Efficient Implementation of Smalltalk Activation Records in Language Implementation Frameworks

Fabio Niephaus
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Tim Felgentreff
Oracle Labs
Potsdam, Germany
tim.felgentreff@oracle.com

Tobias Pape
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
tobias.pape@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Language implementation frameworks such as RPython or Truffle help to build runtimes for dynamic languages. For this, they make certain design decisions and trade-offs upfront to make common language concepts easy to implement. Because of this, however, some language-specific concepts may be rather tedious to support, especially the modification of activation records. For example, Smalltalk provides reification of activations through *context* objects. Since they are used to implement other mechanisms such as exception handling on the language-level, contexts need to be entirely supported by the underlying runtime. We present an approach for efficiently implementing Smalltalk context objects in frameworks that do not support unrestricted modification of activation records.

CCS CONCEPTS

• **Software and its engineering → Interpreters**; *Runtime environments*; Just-in-time compilers.

ACM Reference Format:
Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2019. Efficient Implementation of Smalltalk Activation Records in Language Implementation Frameworks. In *Proceedings of <Programming> 2019 (MoreVMs'19).* ACM, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Virtual machines (VMs) for dynamic languages commonly manage activation records in the form of a stack of call frames. Depending on the implemented language, a VM may have to expose some of the internal data structures it uses for this to the language. Language implementation frameworks such as Truffle [13] or RPython [1] provide facilities to manage call frames in an efficient way. Neither framework, however, supports unrestricted modification of activation records (such as returning to a different sender) which are required to implement Smalltalk-80 [5]. We present an approach for implementing full support for Squeak/Smalltalk context objects in both Truffle and RPython.

MoreVMs'19, April 2, 2019, Genova, Italy
2019. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

2 BACKGROUND

Smalltalk exposes activation records as first-class objects to the language. Unlike in many other languages, these *context* objects are arbitrarily modifiable and can therefore be used as continuations [12]. Control flow can easily be manipulated by changing a context's instruction pointer or its sender context. This allows, for example, a language-level implementation of exception handling and other language mechanisms.

In Truffle, a dedicated frame implementation has to be used to represent an activation record of the guest language while in RPython, language implementers have to designate one interpreter-level class to allow it to be stack-allocated. In the remainder of this paper, we will use the term "frame" for an activation record in both frameworks, and "context" in Squeak/Smalltalk. Moreover, the two frameworks provide a Just-in-time compiler (JIT) to improve language performance. A common optimization applied by these JITs is to avoid the allocation of frames on the heap by reusing the machine stack. For this reason, Truffle differentiates between *virtual* and *materialized* frames, which are allocated on the machine stack and heap, respectively. Furthermore, Truffle allows language implementers to request the materialization of a virtual frame, consequently forcing it to heap. In RPython, on the other hand, fields of the frame class must be marked as *virtualizable* and special references ("virtualrefs") must be used to chain them. Only then the framework's JIT may decide to allocate these fields on the stack.

3 APPROACH

For an efficient Smalltalk implementation, it is worthwhile to optimize the representation of Squeak/Smalltalk context objects [2]. In frameworks like Truffle and RPython, we suggest to use two different representations: A *stack-allocated* context is represented by a frame on the machine stack. In contrast, a *heap-allocated* context is used when all fields of the context need to be allocated on the heap, for example when they are needed outside of the current scope. As an effect of the framework's JIT, a context may actually be represented in a hybrid manner in case escape analysis is able to allocated only some fields of the corresponding frame on the stack.

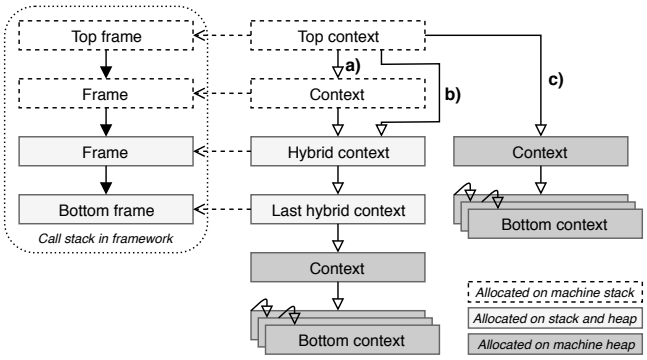

Figure 1: The framework-level frames can be in sync with contexts (a), or out of sync after a sender has been changed to a context of the current stack (b) or a different stack (c).

Furthermore, a complete implementation also needs to support sender modifications. Both frameworks, however, try to aggressively allocate frames on the stack for performance reasons. For this, they assume that senders are immutable which is a valid assumption for most languages, but not for Smalltalk.

In the prevalent case that activation records are not modified, each context is ideally represented by a corresponding stack-allocated frame object in the language framework (cf. Figure 1a).

The sender of a context can change, which happens frequently for example as part of Squeak/Smalltalk's exception handling. In this case, the chain of contexts and the stack-allocated frames may get out of sync (cf. Figure 1b). The sender of the current context may be changed to a different parent context which is further away from the original sender in the chain. The two stacks can be brought back in sync by unwinding all stack-allocated frames until the frame representing this parent context is reached.

