A Bootstrapping Infrastructure to Build and Extend Pharo-Like Languages

Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi

A Bootstrapping Infrastructure to Build and Extend Pharo-Like Languages

G. Polito   S. Ducasse   L.Fabresse   N.Bouraqadi

RMoD Project-Team, Inria Lille–Nord Europe, France
CAR Team, Institut Mines-Telecom, Mines Douai, France

Abstract

Bootstrapping is well known in the context of compilers, where a bootstrapped compiler can compile its own source code. Bootstrapping is a beneficial engineering practice because it raises the level of abstraction of a program making it easier to understand, optimize, evolve, etc. Bootstrapping a reflective object-oriented language is however more challenging, as we need also to initialize the runtime of the language with its initial objects and classes besides writing its compiler.

In this paper, we present a novel bootstrapping infrastructure for Pharo-like languages that allows us to easily extend and modify such languages. Our bootstrapping process relies on a *first-class runtime*. A first-class runtime is a meta-object that represents a program's runtime and provides a MOP to easily load code into it and manipulate its objects. It decouples the virtual machine (VM) and language concerns by introducing a clear VM-language interface. Using this process, we show how we succeeded to bootstrap a Smalltalk-based language named Candle and then extend it with traits in less than 250 lines of high-level Smalltalk code. We also show how we can bootstrap with minimal effort two other languages (Pharo and MetaTalk) with similar execution semantics but different object models.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible Languages

General Terms Languages, Object-Oriented Programming

Keywords Bootstrapping, OOP, Traits, Pharo, Metaprogramming

1.   Introduction

A language initialization, or *bootstrap*, is the process where the initial elements of the language are set up. In high-level languages, such as Ruby or Smalltalk, this bootstrap requires also the creation of initial objects and classes such as the Object base class and the Boolean true and false objects (cf. Section 2). The language bootstrap usually fixes the semantics of the language, for safety reasons or language design. It is however desirable to have easy access to the language to modify it and extend it. For example, studying Pharo [BDN+09], a smalltalk-inspired language and platform, we identified a need for better support in the introduction of new features such as traits [SDNB03] and first-class instance variables [VBLN11].

Changing an existing language is indeed challenging without the proper knowledge or infrastructure. On one hand, languages whose language bootstrap is VM-based (*i.e.,* defined inside virtual machine (VM) routines) fix several language features and prevent us to change it without changing the VM. In addition, such routines may mix VM and language concerns, making the code harder to understand and change. On the other hand, reflective languages [Smi84] such as Lisp or Smalltalk provide the means to modify the language elements at runtime. However, as these languages contain circular definitions [CKL96] we need to stage these changes to avoid metastability problems [KdRB91] *i.e.,* a change in the language may introduce a meta-call recursion and turn the system unusable.

This paper takes a high-level low-level programming approach [FBC+09] and revisits an already well-known technique: an object-oriented reflective language **bootstrap**, as it is known in the context of compilers (*i.e.,* a circular bootstrap, where a compiler can compile itself). We believe this concept has not been fully explored for object-oriented reflective languages, where the necessary infrastructure is still *ad hoc* and the impact it has on the development process of

language engineers is usually unknown. In this context we pose the following research question: *What is the infrastructure required to execute a circular bootstrap of an object-oriented reflective language?*

In this paper we propose a circular bootstrap infrastructure that allows us to build an object-oriented reflective language such as Pharo using itself (cf. Section 4). Our solution overcomes the metastability issues in an elegant manner by transparently breaking the circular definition of the language during its definition: the language bootstrap is described in itself using the full power of the language but executed by a specialized interpreter that manipulates the language elements as behavior-less data structures. It also avoids to duplicate code in an external VM-based bootstrap by reusing the means to create the language's structures already available in reflective languages.

Once bootstrapped we can easily modify the language to change critical parts of the language runtime and extend the language model adding *e.g.,* traits [SDNB03] without VM changes and avoiding the staging required by circular definitions of the language (cf. Section 5). A **first-class runtime** in our infrastructure provides a high-level API to manipulate the low-level wirings of the language (cf. Section 6). This allows us to easily extend our bootstrap and thus, our language. Additionally it makes a clear distinction between VM and language concerns. This allows us to modify the language without changing the VM.

We implemented our bootstrapping infrastructure on top of the Pharo language (cf. Section 7). Finally, we evaluate it by conducting and measuring different experiencies with languages with similar execution as Pharo (cf. Section 8). First, we show how we can bootstrap a Smalltalk implementation named Candle and extend it with traits; then we bootstrapped from scratch MetaTalk, a Smalltalk implementation with mirror based reflection, and the Pharo language that includes extensions such as traits and first class instance variables. These languages can then run on a stock VM without modifications. The co-evolution of VM code and language will be addressed in future work.

2. Bootstrapping Definitions

Bootstrapping is known in many different contexts with different meanings (*e.g.,* a machine's startup process, the process to build a more complex system from a simpler one). Particularly in programming languages we can see it is broadly used with two different meanings: the startup of the language runtime elements (*e.g.,* as in Java's bootstrap class loader [LB98]) and the self-compilation also known as a compiler's bootstrap. To avoid confusions in terminology, this section states the definitions we use in this paper. Piumarta et al. [Piu06] define a programming language as follows:

DEFINITION 1 (Programming Language). *A programming language L is a combination of syntax, semantics and pragmatics, a compiler and a runtime.*

