Powerlang: a Vehicle for Lively Implementing Programming Languages

Javier Pimás
Aucerna
Buenos Aires, Argentina
jpimas@aucerna.com

Guido Chari
Czech Technical University
Prague, Czechia
gchari@dc.uba.ar

Abstract

Developing a programming language from scratch, with a stable and yet efficient runtime environment could be considered a titanic task. To mitigate this situation, and promote the early emergence of new languages, two main alternatives have been proposed: adopt an established virtual machine or exploit meta-compilation frameworks. In addition, micro VMs have been proposed as a foundation for developing VMs on top of a thin layer of abstraction similar to the micro kernel concept proposed for operating systems. Each of these approaches represents a compromise solving the tensions between flexibility, correctness, efficiency, and effort when designing a language runtime. Accordingly, each approach exposes its benefits while still suffers from limitations.

The Bee development team has been developing a complete Smalltalk system for more than a decade. During the journey, we implemented several artifacts needed to build a virtual machine: parsers, compilers, assemblers, bootstrapping utilities, (native code) debuggers, remote execution protocols, simulation tools, and garbage collection algorithms, among others. After weighting all the different paths followed during these years, in this position paper we advocate for a novel approach for building language runtimes. Concretely, we envision Powerlang, a framework that would enable to design, debug, inspect, compile, optimize, and test new programming languages, with a moderate effort and significant versatility. Powerlang represents a novel point in the design space of this kind of solutions. Finally, Powerlang is developed using a live environment like Smalltalk. We also elaborate on why this is a excellent match to host such a framework.

Keywords programming language implementation, virtual machines, live environments

1 Introduction

Virtual machines are a widespread approach for implementing programming languages. Beyond executing language statements, VMs usually provide desired features such as (adaptive) optimizations, security enforcements, portability, and automatic memory management, among many others. Each of these functionalities is usually hard to implement in isolation, and the complexity is amplified in VMs as a consequence of their interleavings. Accordingly, building full-fledge VMs is usually a matter of expert teams of system programmers. It still usually takes them several years to reach a stable and practical artifact.

Unsurprisingly, whenever a new programming language emerges, its runtime development becomes a stone in the shoe. There exists three main alternatives to alleviate this situation, when considering VMs. The option that demands less effort is to reuse an established VM such as Java HotSpot [3]. Scala [4] and Kotlin are two mainstream languages that successfully compile to Java bytecodes. However, compiling to the JVM means you must use its garbage collector, threading model, etc. In addition, a semantic mismatch has already been observed when compiling arbitrary languages to a bytecode that was originally conceived to implement one particular family of language [2, 6].

On a more versatile side of the spectrum Wang et al. [7] proposes micro VMs, an abstraction over the low-level substrate most VMs should workaround: hardware, memory, and concurrency. Micro VMs are, thus, a thin layer providing minimal key services for code execution, memory management, and threading. On one hand, pursuing minimality makes the approach flexible enough to be suitable for a wide range of new languages. On the other hand, language implementers still need to cope with many low-level tasks such as implementing optimizations or a concrete garbage collector. It still remains to be proved whether using micro VMs it would be possible to build a rich variety of modules that could be reused by other language runtimes.

In between those approaches appears the Graal VM [8] and PyPy [1]. These meta-compilation frameworks request to be fed with an interpreter describing the guest language semantics. As a result, they generate a runtime including an efficient JIT compiler honoring the language semantics and a garbage collector. They have been proven useful for several different programming languages such as, Ruby, Smalltalk, R, and Racket, among many others. However, meta compilation approaches still rely on a concrete host VM. Thus, to implement features that the VM does not support the alternatives are to build abstractions that usually sacrifice performance, or resign support for those features. In addition, to reach practical performance, the approaches usually suffer from long warming-up times.

IWST'19, August 27–29, 2019, Cologne, Germany
2019.

Approach	Base Perf	Flexibility	Ease/Liveness
(a) Other VMs	★★★★	★	★★
(b) Meta VMs	★★★★★	★★★	★★★
(c) Micro VMs	★★★	★★★★	★★
(d) Scratch	★	★★★★★	★
(e) Powerlang	★★★	★★★★	★★★★★

Figure 1. Properties of the different approaches.

Finally, to reach practical peak-performance, developers must become experts on the respective DSL they provide, which is usually not trivial.

