Powerlang

Release 0.1.0

Jan 18, 2022

Contents

Powerlang is a research vehicle to explore the implementation of programming laguages. While focused on Smalltalk, it aims to be a generic language creation toolkit, something you can use to either evolve Smalltalk, or either build a new language implementation of choice. We provide tools for coding, bootstrapping, compiling, jitting and debugging the implementation.

Bee Smalltalk serves as a reference implementation using the framework. We are implementing an evolution of the classic Smalltalk-80. Something adapted to the new kinds of systems that are common these days: from embedded to servers, with or without a gui, supporting remote development from scratch.

CHAPTER 1

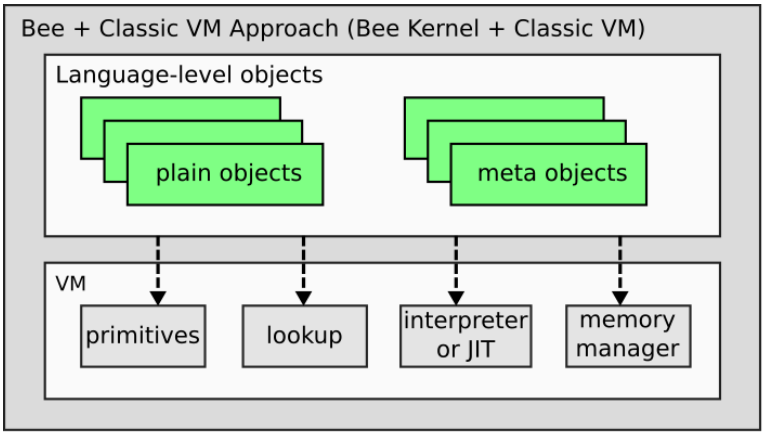
Design

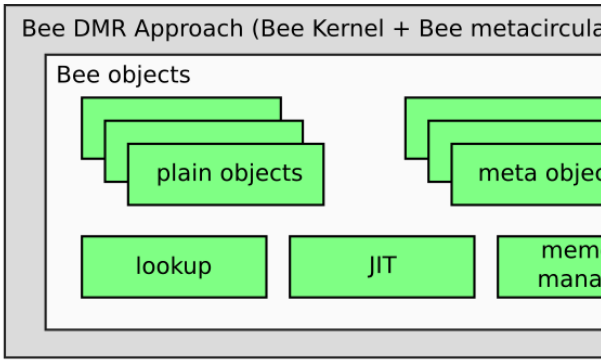
Below is our humble vision of what a modern Smalltalk design should look like, the direction in which we want to go.

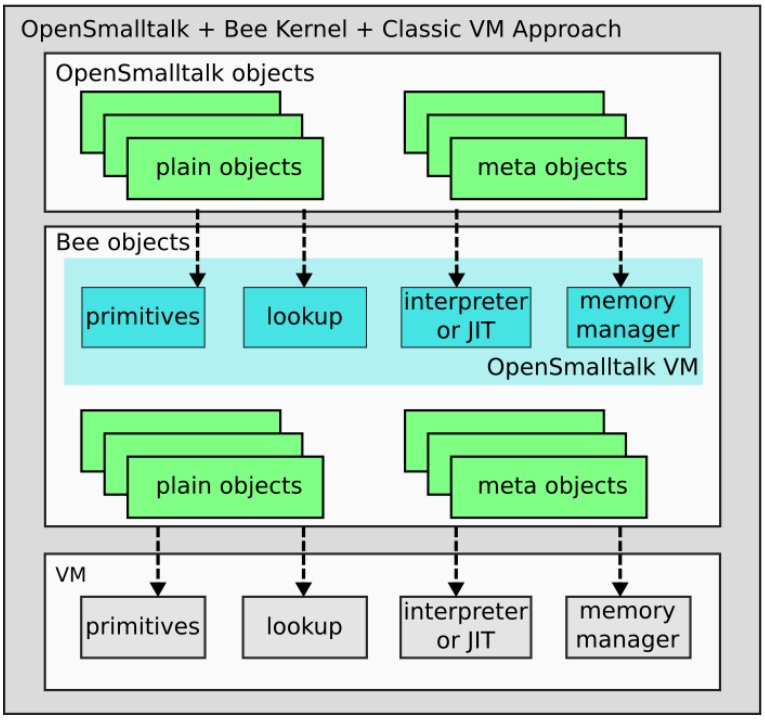
We want a minimal modular Smalltalk, that starts from a little kernel and that can load other modules on-the-fly to grow exactly as needed. The bootstrap process is done mostly from a specification (source code), using different dialects (Pharo et al). This allows sealed-system development which is required when doing big or complex changes, specially in design. Of course, the live environment development will be supported as usual in Smalltalk-80 systems. Computation is represented using Smalltalk Expressions (or SExpressions), a lower-level representation of abstract syntax trees (ASTs) that is encoded into byte arrays (called astcodes).