The syntax is the set of rules that restricts the legal content of a program's source code. The semantics are the meaning of that content (*e.g.,* how a method invocation will be executed at run-time). The pragmatics are the range of visible effects and values that are possible during a program execution (*e.g.,* the range of available integer values). The compiler is a program that translates a program's source code written in our language to a machine executable run-time representation of that program (cf. Figure 1). The runtime is the software that support the execution of a program at runtime, providing the builtin structures and functions available in the language (*e.g.,* the initial objects of the language) and services such as memory management and cross-cutting optimizations. Notice that the language syntax, semantics and pragmatics are embedded within the compiler and the runtime.
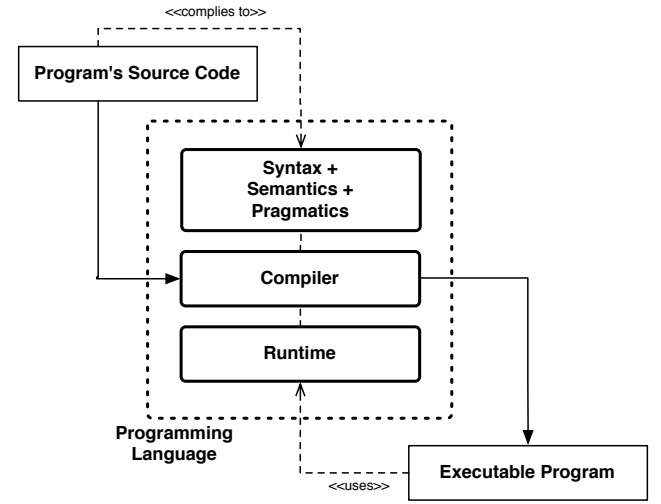

Figure 1. How a language is composed.

Based on this definition, we can define bootstrapping as follows:

DEFINITION 2 (Language Definition). *A language definition D_L of a programming language L is a textual representation all elements of L. That is, it is the source code used to create an executable runtime and compiler for L.*

DEFINITION 3 (Programming Language Bootstrap Process). *A bootstrap process of a programming language L is a process that produces a programming language L from a language definition D. That is, it produces the set of its syntax, semantics and pragmatics, its compiler and runtime.*

DEFINITION 4 (Circular Bootstrap Process). *A circular bootstrap process (or self-bootstrap process) of a programming language L is a bootstrap process of L where its definition D_L is expressed in L.*

Example: The process that creates a C compiler from assembly source code is the bootstrap of this C compiler. The process that creates a C compiler from C source code of the compiler is a circular bootstrap of the C compiler.

We cannot apply our bootstrap definition to VM-based languages as we do with a language such as C. In VM-based languages the program's runtime is responsible of the VM pursuing (mainly) the goals of abstraction and portability. To consider these cases and make such a distinction we introduce first the following definitions:

DEFINITION 5 (Language Runtime). *The language runtime R_L of a programming language L is the set of elements and structures that belong to the language and are required to run a program.*

DEFINITION 6 (Virtual Machine Runtime). *The virtual machine (VM) runtime VMR_L of a programming language L is the set of elements and structures that belong to the virtual machine and are required to run a program. These elements are often invisible to the language.*

Example: The language runtime of the languages Smalltalk and Ruby are the initial objects and classes of the language *e.g.,* nil, true, false, Object class and Class class. Their VM runtimes implement support for memory management and program execution. VMs manipulate these programs but are however invisible to them (cf. Figure 2).
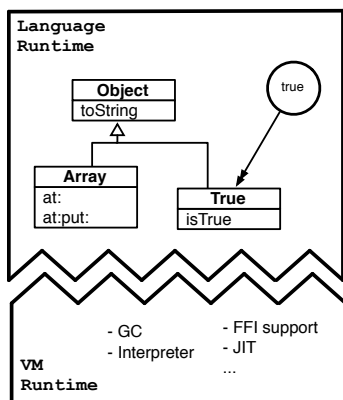

Figure 2. VM and Language Runtime.

A bootstrap of a VM-based language should create the initial objects and classes of the language runtime. Then, just writing a compiler for a VM-based language L in the language L is not enough to bootstrap a VM-based language because these compilers produce often *bytecodes* that do not include runtime information (this is the case of *e.g.,* Java's and Smalltalk's compilers).

DEFINITION 7 (Partial Bootstrap Process). *A partial bootstrap process of a language L is a process that produces a part of language L from a language definition D* i.e., *it pro-*

duces its syntax, semantics, pragmatics, compiler, runtime or a combination of them, but not all of them.

Example: Just by themselves, a self-compiling Smalltalk compiler and the startup of the Smalltalk's language runtime are partial bootstrap processes. However, their combination results in a complete bootstrap of the Smalltalk language.

In this paper, with the objective of changing the programming model of a programming language we focus on the *circular bootstrap of its language runtime*. Bootstrapping a compiler is out of the scope of this paper as it is a more known technique.

DEFINITION 8 (Language Runtime Bootstrap Process). *A language runtime bootstrap process for a programming language L is a process that starts (produces) the language runtime of a program that is written in L from a language runtime definition R_L.*

DEFINITION 9 (Language Runtime Definition). *A language runtime definition R_L for a language L is the definition of the initial runtime structures of L, so a program can run on it* e.g., *the initial objects and classes.*

We can combine these last definitions with Definition 4 to define *e.g.,* a partial circular bootstrap process. We skip those definitions for space reasons and because they do not bring ambiguity.

3. Challenges of Extending a Language Runtime

In this section we present the problems raised by VM-based language runtime bootstraps. Then, we present the specific challenges that circular bootstrapping has to resolve to be beneficial.

3.1 Fixed language runtimes

Object-oriented languages have often a VM-based language runtime bootstrap *i.e.,* the runtime bootstrap is described in routines that are part of the VM. This decision is indeed practical as the VM can safely initialize the language structures and solve the language bootstrapping issues avoiding recursions [KdRB91] (*e.g.,* create the first-class without a class). For example, Figure 3 shows an excerpt of the code that initializes the class hierarchy in the Ruby VM written in C[1]. From this code, Ruby's basic class hierarchy is composed by BasicObject as its root, followed by Object, Module and Class. These classes are created manually without a class, and once the class Class is available, their class references are updated. Similar code is in place to initialize basic objects such as nil, true and false and other classes.

