on the heap. An exception is the case of small integers, which are represented as tagged pointers. Special objects that have to be known to the interpreter, for example the process scheduler, are stored in a global table.

The majority of the Squeak VM is implemented in a subset of Squeak Smalltalk, named Slang. The Slang source code is then translated to C code to compile and link with the low-level, platform-specific C code. Slang is a very restricted subset of Smalltalk which does not support classes, exception handling, or other object-oriented language features. Therefore, Slang does not provide a higher level of abstraction than C.

Nevertheless, using Slang has advantages compared to writing C code manually. First, the translator applies several optimizations such as generating C switch statements for the dispatch loop or inlining procedure calls. Second, since Slang is a Smalltalk subset it can be run within another Squeak image, which can be very useful for debugging. As Squeak allows for incremental compilation, the implementation of the VM can be changed while it is running. In this way, time consuming edit-transform-compile-run cycles can be avoided.

The approach taken by PyPy is similar to that of Squeak/Slang, as the VM implemented in RPython can also be run directly without transformation and compilation. However, the key difference is that RPython (restricted Python) is much less restrictive than Slang. RPython provides object-oriented language features such as objects, class-based inheritance, exceptions, and translation-time reflection and metaprogramming [9, 8]. As discussed throughout this paper, RPython's extended capabilities simplify the implementation of the VM in many ways, ranging from using code generation and annotations to avoid boilerplate code, to the automation of complex, far-reaching optimizations like tagged integers.

5 Evaluation

In this section we present a comparison of performance and codebase size of different Smalltalk VM implementations.

5.1 Performance Benchmarks

To analyze VM performance, we use the TinyBenchmarks suite which is part of the Squeak mini image[2]. The TinyBenchmarks tests bytecode interpretation and message send performance. We refer the reader to Appendix A for the complete source code of the benchmark suite. For the Smalltalk platforms that do not support direct loading of our reference image, we ported the source code manually. All the platforms successfully run the TinyBenchmarks and produced the following two figures:

Bytecodes per second. To compare the performance of a virtual machine, we need to know how fast the bytecodes are processed by the VM. This is the first number reported by TinyBenchmarks. The value is calculated from a bytecode-heavy implementation of the "Sieve of Eratosthenes". The result is calculated using the runtime performance of this algorithm and the number of executed bytecodes. The number of executed bytecodes is the number of bytecodes that Squeak executes when running the benchmark, which makes the number meaningful even on different implementations.

Sends per second. In Smalltalk everything happens by message sends, with the exception of some transparently inlined control structures. Therefore an efficient implementation of message sends is crucial. The second number reported by TinyBenchmarks is the message sends (method lookup and method invocation) per second. It is calculated from the the runtime performance of a send-heavy recursive calculation of Fibonacci numbers.

In Figure 6 we present the result of running the TinyBenchmarks on various VMs in relation to the original Squeak VM. The machine used was an Apple Mac

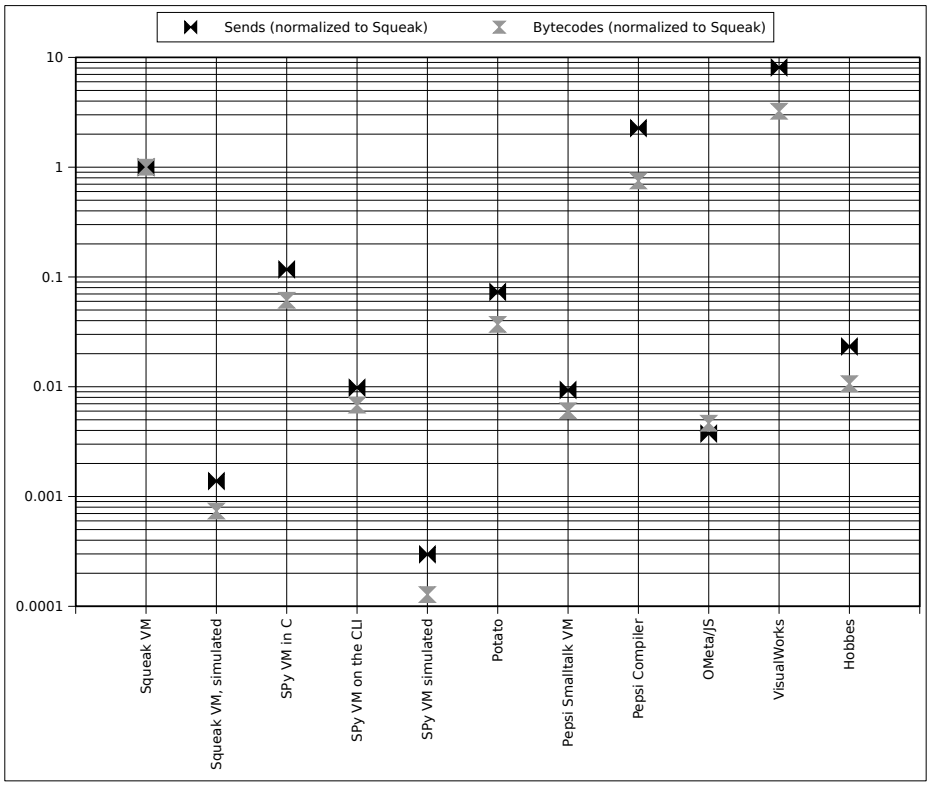

Fig. 6. Benchmark results for the TinyBenchmarks for various VMs, normalized to the Squeak VM

Pro (2 × 3 GHz Dual-Core Intel Xeon, 3 GB RAM). All the benchmarks were run 20 times; the final numbers are the arithmetic mean of those measurements.