Based on the expierence we gathered developing Bee [5], a dynamic meta-circular runtime for Smalltalk, we propose to explore a novel point on the design space of VM development frameworks. In contrast to other approaches, we highlight the relevancy of interactive and extensive tooling for building, maintaining, and evolving programming language runtimes. We dubbed the resulting idea Powerlang. It consists of a top-down approach, abstracting most of the concrete tools and algorithms we developed during our long journey, to make them applicable to build arbitrary languages.

Figure 1 illustrates Powerlang's goal by representing different characteristics compared to other language implementation approaches. Targeting established VM bytecodes (a) provides a good performance, but imposes constraints and a lack of live tools. Metacompilation frameworks (b) increase flexibility and provide interesting performance with few development efforts. Micro VMs (c) and "from scratch" implementations (d) are even more flexible, but require much more involvement from implementers. Powerlang (e) aims to offer as much flexibility as the Micro VMs provide, while still reducing the load and the complexity put on the programmer's back as much as possible.

In the remaining of the paper we elaborate on the main ideas behind Powerlang, describe a subset of the modules we propose to include in a first iteration, and we develop on why we posit that Smalltalk is a perfect match for implementing a flexible language implementation framework.

2 Proposal

In the light of the current state of language implementation frameworks, we propose an alternative idea we conceive as Powerlang. [1] Our approach focuses on a live and interactive experience for language developers, and thus, we focus our attention on tooling, and advocate for Smalltalk as the host language. Below we describe a set of concepts (and tools) that, in our vision, will be the cornerstones of Powerlang: bootstrapping, debugging, and optimizing.

[1] An early first iteration of Powerlang can be found in https://github.com/melkyades/powerlang

2.1 Bootstrapping

Bootstrapping is the procedure that allows to execute a language from the first time until it is self sustainable. For example, a Smalltalk system needs a heap of objects with a concrete set of classes and methods in order to be able to run. The construction of such a heap can be done from a set of definitions. After the bootstrap has been done and the language is already self-sustainable, it becomes possible to take an existing heap and clone it to generate new versions of the *image*. But a principled approach to create new heaps from scratch simplifies different tasks that language implementers may face during a language lifetime:

- Modifications to the layout of entities in memory.
- Bit-by-bit image reproducibility.
- Deploying minimal language versions.

We claim that a programming language implementation environment should not only ease the creation and inspection of a base heap but also aid the bootstrapped language to load one or many of them into memory. But, bootstrapping a heap is prone to low-level bugs hard to track. Therefore, a language implementer should be equipped with tools to inspect the output and verify its correctness. Figure 2 shows an early prototype of such inspector in Powerlang.

2.2 Compiling, Simulating and Debugging

We believe that designing a complete language can be best accomplished as an iterative process. Language designers should be able to test features incrementally, being able to compile and even simulate execution after designing each building block. Even if the language is not yet complete, nor fully functional. Debugging of the statements of the language should be possible since the initial stages. This means that debuggers should be able to adapt to different execution mechanisms: at early stages the programs could be simulated or interpreted, and later be run on JIT –or AOT– based runtimes. The debugger should be able to switch between a transparent high-level view of execution and a low-level view of the machine code.

In our initial prototype, we designed an intermediate representation based on Smalltalk expressions, analogous to Lisp S-expressions[2]. It consists of a tree of expressions similar to a low-level abstract syntax tree, apt to be traversed for simulation and JIT-compilation. We expect this design to be flexible enough for implementing a wide range of languages.

A debugger for a language developed using Powerlang could look like the one shown in Figure 3. In the example, a Smalltalk snippet is being debugged, accompanied by a matching section of assembly, which could be used to debug a JIT compiler. A similar variation could be used to debug a simulated execution, without native code, or even without a completely working language, just a reduced set of methods. The elements required for implementing such debugger