This piece of code shows at first glance that the VM fixes the classes and object model of the language. The

---

[1] Taken from the version 2.1 of the Ruby VM in http://svn.ruby-lang.org/repos/ruby

```
void Init_class_hierarchy(void) {
  rb_cBasicObject = boot_defclass("BasicObject", 0);
  rb_cObject = boot_defclass("Object", rb_cBasicObject);
  rb_cModule = boot_defclass("Module", rb_cObject);
  rb_cClass =  boot_defclass("Class",  rb_cModule);

  rb_const_set(rb_cObject, rb_intern("BasicObject"),rb_cBasicObject);
  RBASIC_SET_CLASS(rb_cClass, rb_cClass);
  RBASIC_SET_CLASS(rb_cModule, rb_cClass);
  RBASIC_SET_CLASS(rb_cObject, rb_cClass);
  RBASIC_SET_CLASS(rb_cBasicObject, rb_cClass);
}
```

Figure 3. Code of the Ruby VM that initialises the class hierarchy (excerpt). The VM code fixes the language class hierarchy.

language object model is fixed and prevents us to easily change it without recompiling the VM. A second side-effect of this decision is that we manipulate objects of a language using the wrong level of abstraction. Indeed, we see the objects as memory structures and we manipulate them using pointer arithmetic instead of the usual abstractions that the language offers. Finally, this kind of language bootstrap is a source of *spaghetti code*, as it makes easy to mix VM and language concerns. It becomes also difficult to recognize whether a piece of code belongs to one or the other. Figure 4 illustrates this with an excerpt of the JikesRVM[2] [AAB+00]. In this example, the memory manager is initialized in the middle of the initial class loading phase. This introduces a temporal coupling that prevents changing this code without VM specific knowledge.

```
private static void finishBooting() {
  ...
  MemoryManager.postBoot();
  ...
  runClassInitializer("java.lang.Runtime");
  runClassInitializer("java.lang.System");
  runClassInitializer("sun.misc.Unsafe");
  ...
  MemoryManager.fullyBootedVM();
  ...
  runClassInitializer("java.util.logging.Level");
  runClassInitializer("gnu.java.nio.charset.EncodingHelper");
}
```

Figure 4. Code of the JikesRVM that initialises the initial classes of the runtime (excerpt). The code performing the initial class loading is mixed with the code that initialises the memory manager of the VM.

---

[2] Taken from the version 3.1.3 of the JikesRVM in http://sourceforge.net/projects/jikesrvm

3.2 Circular Bootstrapping

To solve the problems above we propose the introduction of a circular bootstrap process. Bootstrapping is a beneficial engineering practice because it raises the level of abstraction of a program, following the principles of *high-level low-level programming i.e.,* expressing low-level concerns using high-level languages [FBC+09]. High-level low-level programming simplifies the complexity of a language runtime bootstrap in several ways:

Abstraction. Developers benefit from the abstractions of the high-level language they are using. For example, inheritance and polymorphism permit better ways to organize a program's behavior, enhancing developer's productivity. Also, they may benefit from the services that the high-level language provides such as garbage collection or cross-cutting optimizations.

Tooling. Developers can edit or debug the language runtime creation using the tools they normally use to edit their high-level programs. They do not depend on several sets of tools, or on doing a *mind-shift e.g.,* thinking in Ruby or Smalltalk objects while programming with C memory-structures.

3.3 Challenges of Circular Bootstrapping

While a circular bootstrap is indeed beneficial, we identify two different challenges that arise from building a circular bootstrap process of a language runtime:

Differentiate VM and Language Concerns. To minimize the changes in the VM when modifying a language, we need to understand which are VM concerns and which are language concerns. For example, a main VM concern is the language's *execution model i.e.,* the set of rules that the language follows to execute code. Changes that require modifying the execution model (*e.g.,* changing the way messages are passed between objects) will probably require VM modifications. By understanding these differences, we can however build new features without changing the execution model *e.g.,* the Pharo and Ruby languages introduce implicit metaclasses[3] while the VM has no knowledge about them.

Manage Circular Definitions. Introducing features such as traits [SDNB03] in an existing language becomes more challenging when we use traits to define the language runtime itself. For example, we can use traits to define the collection library of our language to simplify the collection library code [BLDP10]. However, this introduces circular definitions in the language, as probably the language runtime is defined in terms of basic collections also. Introducing such changes is often complex and requires a staged process *e.g.,* first define the language run-

---

[3] an implicit metaclass is a class that is created for each class automatically

time without traits, then introduce the notion of trait, finally redefine the language runtime with traits.

These challenges leads us to our main research question:

What is the infrastructure required to execute a circular bootstrap of an object-oriented reflective language?

4. A Circular Bootstrapping Infrastructure

To answer the question above, we present Seed, our solution based on a specialized code interpreter and first-class runtimes. Our solution succeeds to bootstrap Pharo-like languages *i.e.,* object-oriented reflective languages that run on top of the Pharo VM. The reflective property of these languages allows us to reuse the reflective API of the language to bootstrap itself. Additionally, the Pharo VM imposes a class-based with single inheritance object model. We believe however that this infrastructure can be adopted by other reflective object-oriented languages. We illustrate our examples with the Candle language, a simple Smalltalk-based language that runs on top of the Pharo Virtual Machine[4]. Section 5 shows how this infrastructure is used further to bootstrap other languages with different object models but same execution semantics.

4.1 Our Solution in a Nutshell

We propose Seed, a bootstrapping infrastructure for Pharo-like languages based on a *bootstrapping interpreter* and a *first-class runtime* (Figure 5). We use an example to show the different extension points required to support extensions.

The Candle Example. To show our solution let's consider the circular language runtime definition of the Candle language (Figure 6). This *definition* is based on MicroSqueak [Mal]. We adapted it to run on top of the latest Pharo VM and we added manually some the code for the initial startup and class initialization that was missing.