Squeak VM. This is the original Squeak VM, written in Slang and transformed to C. It is heavily optimized and represents our point of comparison.

Squeak VM, simulated. The Slang code running in the Squeak VM interpreting the image is about 1000 times slower. The system is hardly usable like this, but it is a valuable means to debug the VM with the Smalltalk tools.

SPy VM in C. The result of our written VM after a week of intensive development is not at all bad. It runs at about a tenth the speed of the Squeak VM. This particular version was translated to C, using PyPy's generational GC and profile-guided optimizations.

SPy VM on the CLI. SPY translated to CLI (.NET Common Language Infrastructure) bytecode and running that on Mono is a significant factor slower than translating SPY to C. We assume that this is partially due to PyPy's CLI backend rendering some RPython constructs inefficiently.

SPy VM, simulated. This is SPY running untranslated on top of CPython[9] (the normal Python interpreter). Similar to running Slang code simulated on another Squeak, this is unusably slow but very useful for testing and debugging.

Potato. The VM running on Java is amazingly fast. Certainly this also has to do with the experience of the author with implementing other Smalltalk VMs.

Pepsi Smalltalk VM. The Squeak VM written with Pepsi is rather slow. The reason for this is that the VM is written in a highly dynamic Smalltalk-like language, which requires a repetition of lookups and message send per single bytecode in the interpreting VM.

Pepsi Compiler. Eliminating these lookups through the compilation of the code down to machine language, brings a huge performance boost. Currently this does not happen automatically through a JIT compiler, but it can be simulated by compiling the Smalltalk code of our benchmark using the Pepsi compiler. This removes the interpretation step from the code, but retains the fully dynamic object model.

OMeta/JS. We ran our OMeta/JS in the Safari Web Browser, as it has one of the fastest JavaScript engines available. We were amazed that it is in the same league as the CLI and Pepsi VM.

VisualWorks. VisualWorks is the fastest Smalltalk VM available today. It uses both sophisticated JIT compiler and memory management technology. The source code is not publicly available.

Hobbes. Running the benchmark reveals that Hobbes is around 100 times slower than the original Squeak VM. However, we have to point out that the Smalltalk-80 user-interface is responsive and comparable in speed to the machines of that time. A reason for that is certainly that Hobbes is running on VisualWorks.

[9] http://python.org

5.2 Lines of Code

To give a rough estimate of the comparative complexity of different VM implementations, we included Table 1 with a listing of approximate size of the respective codebase, measured in thousands of lines of code (KLOC).

Implementation	**Language**	**KLOC**
Squeak VM	Slang	8.9
Squeak VM (translation)	C	22.8
Spy VM	RPython	2.6
Spy VM (translation)	IL	130.4
Spy VM (translation)	C	187.7
Hobbes VM	Smalltalk	10.0
Potato VM	Java	4.7
Pepsi VM	Pepsi	10.9
Pepsi VM (translation)	C	2.1
OMeta	Javascript	1.4
VisualWorks	C	174.7
#Smalltalk	Smalltalk	7.0
Little Smalltalk	C	4.0
Little Smalltalk	Java	1.8

Table 1. Comparison of VM implementations in KLOC.

As shown in Table 1, Spy's RPython source is relatively compact. Spy's RPython source measures only 2600 lines of code, whereas the Slang source for Squeak requires 8900, and even the relatively compact Potato VM weighs in at 4700. This provides further evidence that the higher level of abstraction afforded by RPython is useful for keeping the implementation clean and uncluttered. As discussed in Section 3, we took advantage of a number of RPython features, including annotations, exceptions, and post-processing transformations, to simplify the Spy source and to improve performance. Without such features, Spy would be significantly more complex and more difficult to maintain. Please also note that although our implementation is fairly complete, there are still missing parts (see Section 3.4).

6 Future Work

The section discusses future work regarding Spy. Currently Spy lacks support for several primitives that are needed to make it a realistic replacement of the original Squeak VM. In particular these are primitives for UI and threading. With regard to Squeak's plugin mechanism, we aim to find a way to reuse its interfacing with external functions so we can avoid redoing the work to interface

with third-party libraries. When this is done Spy should be a slow but usable replacement for the original Squeak.

Afterwards, we can concentrate on speed optimizations. We plan to implement some straightforward optimizations in the VM, the most obvious example being a method cache. The shadow approach described in Section 3 should make this straightforward, since the shadows of classes are already kept up-to-date automatically and are thus an obvious place to put a method cache. This should get rid of the most obvious inefficiencies in the current VM.