[2] Except for the lack of homoiconicity

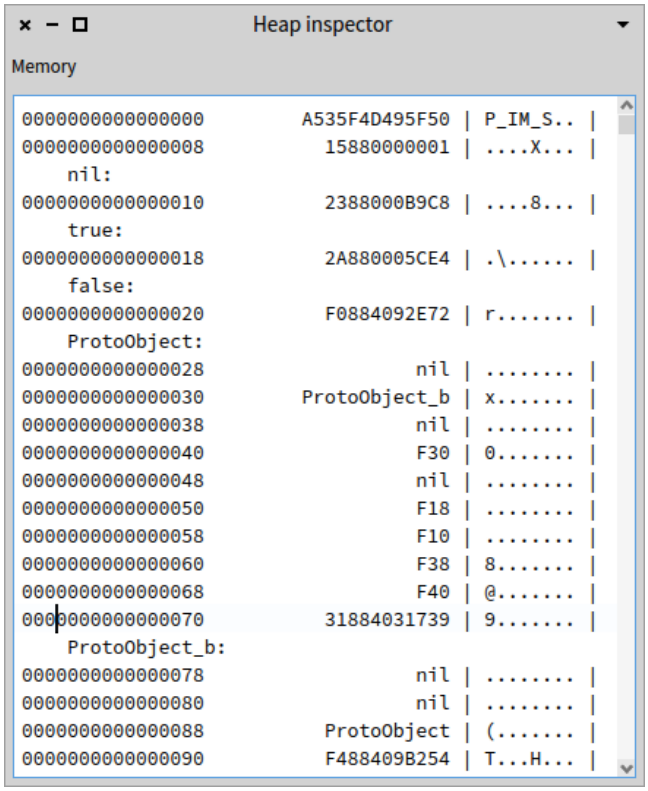

Figure 2. An inspector showing the contents of a bootstrapped heap, including the offsets and labels of objects. Other views could be added to improve visualization and to allow interactivity.

could be obtained directly from the framework, as it would integrate all components: compiler, native-code generator and debugger or simulator.

2.3 Interaction with Foreign World

Communication between the language and the outside world (through foreign-function and application binary interfaces) is, usually, a source of complexity for language implementers. We consider that an implementation environment must aid language designers in this task by providing reusable components that simplify the generation of such interfaces. For example the compiler provided by Powerlang should have an API for calling external functions, which the language designers could directly use instead of implementing their own. On one hand it could help the programmers use external libraries within the new language, by facilitating parsing of native debug information formats such as ELF and PE, and C header files. On the other hand Powerlang, should also make it simple for programmers to export those same formats to allow the new language be used within other languages.

2.4 Optimizing

Obtaining the best performance for a programming language is hard. It usually requires implementing JIT- and AOT-compilers, with interpretation, baseline and optimizing stages, using different intermediate representations, complex algorithms for inlining, instruction selection, register allocation and assembly emission. Garbage collection and multiprocess/multithread synchronization primitives are also typical requirements.

Currently, there is no single VM that can tackle all optimization dimensions at the same time. Some VMs optimize for performance, but are big. Others optimize for reducing runtime size, but are slower. Others optimize for dynamicity, allowing more changes at runtime. Others prefer implementation simplicity for security. Each VM framework might require usage of their own set of tools. The programming language design IDE should not stand in the middle. Instead, it should provide ways to plug the initial language implementation to the different VM frameworks, exploiting most benefits of each side.

2.5 Safety

Security is an important aspect of most modern programming languages. It involves different dimensions: programs written in the new language should provide abstractions to diminish the amount of programming errors to the minimum possible; the language itself should aim to be sound regarding to things like its type system or synchronization primitives; the execution environment implementation should be reliable, both to the programmer and to the user of the programs written in the language, allowing to compile and run complex programs according to their specification, without comprising the security of the users of the program. A language implementation framework should provide tools to verify properties of the language itself, or to assure the quality of the runtime environment.

3 Related Work

We already revisited the most relevant works, related to our vision, in the introduction. We briefly remark below the main differences we anticipate between these works and our approach. In contrast to translating a language to an established VM or, developing a simple interpreter within a meta-compilation framework, we advocate, first, that Powerlang would be a much more flexible option. We provide a whole live development environment specifically targeted to design programming languages. Java and Python, host languages of Graal and PyPy respectively, lack the live development experience that Smalltalk grants.

The Micro VM approach consists mainly of a specification of a minimal layer needed to build VMs. From this perspective, the approach is analogous to that of micro kernels for operating systems. The specification can be implemented