This language definition is the main point of extension of our language: code refactorings, removals and additions of classes and methods, can be applied by directly modifying this definition. The **bootstrapping interpreter** is a code interpreter (*e.g.,* an abstract syntax tree interpreter) that interprets this definition to bootstrap the language runtime: create the classes and objects defined in it, and execute specific code to initialize them. The bootstrapping interpreter is additionally in charge of solving the bootstrapping issues of the language *e.g.,* creating a first object without a class, and creating a class without a class. This interpreter is the second point of extension of our infrastructure: we need to extend it when we add circular behavioral extensions to our language, such as traits [SDNB03] or first-class instance variables extending the language semantics [VBLN11].

The bootstrapping interpreter interacts with the language runtime under construction through a **first-class runtime** object. A first-class runtime is a meta-object representing the language runtime and providing a meta-object-protocol (MOP) [KdRB91] to manipulate it. This meta-object provides a high-level API and encapsulates VM concerns during the bootstrap, so we only have to deal with language concerns. Section 6 discusses about first-class runtimes and their MOP.
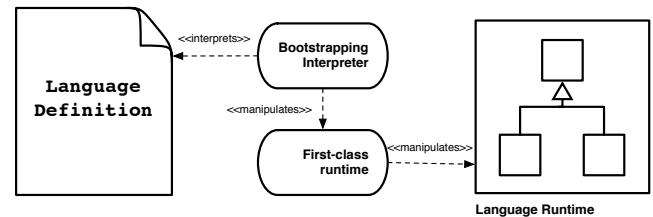

Figure 5. Solution overview. A bootstrapping interpreter uses the self-description in the language definition to build the language through the a clear VM-language interface.

```
"Newly added code for initial objects"
nilObject := UndefinedObject basicNew.
trueObject := True basicNew.
falseObject := False basicNew.

"existing code"
nil subclass: #ProtoObject
  instanceVariableNames: ''.

ProtoObject subclass: #Object
  instanceVariableNames: ''.

Object subclass: #UndefinedObject
  instanceVariableNames: ''.

ProtoObject >> isNil
  ^ false

UndefinedObject >> isNil
  ^ true

"Newly added code for class initialization"
String initialize.
```

Figure 6. Excerpt of Candle's definition. It includes the creation of basic objects, classes and methods.

Once the execution of the language definition is finished, the language runtime is considered as bootstrapped and programs can be loaded and run on top of it. We believe from our observations in existing code from other language implementations, that this infrastructure can be generalized and used in languages such as Ruby, Javascript or even Java. This issue is however not further discussed in the scope of this paper and is a future work.

---

[4] The name Candle comes from the idea of a small light. Candle is inspired in the Pharo language, a bigger light.

4.2 Solving Bootstrapping Issues

Once we have a language definition as the one in Figure 6, our next question is: How do we execute[5] such definition? Indeed, initially our language runtime contains no classes, methods nor objects. We identify two main bootstrapping problems or *paradoxes* that arise from executing the definition's code: inexistent classes and inexistent methods. Let's illustrate these problems by studying the following expression:

UndefinedObject basicNew.

Bootstrapping issue #1: Inexistent Classes. During the execution of any expression at bootstrap time, we may find inexistent classes as UndefinedObject is in the example. Creating the inexistent class at this moment would introduce another *paradox*, because creating the UndefinedObject class requires even more classes to exist such as String, Array or Class.

 Solution: stub classes. Our bootstrap process breaks the circularity by transparently introducing stub classes. Stub classes contain the minimal requirements to execute simple operations such as the instantiation primitive (basicNew in this example). Later, when the real classes are created by the process, stubs are replaced.

Bootstrapping issue #2: Inexistent Methods. Since our circular bootstrap is an object-oriented program, any operation we want to apply should be resolved as a *message-send* to an object *e.g.,* the message basicNew in the example. However, even if classes exist, methods may not be yet installed in the system.

 Solution: alternative method lookup. Our bootstrap process resolves message sends by providing an alternate method lookup. Instead of looking-up methods in the (possibly incomplete) class hierarchy of the bootstrapped runtime, the bootstrapping interpreter performs the method lookup in the language definition where all the information is available.

Figure 7 illustrates the behavior of the bootstrapping interpreter during the execution of the "UndefinedObject basicNew" expression. First, if the class UndefinedObject does not exist, it creates a stub UndefinedObject class and maps it to its corresponding definition in the language definition. Further usages of this class will use the existent stub instead of creating new ones. Next, to interpret the basicNew message the interpreter looks it up in the class of the object in the language definition. As the class from the language kernel and the language definition are mapped, the interpreter knows where to start the method lookup. Finally, the found method is executed in the language kernel and an instance of the UndefinedObject class is created.

5. Extending a Bootstrapped Language

The Seed infrastructure provides two main language extension points: the language definition and the bootstrapping interpreter. In this section we evaluate how easy it is to extend Candle with traits. We first introduce the notion of traits for the end-user of the language. In a second step we introduce a circularity in the language: we define traits using themselves. Besides we apply this change in two stages for clarity, our infrastructure can apply all these changes at the same time simplifying the bootstrapping process.

5.1 Extending the Definition: Introducing Traits

Modifications that do not change the language execution semantics only require extending or modifying the language definition. This is the case of changes such as renames, additions and removals of classes and methods or modifications in the class hierarchy. Addressing a bug in the language bootstrap becomes simple as we don't require modifying nor understanding VM code to do such a change.

We can implement a simple version of traits[6] in Candle with method flattening. Trait methods are directly installed inside the classes that use them. This allows us to add traits without modifying the VM execution semantics. Our trait definition is as follows:

```
Object subclass: #Trait
    instanceVariableNames: 'name methods'.

Trait >> addMethod: aMethod
    methods add: aMethod

"[...]methods for installing methods into traits[...]"

Trait >> uses: aTrait
    aTrait methods do: [ :m | self installMethod: m ]

Class >> uses: aTrait
    aTrait methods do: [ :m | self installMethod: m ]
```

As long as no class in the language definition uses traits, traits are not needed during the bootstrap's execution, limiting our changes to the language runtime definition. Once bootstrapped, the language user can define its own traits and use them inside his application.