An area that the PyPy project is currently researching is the automatic generation of just-in-time compilers from interpreters using partial evaluation techniques. The language implementor needs to guide this process with a small number of hints in the interpreter source code. This already works well for PyPy's Python interpreter, where speedups of up to 200 times over normal interpretation can be achieved for simple integer arithmetic [6]. We hope to be able to apply these techniques to Spy as well to get a high-speed VM implementation that could eventually surpass Squeak's performance. This would allow us to get a just-in-time compiler with very little effort, while retaining our easy-to-understand interpreter source code.

While the Spy project specifically tries to use PyPy's toolchain to implement a Squeak VM, it would be worthwhile future project to try to apply some of the ideas of the PyPy project to a pure Squeak setting. This would mean implementing a translation toolchain for a subset of Smalltalk that is higher-level than Slang and then building a VM in it. Doing that would allow it to evaluate PyPy's concepts and to explore the design space for this sort of approach.

7 Conclusion

We have described the implementation details of the Spy project, and provided benchmark results which we believe demonstrate the potential of the Spy project: despite the lack of fundamental optimizations such as a method cache, and the fact that it was coded in a high-level language (complete with garbage collection and other modern amenities), Spy delivers performance competitive with or better than other alternative Squeak implementations.

Spy was developed partly as an experiment to see how suitable the PyPy toolchain would be for a Smalltalk implementation. We found that PyPy is indeed a very useful tool for quickly implementing a virtual machine. The fact that Spy was developed in only one week of development attests to the productivity boost offered by PyPy. By using a high-level language like RPython, and in particular one with support for metaprogramming, we were able to reduce errors and eliminate boilerplate code throughout the system. Furthermore, PyPy's support for translation aspects enabled us to experiment with systematic, low-level optimizations, such as tagged integers, easily and without changes to the interpreter source.

We are confident that with further development, Spy could join Squeak as a realistic platform for Smalltalk development.

Acknowledgments

We thank Orla Greevy for her careful review of a draft of this paper. The members of the Software Composition Group, University of Bern, gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008), and the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance". Adrian Kuhn acknowledges the support of CHOOSE, the Swiss Group for Object-Oriented Systems and Environments.

References

1. Carl Friedrich Bolz and Armin Rigo. Memory management and threading models as translation aspects – solutions and challenges. Technical report, PyPy Consortium, 2005. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
2. Marcus Denker. Entwurf von optimierungen für squeak, 2002. Studienarbeit, Universität Karlsruhe.
3. Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.
4. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.
5. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.
6. Samuele Pedroni and Armin Rigo. JIT compiler architecture. Technical report, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
7. Ian Piumarta and Alessandro Warth. Open reusable object models. Technical report, Viewpoints Research Institute, 2006. VPRI Research Note RN-2006-003-a.
8. Armin Rigo, Michael Hudson, and Samuele Pedroni. Compiling dynamic language implementations. Technical report, PyPy Consortium, 2005. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.
9. Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium, OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.
10. Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.
11. Alessandro Warth and Ian Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.

A Source code of TinyBenchmarks

```
Number>>tinyBenchmarks
    | t1 t2 r n1 n2 |
    n1 := 1.
    [ t1 := Time millisecondsToRun: [ n1 benchmarkPrimes ].
      t1 < 1000 ] whileTrue: [ n1 := n1 * 2 ].

    n2 := 28.
    [ t2 := Time millisecondsToRun: [ r := 28 benchFibonacci ].
      t2 < 1000 ] whileTrue: [ n2 := n2 + 1 ].

    ^ ((n1 * 500000 * 1000) // t1) printString , ' bytecodes/sec; ' ,
      ((r * 1000) // t2) printString , ' sends/sec'

Number>>benchmarkPrimes
    | size flags prime k count |
    size := 8190.
    1 to: self do: [ :iter |
        count := 0.
        flags := (Array new: size) atAllPut: true.
        1 to: size do: [ :i |
            (flags at: i) ifTrue: [
                prime := i + 1.
                k := i + prime.
                [ k <= size ] whileTrue: [
                    flags at: k put: false.
                    k := k + prime ].
                count := count + 1 ] ] ].
    ^ count

Number>>benchFibonacci
    ^ self < 2
        ifTrue: [ 1 ]
        ifFalse: [
            (self - 1) benchmarkFibonacci
                + (self - 2) benchmarkFibonacci + 1 ]
```

B How to download and run the Spy project

Make sure you are running Python version 2.5 or higher, and checkout the project from subversion

```
> svn co http://codespeak.net/svn/pypy/dist pypy-dist
> cd pypy-dist
```

Now, let's generate some Squeak VMs. Switch to the translation goal folder and run the toolchain

```
> cd pypy/translator/goal
> ./translate.py --gc=generation --batch targettinybenchsmalltalk.py
```

To run the generated executable:

```
> ./targettinybenchsmalltalk-c
```

If you browse the target's Python file, you'll find some fixture code together with a function called `entry_point(argv)`. The fixture code is executed before the toolchain takes over. It may use the full power of Python and is not restricted to RPython. Then, the toolchain is started up, taking the `entry_point` function and the fixture's result as an input, to generate the VM. Therefore, all code eventually called by the entry point must conform to RPython.