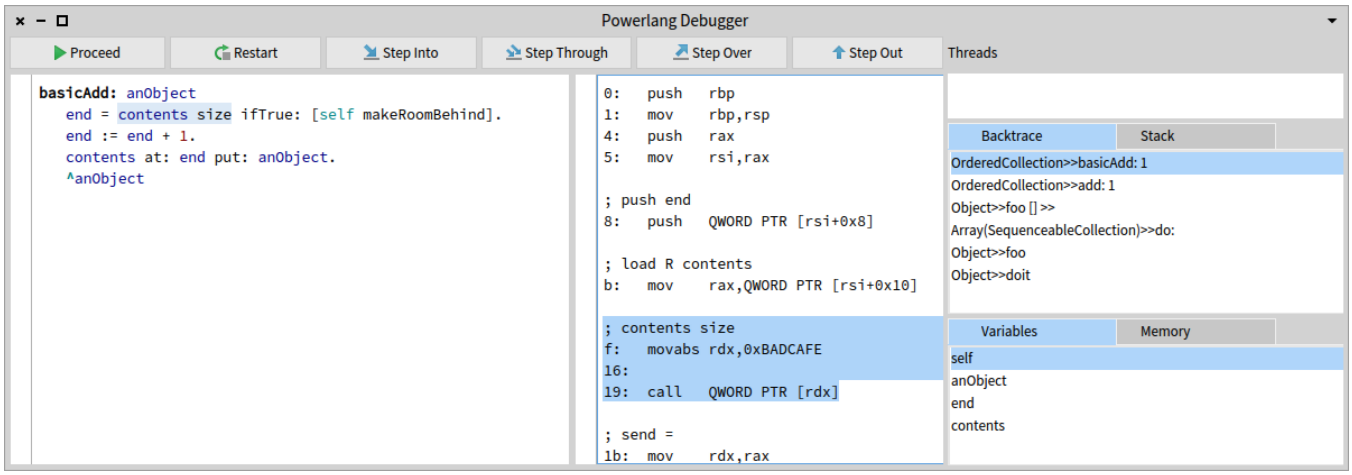

Figure 3. A mock-up of a debugger for Smalltalk using Powerlang. The heap could be running in another process or simulated in an array. The debugger's layout should be adaptable to display the most relevant information of each guest language.

in several different ways. In top of those implementations clients could build different languages tools. Accordingly, the approach follows a bottom-up strategy, leaving to eventual clients the burden of creating the proper abstractions in top of the kernel. In contrast, we propose to follow a top-down approach, by abstracting a whole set of tools, and eventually developing new ones. We focus on live tools to provide an interactive way of developing language runtimes with continuous feedback and on-the-fly adaptations. This way, the effort of clients should be considerably reduced.

4 Final Remarks

We have presented our vision on Powerlang, a live framework aimed to help developers design and implement arbitrary programming languages, without resigning flexibility nor efficiency. As a distinctive characteristic, Powerlang was conceived to provide language developers with a live development experience. We envision that language developers would be able to lively debug and modify, on-the-fly, even the lower-level components of the runtime for their languages.

We already validated several of these ideas while implementing Bee. At this stage we are abstracting a framework based on all the tools we developed during the journey. It still needs to be explored if the ideas apply in general. Our goals for the (near) future are mainly focused on three different concrete directions. First, finish an iteration of our abstraction process to start experimenting building new languages on top of Powerlang. We expect this process to feed us with sufficient feedback for slowly reaching an stable set of tools and APIs. A second interesting research path to explore is the relation between Powerlang and Micro VMs. Namely, is it possible to conceive Powerlang as a set of modules that could be implemented on top of a Micro VM specification or essential incompatibilities would raise in the attempt?

References

[1] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*. ACM, 18–25. https://doi.org/10.1145/1565824.1565827

[2] Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. 2012. On the Benefits and Pitfalls of Extending a Statically Typed Language JIT Compiler for Dynamic Scripting Languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 195–212. https://doi.org/10.1145/2384616.2384631

[3] Open JDK. 2017. Open JDK. http://openjdk.java.net/

[4] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. 2004. *An overview of the Scala programming language.* Technical Report.

[5] Javier Pimás, Javier Burroni, Jean Baptiste Arnaud, and Stefan Marr. 2017. Garbage Collection and Efficiency in Dynamic Metacircular Runtimes. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17)*. ACM, 12. https://doi.org/10.1145/3133841.3133845

[6] Michiaki Tatsubori, Akihiko Tozawa, Toyotaro Suzumura, Scott Trent, and Tamiya Onodera. 2010. Evaluation of a Just-in-time Compiler Retrofitted for PHP. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. ACM, 121–132. https://doi.org/10.1145/1735997.1736015

[7] Kunshan Wang, Yi Lin, Stephen M. Blackburn, Michael Norrish, and Antony L. Hosking. 2015. Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 321–336. https://doi.org/10.4230/LIPIcs.SNAPL.2015.321

[8] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381