5.2 Extending the Interpreter: Circular Traits

Circular definitions add new bootstrapping issues, as they alter the execution and semantics of the bootstrap. The boot-

[5] The execution of such language definition will produce a bootstrapped language.

[6] The author will notice that this version is not a complete version as it does not implement *e.g.,* conflict validation. A complete trait implementation is out of the scope of this paper.
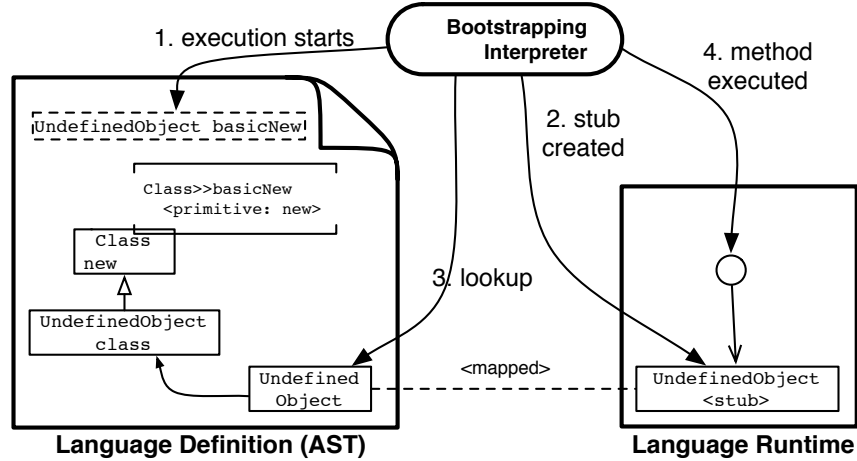
Figure 7. The Bootstrapping interpreter in action. A stub class is created for a non existent class. Each class is mapped to its description in the language definition. The lookup is then performed inside the language definition. Once the method is found, it is executed inside the language kernel.

strapping interpreter should be aware of these circular definitions to properly resolve them. Fortunately extending the interpreter takes only a couple of special lines of code (besides extending the parser).

To continue with our example from above, we can use the power of traits inside our language to define traits themselves. We can for example factor out the method uses:, repeated in Trait and Class, into a trait TTraitable. We modify the Candle definition as follows, defining our new trait and modifying the classes Trait and Class to use such trait.

```
Trait named: #TTraitable.

TTraitable >> uses: aTrait
    aTrait methods do: [ :m | self installMethod: m ]

Object subclass: #Trait
    uses: TTraitable.

Object subclass: #Class
    uses: TTraitable.
```

This change introduces a new circularity: the uses method that defines trait usage is defined by a trait. The bootstrapping interpreter needs knowledge on traits to run the new bootstrap. We can extend the AST of the bootstrapping interpreter with trait knowledge in a couple of high-level code methods as in:

```
BootstrapClassAST >> methods
    "this method is used during the bootstrap method lookup"
    ^ super methods , self traitMethods

BootstrapClassAST >> traitMethods
    "appends all the methods from the traits"
    ^ self bootstrapTraits gather: [ :trait | trait methods ]
```

At the end, this change included 30 lines of code of the trait implementation and 215 lines of code extending the bootstrapping interpreter (that included trait parsing and semantic analysis and conflict resolution of traits). We believe this cost is low enough in comparison with applying this change modifying the virtual machine or making a bootstrap in a staged way. A staged process would generate several intermediate (and bug prone) versions of the code until reaching the last version. Sections 7 and 8 discuss further the cost of implementing our infrastructure and extending it.

6. First-class Runtime MOP

The bootstrapping interpreter works by interpreting the language definition ASTs and applying its effects into a language runtime. For this, the bootstrapping interpreter uses a first-class representation of the bootstrapped runtime, namely an object space. An object space is a *meta-object* that eases the bootstrap manipulations through its meta-object protocol (MOP) [KdRB91]. This MOP is based on our previous work on the object space model [PDFB13]. Following, we present the basic operations of the object space MOP that we use for bootstrapping divided into three main categories: code loading MOP, instance manipulation MOP and VM-language configuration MOP. Object spaces encapsulate particular VM details such as class or object format, making our bootstrapping infrastructure agnostic of such details.

6.1 Code Loading MOP

The bootstrap must support the installation of new code. In our context of object-oriented applications, this implies the proper installation (and deinstallation) of classes and methods.

create class *<name>*, *<spec>*. It creates a class named *<name>* whose instances will follow the specification

<spec> i.e., their type and number of slots. It returns a meta-object of the newly created class.

get class by name *<name>*. It returns the meta-object of the class named *<name>*.

compile method *<source code>*. It creates a new method by compiling *<source code>*. Names (*e.g.,* class names, globals) inside *<source code>* are linked to the corresponding objects and classes inside the virtualized application. It returns the meta-object os the newly created method.

install method *<class>, <method>*. It installs *<method>* as part of *<class>*. This operation makes available this method to the rest of the code in the virtualized application.

swap identity *<old object>, <new object>*. It replaces all references to *<old object>* by references to *<new object>*. This operation is important to replace stub objects once the full class is created.

6.2   Instance Manipulation MOP

Besides basic primitives to create classes and methods, the bootstrap will create and configure normal objects. The following MOP provides the basic operations for such a manipulation.

create instance *<class>*. Creates an instance of *<class>* *i.e.,* an instance that conforms to the spec of *<class>*, containing the number and type of slots described in it. It returns a meta-object of the newly created instance.

instances of *<class>*. It returns a list of the instances of *<class>*.

get class of *<object>*. It returns the meta-object that corresponds to the class of *<object>*.

set class *<object>, <class>*. It changes the class of *<object>* to *<class>*, if both classes have the same spec.

get slot *<object>, <slot name>*. It returns the meta-object that corresponds to the object referenced by the slot named *<slot name>* of *<object>*.

set slot *<object>, <slot name>, <new value>*. It sets the slot named *<slot name>* of *<object>* to *<new value>*.