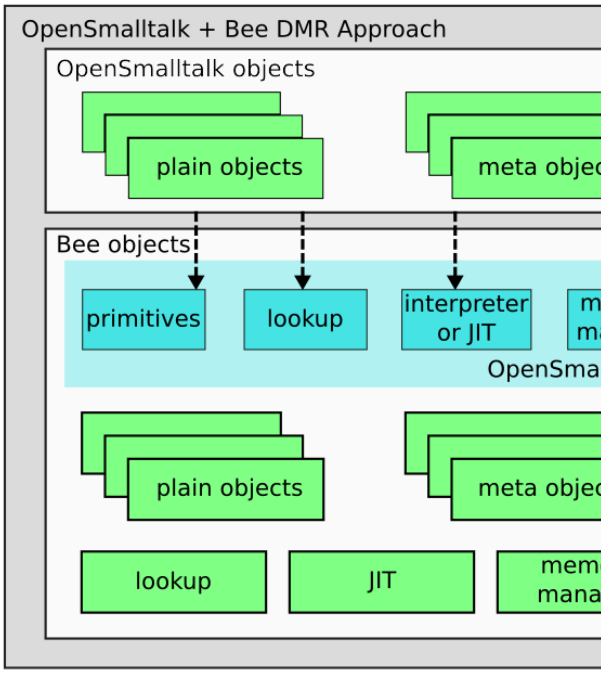
Namespaces are supported from the beginning, and form the base of packages. Packages are built into (binary) image segments that load very quickly. A package distribution system computes dependencies and fetches prebuilt image segments for quick setup, update and deployment.

The base system can be used to allow creation of different dynamic languages. In particular, we expect Bee Smalltalk to expand on two different axes: on one hand, it can grow to become a full, tightly integrated Smalltalk; on the other hand, its kernel can be the base system on top of which we support other Smalltalks (or other languages). The main candidate for the latter approach is to make the opensmalltalk-vm run on top of Bee kernel, instead of translating it to C and compiling. This would make Squeak/Pharo/Cuis run on top of Bee, which could either run self-hosted (if using the DMR) or on top of another VM (like the OMR). Below we present a look of different possible mixes and matches of language environments and hosting runtimes.


Bee + Classic VM Approach (Bee Kernel + Classic VM)
Language-level objects
plain objects
meta objects
VM
primitives
lookup
interpreter or JIT
memory manager


Bee DMR Approach (Bee Kernel + Bee metacircula
Bee objects
plain objects
lookup
JIT


OpenSmalltalk + Bee Kernel + Classic VM Approach
OpenSmalltalk objects
plain objects
meta objects
Bee objects
primitives
lookup
interpreter or JIT
memory manager
OpenSmalltalk VM
plain objects
meta objects
VM
primitives
lookup
interpreter or JIT
memory manager


OpenSmalltalk + Bee DMR Approach
OpenSmalltalk objects
plain objects
Bee objects
primitives
lookup
interpreter or JIT
plain objects
lookup
JIT

CHAPTER 2

Implementation

Powerlang-based systems should work on Windows, Linux, Mac, Android and nopsys, including embedded platforms. 64 and 32 bit architectures are the starting point, but if possible we may try even smaller.

At least two runtimes are being developed in parallel. The pure DMR (dynamic metacircular runtime), and the Eclipse OMR. The DMR is an AOT-based approach, that uses Smalltalk to pre-nativize a Smalltalk JIT compiler, and then uses that compiler to nativize on the fly other Smalltalk code. The OMR is a state-of-the-art C++ runtime engine, that includes plugabble interpreters, JIT-compilers and garbage collectors, which can be used to support languages like Smalltalk.

Smalltalk source code is stored on git, using a tonel-like format. We store just code and definitions in the repo, no artifacts. Build artifacts go through Continuous Integration from the very beginning. Each OS platform is put in a separate namespace and built into separate image segments, which are autoloaded according at startup according to the running platform. The system can be debugged remotely through a vdb/gdb connection, which allows both local and remote debugging.

CHAPTER 3

Setup

To setup the development repository, see the instructions in Powerlang repo.