Moreover, the new sender of a context can also be a new materialized context from a different stack of contexts (cf. Figure 1c). Then, all frames the framework manages need to be unwound and the return value needs to be passed to the new context.

4 IMPLEMENTATION

An implementation of our approach has proven to work well in RSqueak/VM [3], a Squeak/Smalltalk VM written in RPython. Contexts are represented by ContextPartShadow which has virtualizable attributes and its sender field references another ContextPartShadow through a virtualref. To keep Squeak/Smalltalk contexts in sync with their shadows, we use exceptions to signal both non-virtual returns and process switches in RPython.

Similarly, we have implemented our approach in GraalSqueak [10], a Truffle-based Squeak/Smalltalk implementation for the GraalVM. Since Truffle's VirtualFrames must not be referenced by other objects, we identify them using thin marker objects that are stored in the first frame slot. If access to a context object is requested, GraalSqueak allocates and pushes a ContextObject which references a materialized Truffle frame. Further optimization is then left to the Graal compiler. To find a specific sender, Truffle's iterateFrames API is used to determine the corresponding frame. Again, we use exceptions to keep ContextObjects and Truffle frames in sync. Additionally, ContextObjects are fully materialized if they are marked as escaped, that is when they have been stored in some other object or when a process switch occurs.

5 DISCUSSION

Following our approach, we were able to implement support for Squeak/Smalltalk context objects in RSqueak/VM using RPython and GraalSqueak using Truffle. When contexts are allocated on the machine stack, the user interface is refreshed at normal frame rates (around 40 to 50 fps, capped by the programming system) while otherwise the frame rate is around one or two fps.

Moreover, our approach enables another key feature with regard to contexts: the chain of contexts can grow, at least in theory, infinitely. Since the two frameworks are based on Python and Java, it is possible that the framework-level stack overflows. In both VMs, we use the materialization mechanism to unwind all frames when this happens, so that context chains are supported that exceed the framework's stack size.

Implementing our approach in RSqueak/VM required fewer optimizations and was relatively straightforward compared with Truffle, which matches previous observations [7]. In case of sender modifications, for example, we had to avoid materialization of contexts as much as possible in GraalSqueak. Instead, we had to carefully use Truffle's iterateFrames frame-walking facility which triggers deoptimizations in the JIT under poorly documented conditions. This made parts of our approach cumbersome to implement.

6 RELATED WORK

The VM used for VisualWorks 5i [8] also uses different context representations and applied further optimizations to reduce runtime overhead. The OpenSmalltalk VM is the state-of-the-art VM for Squeak/Smalltalk, is still under active development, and uses "married contexts" [9], which are also used in SqueakMaxine [11]. The original "interpreter" VM [6], SqueakJS [4], and Potato[1] for Squeak/Smalltalk always allocate contexts on the heap.

None of these VMs are written in and constrained by a language implementation framework. The OpenSmalltalk VM which optimizes context objects is significantly more complex (300k+ SLOC) compared with RSqueak/VM (≈22k SLOC) and GraalSqueak (≈36k SLOC), which in turn are more complex than SqueakJS (≈8k SLOC) and Potato (≈6k SLOC). These two VMs and the interpreter VM, however, are significantly slower in performance compared with our VMs.

7 CONCLUSION AND FUTURE WORK

We presented an approach for implementing full support for Smalltalk context objects in Truffle and RPython, both of which do not support unrestricted alterations of activation records out of the box. Our implementation strategy played an important role in making the Smalltalk programming system usable. In the future, it would be interesting to see how our approach can be incorporated into language implementation frameworks as well as to measure performance implications in more detail.

[1] https://sourceforge.net/projects/potatovm/

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of Oracle Labs[2], HPI's Research School[3], and the Hasso Plattner Design Thinking Research Program[4].

REFERENCES

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/1297081.1297091

[2] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[3] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. 2016. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies (IWST '16)*. ACM, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/2991041.2991062

[4] Bert Freudenberg, Dan H.H. Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2014. SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, New York, NY, USA, 57–66. https://doi.org/10.1145/2661088.2661100

[5] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman, Boston, MA, USA. The Blue Book.

[6] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. 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)*. ACM, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[7] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 821–839. https://doi.org/10.1145/2814270.2814275

[8] Eliot Miranda. 1999. *Context Management in VisualWorks 5i*. Technical Report. ParcPlace Division, CINCOM, Inc.

[9] Eliot Miranda. 2011. An Arranged Marriage. http://www.mirandabanda.org/cogblog/2011/03/04/an-arranged-marriage/

[10] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '18)*. ACM, New York, NY, USA, 30–35. https://doi.org/10.1145/3242947.3242948

[11] Tobias Pape, Arian Treffer, Robert Hirschfeld, and Michael Haupt. 2013. *Extending a Java Virtual Machine to Dynamic Object-oriented Languages*.

[12] John C. Reynolds. 1993. The Discoveries of Continuations. *Lisp Symb. Comput.* 6, 3-4 (11 1993), 233–248. https://doi.org/10.1007/BF01019459

[13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[2] https://labs.oracle.com/

[3] https://hpi.de/en/research/research-school.html

[4] https://hpi.de/en/dtrp/