6.3   VM-Language Configuration MOP

The VM and language are tightly coupled in order to execute code. We can generalize this relationship as a set of the well-known objects of the language that the VM requires at runtime. Examples of such well-known objects are the boolean objects true and false required to evaluate boolean expressions, or the Array class that may be used internally by the VM. To apply a VM with the newly-created objects, we include in our MOP two basic operations that allows us to modify the interface between the language and the VM.

get special object *<name>*. It returns a meta-object that corresponds to *<name>*. This operation enables the introspection of the current VM-language configuration.

set special object *<name> <object>*. It replaces the object at *<name>* by *<object>*. This operation enables the reconfiguration of the VM-language interface.

7.   Implementation

We implemented our bootstrapping infrastructure on top of Espell, a language runtime virtualization infrastructure for the Pharo language implementing object spaces [PDFB13]. Object spaces are isolate-like applications as in the Multi-tasking Virtual Machine [CD01]. A hypervisor object space can fully manipulate other object spaces through a special library. With this infrastructure, two independent Pharo applications run on top of the same VM: the bootstrapping interpreter inside the hypervisor and the bootstrapped application inside a normal object space. Figure 8 shows an overview of the implementation.
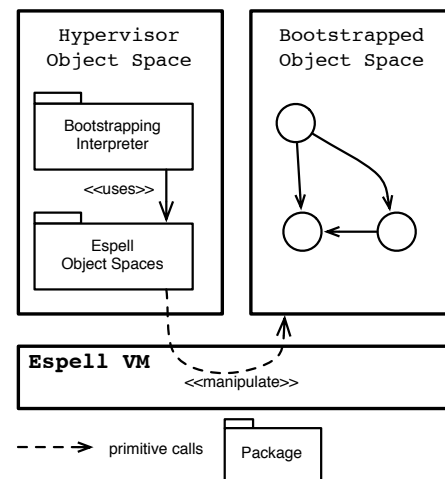

Figure 8. Implementation Overview. Two isolated object spaces run on top of the same Espell VM. The Espell Object Spaces library allows the hypervisor to manipulate the virtualized application. The bootstrapping interpreter inside the hypervisor.

Our Espell implementation comprises some modifications on Pharo Stack VM and one Smalltalk library (object spaces library). The VM modifications provide support to run several object spaces on top of the same VM (and having for example an initially empty object space). These changes include the primitives to resume an object space's execution and patch some primitives such as the ones that iterate over the heap (to only iterate over the correct object space). Espell's object space library implements the object space meta-object and its MOP. Espell object spaces manipulate the bootstrapped runtime through VM primitives. Most of the primitives we use were already existing in the VM. Table 1 offers an overview of the effort of this implementation

measuring its number of lines of code. Implementing Espell supposed a one-time effort of 461 lines of code extending the existing VM and a bit more of 4700 lines of Smalltalk code for the object space library.

Component	Lines of code
Espell VM	461
Espell Library	4735
AST Interpreter*	3422
Bootstrapping Interpreter	455

Table 1. Implementation Effort. Implementation effort of our solution measured in lines of code. All of them are one-time efforts. (*) The AST interpreter is an already existing Pharo library. Thus, we didn't developed it, we just imported it and subclassed it with our bootstrap interpreter. This numbers do not count parsing for AST generation.

All these are one-time implementation efforts. Espell's implementation is portable between different operating systems as they are completely written in Slang, a Smalltalk subset used to build Pharo's VM, so not platform specific code is used. The bootstrapping interpreter extends the already existing AST interpreter of Pharo adding object space interaction. During our evaluation 8 we discuss further the cost of doing a particular bootstrap on this infrastructure.

8. Evaluation

In this section we present our results while bootstrapping three different Pharo-like case study languages. We consider these three languages Pharo-like as they share the same VM execution semantics. However, they present different object-models for the developer. We first present each of our case studies briefly. Then, we present the effort of bootstrapping each of them on our infrastructure and finally some measurements: startup time of the bootstrapped language and the time it took to bootstrap. Each of the measurements we present below were made on a 2.2 Ghz Intel Core i7 machine with memory 8 Gb 1333 Mhz DDR3.

8.1 Case Studies

To evaluate our solution we bootstrapped three Pharo-like languages with our solution. These three languages share a Smalltalk syntax to reuse the parsing and AST infrastructure. Although these similarities, each of the three language kernels possess different object models and reflective models: traits [SDNB03], first-class slots and object layouts [VBLN11] and mirror based reflection [BU04]. Figure 9 illustrates the concepts in each of these languages and their relationships.

Candle. Candle is a Smalltalk-based language with a micro language kernel. Its class model includes class based single inheritance with implicit metaclasses, as Smalltalk-80

and Pharo. We built Candle's language kernel by adapting MicroSqueak [Mal] to run on top of the Pharo VM. This micro language kernel was designed with the explicit goal of being the minimal distribution for the Squeak Smalltalk language.

MetaTalk. Metatalk [PBD+11] is a reflective language where reflection is fully decomposed in explicit meta-objects, namely mirrors [BU04]. Metatalk makes the usage of reflection explicit: a program's execution takes place in the base-level of the language kernel, and it jumps to a meta-level when a mirror is used. Metatalk class model is simpler than Smalltalk's class model. It does not impose meta-classes. Instead, all classes are instances of the single Class class. If there is a need for metaclasses (to share behaviour between classes), the developer can write its own explicit metaclasses.

Metatalk decomposes reflective behaviour as well as the language meta-information *i.e.,* class' names, field order and names amongst others are part of its mirrors, and thus, they belong to the meta-level. When there is not a need for reflection, a Metatalk program can discard its meta-level with all the meta-information in it. This decomposition allows us to bootstrap Metatalk with or without its meta-level. This results in two different language kernels: Metatalk base-level has no reflection at all, while Metatalk with both the base and the meta level is a fully-reflective language.