CHAPTER 4

How to improve this documentation

This documentation was written in rst format and html output was generated by sphinx. You will find these same contents in */docs* directory. To be able to compile them, you will need to install sphinx in your system:

```
pip3 install sphinx sphinx-autobuild
```

then just from root powerlang dir do:

```
sphinx-autobuild docs docs/_build/html
```

which will open a local http server at http://127.0.0.1:8000 that recompiles automatically when files are changed and that lets you browser the docs as you change them. When you have any change, just open a PR in github.com/powerlang

CHAPTER 5

Indices and tables

5.1 Bootstrap

Powerlang is the base system used to generate Smalltalk images from Smalltalk source code written in files. We expect it to let generate images for other languages in the future.

For now, Powerlang runs on top of Pharo, and consists of a set of packages that allow for loading code definitions, to compile them, build image segments, and to generate native code for methods ahead of time, which is required when targetting the DMR runtime.

Code is loaded into a *Ring2* environment, which is as a set of objects that specify what is included in a module. Currently, the main supported code-base is that of Bee Smalltalk. Powerlang has been developed hand-in-hand with Bee, with the hope that in the future other systems are also supported. The Bee code is in a separate git repository, which the makefiles automatically clone into *bootstrap/specs/bee-dmr*.

Bee consists of a main *Kernel* module, which is self-hosted. This means that the kernel is able to perform basic Smalltalk computation without accessing nor requiring other modules. The Kernel includes objects like numbers, collections, classes, methods and modules. If other things are to be used, they are loaded in a modular fashion: OS support, JIT compiler, libraries are out by default but a dependency tracking system allows will allow to easily add stuff.

5.1.1 Bootstrap steps

Generating an executable image segment requires a series of steps, which we describe next. You can also study these steps by looking at *Powerlang-Building* packages. The process is the following:

1. Kernel Genesis

`VirtualSmalltalkImage fromSpec wordSize: n; genesis` instantiates a virtual image that reads Bee *Kernel* module definition, and then builds the objects required for the classes, metaclasses and behaviors. Objects generated during the genesis are of type `ObjectMap`. This hierarchy of types allows to represent the contents of the newly created object slots, and their corresponding spec type and associated behavior and hash. The objects created

by this virtual image are the minimum needed to do any kind of Smalltalk execution. However, it doesn't contain any method, as compiling methods is a more complex step that requires bootstrap initialization of globals, class vars and pool vars. It doesn't even contain the Smalltalk object, which is generated later.

2. Bootstrap duality

Initialization of globals and pools is done through execution of Smalltalk code within the virtual image, and for that we use a `VirtualSmalltalkRuntime`.

However, compiling methods during the bootstrap is more complex than compiling for the current image. The compiler runs on top of the current image, and generates *SCompiledMethods* with literals, which can be integers, symbols, arrays, *SCompiledBlocks*, etc. Those methods and their referenced objects need to be converted to *ObjectMaps*, and end in the bootstrapped world. On the other hand, when the compiler builds methods from source, it may need to access things living in the bootstrapped image. For example, if the compiler finds the sting `Array`, it should generate a method with a literal frame that contains the association #Array -> Array, that belongs to the bootstrapped world.

There is a constant sense of duality while compiling and executing virtually: objects need to be passed back and forth from the local image to the bootstrapped image and vice-versa many times. To deal with this, the method compiler is passed *VirtualClasses*, which account for both the Ring specs and the *ObjectMap* that represents the class in the bootstrapped system. To allow usage of globals and pools or class vars, the compiler the compiler uses *VirtualDictionaries*, which know both their keys as symbols in the local image and also the associations and values that live in the botstrapped image.

3. Bootstrap initialization

After compilation, generated SCompiledMethods have references to both objects in the current image and objects in the bootstrapped image.

And there's yet one last twist: pool dictionaries. Pool dictionaries are more dynamic than class variables. While class variables are all determined beforehand (in class definition), pool variables are defined after some initialization: class variables that point to objects of type `PoolDictionary` are recognized by the compiler, and are used as local pools by it. Before it is possible to compile arbitrary methods, the builder has to initialize pool dictionaries. To do so, it virtualy sends the message `#bootstrap` to the module object corresponding to the module spec being built. The virtual runtime interprets the message send, creating more *ObjectMaps*. During that process the compiler doesn't yet recognize pool variables, so pool vars can't be used to initialize pool vars (it shouldn't be a big limitation though). One extra step is done by the bootstrapper: it sends `declareGlobals` to the module, so that any global object name is put into the module namespace.

After this initial pass, arbitrary methods can be compiled, so the bootstrap process creates the method dictionaries of the classes and fills them. Finally, a last initialization pass is done by sending `initialize` to the module, which can execute more complex initialization code.

