When a Mouse Eats a Python

Smalltalk-style Runtime Development for Python and Ruby

Tim Felgentreff Fabio Niephaus Tobias Pape Robert Hirschfeld

Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

1. Debug Mode is the Only Mode

Debuggers in interactive programming environments are powerful tools to explore and develop systems at runtime.

However, among users of scripting languages such as Python or Ruby, a debugger is sometimes viewed as a rarely used "development time" tool. As Seaton, Van De Vanter, and Haupt have observed [5], debugging support is assumed to come with compromises: there surely must be a *performance* impact; in order to minimize the impact when debugging is not needed, the *functionality* surely must be limited; the *complexity* of debugging couples debuggers closely to just one language; and in order to actually use debugging facilities, one surely must accept the *inconvenience* of having to run the program in a special "debug" mode.

Due to the inconvenience involved in using debuggers in such systems, developers instead set up feedback loops by creating the infrastructure for quick, repeated test executions. This further cements the distinction between "development time" and "deployment time": to a program running in production, this feedback loop infrastructure is not available and any issues have to be reproduced and distilled into tests on a development system before they can be fixed.

As Gilad Bracha has noted,[1] this separation of development time and deployment time stands in contrast to environments in the Lisp, Smalltalk, and Self heritage that view programs as live, continuously evolving systems. The development environment *is* the runtime environment and developers can work with concrete objects and can interrupt, inspect, and modify runtime state, and keep running.

In this work, we present a prototype virtual machine (VM) written in RPython [1] based on the RSqueak/VM that provides Squeak/Smalltalk's live development and debugging for PyPy Python [4] and Topaz Ruby.[2] Of particular interest in this context is how the interpreters can be adapted in a general fashion for Smalltalk-style development, as well as the practical overhead of such an integration.

The core features of our prototype are:

- A combination of multiple RPython interpreters in the same, cooperatively scheduled execution environment,
- Smalltalk-style unhandled exception and edit-and-continue debugging for Ruby and Python.

2. Implementation Sketch

Our implementation is based on the approach to interpreter composition in RPython proposed by Barrett, Bolz, and Tratt [2]. Since we use RPython as our implementation language, we can use its meta-programming and object-oriented model to generically adapt the different interpreters.

Our main RPython interpreter is RSqueak that implements Squeak/Smalltalk, because we use its graphical system as our development environment. Both, Ruby and Python, can be accessed from within Squeak/Smalltalk with a primitive through which Squeak/Smalltalk code can call into the other languages. In addition, objects from different languages can pass across language borders: they simply become opaque *foreign objects* for which method lookup and execution is routed through the corresponding language's interpreter.

However, this combination offers little more than a convenient foreign-function interface — the Smalltalk development model relies on the ability to interact with the environment while the system is running. Smalltalk has traditionally implemented this by using green threads (called *Processes* [3]) with a "UI process" running at high priority to check for user input. Furthermore, all processes and the scheduler are accessible from Smalltalk, so that developers can inspect, interrupt, and modify execution at runtime.

2.1 Yielding to the UI Process

In order to have the environment react to user actions, we integrate the cooperative scheduling mechanism of Smalltalk processes with all interpreter loops: using RPython meta-programming, we patch the Ruby and Python interpreters at convenient locations to yield execution to the Smalltalk scheduler. This happens regularly at operations like bytecode

[1] `https://gbracha.blogspot.com/2012/11/debug-mode-is-only-mode.html`

[2] `http://docs.topazruby.com`

MoreVMs'17 April 3, 2017, Brussels, Belgium

executions, message sends, and/or backwards jumps. The Smalltalk interpreter already yields in the same way after a certain number of backward jumps and message sends.

To make Python and Ruby executions available to the Smalltalk tools for inspection and to the Smalltalk scheduler for resumption, we create an entry point by putting an artificial suspended Squeak frame on the top of the stack. The Smalltalk tools can query this frame for its caller/sender — retrieving a Python or Ruby frame as Squeak/Smalltalk object — or resume it. When this virtual frame continues execution, it returns control to the Python or Ruby interpreter by using the *Stacklet* mechanism for delimited continuations available in RPython.

2.2 Exception Detection

Another requirement of an "always running" system such as Squeak/Smalltalk is to detect unhandled exceptions, stop the execution, and offer to debug them. This is in contrast to the usual approach in Python and Ruby where an unhandled exception can unwind the stack and exit: but if your deployment environment is also your development environment, it must never exit due to an exception.

We detect unhandled exceptions by intercepting the raising of language exception objects in each interpreter on the RPython-level. When an exception is raised, we search the stack for an exception handler. If it is found, we continue raising it, otherwise we interrupt execution immediately — leaving the stack untouched — and return control back to the Squeak/Smalltalk UI process, which can then open a debugger to inspect the erroneous execution.

2.3 Restarting Frames

Finally, Squeak/Smalltalk allows changing and restarting execution frames, which is indispensable for edit-and-continue debugging. Since frames in each interpreter are typically represented as objects in RPython, we can restart them by setting their stack, program counter, bytecode, or any other runtime state to desired values. To use this from Squeak/Smalltalk tools, we expose these frame objects and add a primitive to restart their execution.

3. Performance Considerations

Combining the interpreters of different RPython VMs already results in nicely inlined JIT loops on toolchain level. However, there are three sources of potential overhead in our prototype: the addition of counters for yielding to the UI, the addition of stack-walking to find unhandled exceptions, and a limitation of RPython not being able to apply an optimization that avoids allocation of more than one type of stack frame object.

We minimize the cost of counters by not checking them after each bytecode when we are in machine code, but instead count once at the end of each loop the number of bytecodes executed and possibly yield then. This means we might yield later than intended, but if the counter is chosen to yield roughly every 5 ms, we can easily miss the target by a few factors and still have a responsive UI.

The cost of finding exception handlers is proportional to the distance of the handler from the exception. For most exceptions that occur and are caught (such as stopping an iteration in Python), this distance is small and often within the same JIT trace and, thus, the overhead is often negligible. Exceptions that are uncaught will stop execution and yield anyway, so performance is not as relevant in this case.

To avoid allocations of stack frames, RPython offers a JIT hint called *virtualizables*. When method frames are inlined under normal circumstances, the frame objects do not escape and the JIT can avoid allocation by normal escape analysis. The outermost frame in a JIT trace exists before the beginning and after the end of a loop, however, so its allocation cannot be avoided. We have found in benchmarks that the overhead is usually small if the actual work is instead done inside another method called inside a loop body.

4. Conclusions and Future Work

We were able to enable Smalltalk-style runtime development for Python and Ruby with minimal effort, minimal loss in performance, and in a generic manner that we believe can be applied to other RPython languages as well. As part of future work, we want to investigate Python and an RPython implementation of Prolog in particular to explore if Squeak/Smalltalk's live development can also aid in the creation of better debuggers and other tools for these languages compared with what is currently available.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8. doi: 10.1145/1297081.1297091.

[2] E. Barrett, C. F. Bolz, and L. Tratt. Approaches to interpreter composition. *Computer Languages, Systems & Structures*, 44, Part C:199–217, 2015. ISSN 1477-8424. doi: 10.1016/j.cl.2015.03.001.

[3] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman, Boston, MA, USA, 1983. ISBN 978-0201113716. The Blue Book.

[4] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA'06 Companion*, pages 944–953, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176753.

[5] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 2:1–2:13, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2916-3. doi: 10.1145/2617548.2617550.