Pharo. Pharo [BDN+09] is an object-oriented reflective Smalltalk-inspired programming language. As it is a Smalltalk-80 inspired language, its class model includes implicit metaclasses. Pharo provides also circularly defined traits [SDNB03] and class extensions (*i.e.,* the ability to add methods to a class that belongs to another package). Pharo includes also first-class instance variables (namely slots) that provide a MOP to alter the meaning of instance variable reading and writing [VBLN11]. The current version of Pharo defines slots in their classes (meaning that we should create them during the bootstrap) but are not used circularly yet to specialize behavior.

Pharo is a fully-reflective language, placed at the end of the reflective spectrum. The Pharo language includes introspection in the kernel itself, and also self-modification stratified in three levels: object mutation facilities, a class builder and a compiler. The main challenge in Pharo is that the kernel itself of Pharo is defined by Traits: *e.g.,* the Trait class uses a Trait. First-class slots also add to the self-description of the language. This introduces new circular definitions to be expressed in the bootstrapping interpreter.

8.2 Bootstrapping Effort

Bootstrapping each of these languages had a main effort of creating the language definition. However, most of the code of these definitions was given. The missing code parts that we were not available were the creation of initial objects
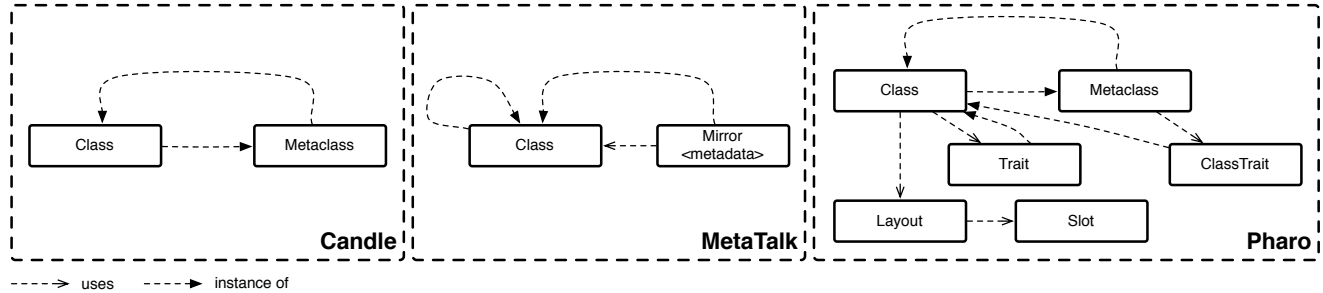
Figure 9. Simplified object model schemas. Schema illustrating the concepts of Candle, MetaTalk and Pharo.

(first objects and classes), the *initialization*[7] order of classes and some adaptations to the particular VM implementation. We introduced each of the missing elements manually in the language definition. Table 2 shows this effort measured in lines of code.

Table 2 also presents the cost in lines of code (LOC) of the extensions we applied. Implementing Traits required on one hand modifying Candle's language definition and on the other hand extending the interpreter. Pharo already included a mature Trait implementation that we did not have to modify. The trait interpreter extension is also a one-time cost and it can be used for both Candle and Pharo bootstraps.

Component	Definition	Traits
Candle	8984*	30
MetaTalk	1274*	n/a
Pharo	92635*	0
Interpreter Extensions	n/a	215

Table 2. Cost in LOC of bootstrapping and extending our case studies. The cost of creating the language definition and extending it. The interpreter extensions are a one-time effort that can be reused by all the bootstraps. (*) The major part of the language definitions were provided.

8.3 Startup Time

From a user point of view a language runtime bootstrap is transparent within the startup of an application. It should be however fast and ensure always the same initial state. To achieve these properties, our solution *caches* the result of the bootstrap process in a snapshot. The bootstrap process is only re-executed when we change the language, otherwise we load the cached version. Caching keeps both properties of application startup: it guarantees the same state and it is fast. In this section we discuss the startup time using our cache

---

[7] The initialization order is different from the bootstrap order. The initialization order is the order in which classes should be initialized. For example, Character should be initialized before String. Our solution does not force the language designer to worry about the order in which the object representing the class of the runtime are created.

technique. The bootstrap time is discussed and evaluated further in Section 8.4.

Table 3 shows a comparison in the startup time of our VM loading our different languages using snapshots. Due to the absence of a Pharo VM-based bootstrap, we contrast our solution with Ruby. We measured the startup times by taking the mean of running each of them 10 times. From the results, we observe our startup time is bigger than Ruby's one but still reasonable, under the third of a second. We believe the difference resides however in the startup of other VM components and is independent of the language runtime bootstrap.

Language	Startup time	
Ruby	64ms	+/-7.1
Pharo	280.8ms	+/-3.4
Candle	186ms	+/-7.6
Metatalk w/o mirrors	202ms	+/-13
Metatalk reflective	205ms	+/-11

Table 3. Startup time in perspective. Comparing the startup time of a ruby application with the same in Pharo, Candle and MetaTalk using a snapshot.

Implementation-wise, the snapshot we used is a memory dump of the VM heap. This heap contains all the objects, classes and methods we created during the bootstrap. At load-time, the memory dump is restored into memory and the VM internals are re-configured to use this heap using the VM setup interface (Section 6). This idea is the same used by languages such as Smalltalk, Lisp, Javascript in V8 or the JikesRVM [AAB+00]. Loading a binary image is as fast as reading the file and putting its contents inside the VM's heap.