3. Module nativization

This step shall be optional (only required by the DMR for a reduced set of modules). The compiler in the local image generated instances of `SCompiledMethod` which contain s-expressions. An SExpressionNativizer traverses them and generates native code for each method and block. The result of this is stored in their *nativeCode* ivar and then transferred into the image segment being built. Kernel nativization for the DMR also requires some extra steps, such as the nativization of invoke, lookup, and write barrier procedures.

4. Image segment wrap-up

During all the bootstrap process all important objects are created by the builder. To finally generate an image segment, the builder creates an `ImageSegmentWriter` and passes it the roots, an ordered list of objects from where it will calculate what are the objects to be included in the file. This step writes the objects put into a binary stream with a particular format that can be loaded by a launcher written in C++. The launcher is in */launcher* directory. The writer is specified a base address at which the file should be loaded in memory, and knows how to encode each object oop according to that address.

5.2 DMR runtime

The Dynamic Metacircular Runtime flavor of powerlang is an implementation of a runtime that is completely self hosted. There is no other "VM", the generated image contains everything needed to execute its code.

The SExpressions (Source Expressions) are AOT nativized, so that all needed code can be executed and, when required, a JIT compiler can be loaded into that image so that new code can be nativized on-the-fly.

Memory and garbage collector are written in the same language as the rest of the system and incorporated into the bootstrapped image when building the system from the spec.

5.2.1 ABI of the DMR runtime

The native code of the DMR follows a series of rules that form an invariant which must be maintained while executing managed code. An optimizing compiler might alter those rules during moments considered atomic, but in general the rules apply all the time.

The general design favors design simplicity as the most valued property. Then comes performance, which should be take into account too.

Execution context

The DMR is designed as a register machine with a stack. There is a set of abstract registers, which are mapped to concrete ones depending on the target processor ISA:

Table 1: Registers!

Abstract name	Concrete name (amd64)	Type	Description
IP	RIP	callee-saved	Instruction Pointer
SP	RSP	callee-saved	Stack Pointer"
FP	RBP	callee-saved	Stack Frame Pointer
R	RAX	volatile	Receiver and Return value pointer
M	RBX	callee-saved	Method/Block native context
S	RSI	callee-saved	Self
E	RDI	callee-saved	Environment
A	RDX	volatile	Argument
T	RCX	volatile	Temporal
V	R11	volatile	Volatile
nil	R12	fixed	nil
true	R13	fixed	true
false	R14	fixed	false
G	R15	fixed	commonly used global objects

IP (Instruction Pointer), SP (Stack Pointer), and FP (Frame Pointer) are self describing. R register contains the Receiver at the instant at which a message is going to be sent, and contains the Return value at the moment when a method is about to exit. When entering a method, R register (which is volatile) is stored into S (Self), which is callee-saved. This allows to have a pointer to self permanently in a register while executing a method. Previous S is restored by the callee at exit, loading it from the stack frame of the caller. M (currenty executing Method) provides access to a method or block's native code and literals. NativeCode objects (the ones pointed by M) know the CompiledMethod or CompiledBlock that generated them, and the literals used within native code. They also point to the byte array that holds the machine instructions to be used by the processor. M is restored when returning in the same way than S. A, T and V register names are just denotational. This means they were named like that because of their main uses, but they can be used for different things. They are usually free, ready for usage. We describe the way they work first and give some examples later. A (Arg0) is used whenever a register is needed for fast/inline arguments, like with inlined binary integer operations. It is not used for passing real arguments in message sends. T (Temp0) is used to store temporary values during operations that require a free register. V (Val0) is used to load constants. It is needed because typical ISAs do not let use full 64-bit constants in instructions, so to use a big constant (like a pointer) you must first load it into some register. Nil, true and false registers are loaded when entering from C code, and leave like that forever (as in C they are callee-saved, it is not necessary to restore them when calling C code).

Arguments, temporaries and environment

Arguments are pushed into the stack from left to right. They are not passed in registers because that would complicate the general design and also debugging. Temporaries are stored in the stack for the same reasons. Leftmost temporary is pushed first, which usually means it is stored at the higher addresses (amd64). When a method has temporaries shared with child blocks, or non-local returns, it creates an environment and pushes it (and the previous one) into the stack before temporaries. The environment is an object of class array, that will have as many slots as shared temporaries.

Block closures also create an environment, again of a size that is equal to the amount of temporaries they share with their child blocks.

Native code examples

Todo: add a couple of examples of usage of registers in native code

Todo: add an example picture of the stack

- genindex
- modindex
- search

CHAPTER 6

Things to do (PRs are welcome!)

Todo: add a couple of examples of usage of registers in native code

original entry

Todo: add an example picture of the stack

original entry