8.4 Bootstrap process time

The bootstrap process time depends on the size and complexity of the bootstrapped language runtime. Table 4 presents the code size of each of these language bootstraps in terms of their code entities (classes, traits, mirrors) and methods. We can observe that Pharo presents a lack of mod-

ularity of the language that impacts in the amount of code elements we have to build. Pharo is historically a monolithic system which precludes us to build a minimal system by the moment of writing this paper. In fact, the Pharo language we are bootstrapping represents a subset of the full Pharo language as it is distributed.

Language	Code entities	Methods
Candle	100*	875
Metatalk w/o mirrors	25	114
Metatalk reflective	58*	166
Pharo	626*	6812

Table 4. Language Code Size. Amount of code entities and methods in each of the case study languages.(*) Pharo and Candle have implicit metaclasses, meaning that for each created class, an associated metaclass is created even if not necessary. Metatalk introduces a mirror object for each of the classes in the language.

Indeed, larger and more complex bootstrap processes, as in the case of Pharo, lead to slower bootstrap times. Fortunately our snapshot cache strategy avoids paying the bootstrap process time at each startup. Indeed, a normal user only pays the startup time that we measured in the previous section (cf. Section 8.3). We measured our bootstrap times using an unoptimised AST interpreter in Table 5. This time comprehends the entire bootstrap process: from parsing the code in the language definition to its complete setup. We executed each of these benchmarks 10 times.

Language	Bootstrap time	
Candle	86.75s	+/-8
Metatalk w/o mirrors	0.96s	+/-0.11
Metatalk reflective	13.7s	+/-0.06
Pharo	2h30m	+/-10m

Table 5. Building Benchmarks. Comparing the execution time of the bootstrapped languages using AST interpretation and partial evaluation. (*) Pharo and Candle have implicit metaclasses, meaning that for each created class, an associated metaclass is created even if not necessary. Metatalk introduces a mirror object for each of the classes in the language.

We can observe from our measurements that bootstrapping Metatalk takes in average 1 second if no mirrors are built and 13 in the reflective Metatalk case. Candle bootstrap is slower, in the order of 1 minute and a half, mainly because it contains eight times more methods than Metatalk. These are however acceptable times for debugging. The worst case is Pharo, where creating a class is an operation that takes in average 17 seconds.

Optimizing the bootstrap process time is only necessary for debugging purposes. The final user of the language will use a cached version of the system and will perceive no difference. However, optimizing the bootstrap is a challenging task: since the main purpose of the bootstrap process is to easily change part of the semantics and structure of the language entities we cannot fix them statically to optimize them. In exchange, we chose to optimize the interpretation cycle using a dynamic bytecode compiler that compiles the interpreted code on demand. This compiled code is cached and executed directly on the VM bypassing the interpretation step in following executions. We implemented dynamic compilation to optimize Pharo as it presents the worse of our results (cf. Table 6). We reduced the total bootstrap time by a factor of 2.85. We observed a mayor improvement on class creation, where the time improves from 17 to less than half a second. Please notice that the current implementation only optimizes class creation. There is still room for improvements since we did not optimize method compilation nor parsing. Our plans are to improve the method execution speed since the bootstrap should be part of the future Pharo release.

	Interpreter		Compiled		Factor
Total	2h39m	+/-10m	52m38s	+/-3m39s	2.85x
One class	17s	+/-1	0.4s	+/-0.2	39.85x

Table 6. Comparison of bootstrap time in absence and presence of dynamic compilation in Pharo.

9. Related Work

We present three different categories of related work. First, we present a similar approach to bootstrap in Common Lisp's bootstrap. Second, we present some high-level low-level programming scenarios that aim to solve similar problems than ours, focusing on the VM side. Finally, we present approaches similar to object-spaces to clarify the VM-language interface and allow its manipulation from an external entity.

9.1 Reflection and Open Object Models

Reflective systems and languages provide support for accessing to a program's representation and change it at runtime [Smi84]. To enable reflection, mainstream languages such as Java (in some degree), Ruby or JavaScript introduce *causal connections i.e.,* the programming language incorporates structures that represents aspects of itself (*e.g.,* classes, objects), in such a way that if one structure changes the aspect it represents is updated accordingly, and vice-versa [Mae87]. This introduces a circular reference between the reflective and the base layers of the language with a risk of infinite meta-recursions [CKL96]. Indeed, when the meta-level tries to change some code that it uses at the same time. Denker et al. partially solve this problem in Reflectivity [DSD08], a reflective framework that avoids meta-

recursions by tracking reflective calls and control when reflective features are activated. Reflectivity succeeds to modify and scope behavioral changes for different meta-levels inside the same reflective architecture. However, it does not provide with support for structural changes of the code the reflective framework depends upon, such as changing the structure of classes or removing methods and classes.

Piumarta et al. [PW06] present an open extensible object model that can be used to define the JavaScript and self prototype models and traits. While this object model is simple enough, it is meant to design a language from scratch. The problem we face in the current paper is different: we want to extend an already existing language. In such way, we can benefit from existing libraries and the existing VM implementation. Such extensible object model is used to define COLAs [Piu06]. In this white paper, Piumarta describes the infrastructure to build and circularly define a language using such object model. This infrastructure is based on the idea of an open system where no part is hidden (and can thus be changed) such as in our solution. Regarding the complexity, we believe our solution offers good results and is much simpler to implement than Piumarta's, which requires the implementation of two circularly defined languages (a coke and a pepsi).

9.2 Common Lisp Bootstraps

Within the approaches for bootstrapping reflective languages we can find the Common Lisp bootstrap [Rho08, Str14]. They describe their approach for generating a new virtual machine and image for Lisp:

1. A cross compiler is installed inside the host environment.

2. The cross compiler generates Lisp object files by using a special namespace, isolated from the host.

3. Those files are then loaded into a byte stream representing the memory layout of a Lisp image.

4. Once the image is built, the virtual machine loads and initialises it.

Rhodes does not discuss the challenges of a bootstrap process, nor many of the problems it solves besides the self-description of the system. The article does not clearly answer the problems we encounter. It does not show that the process can be applied to other languages.

9.3 Metacircular VMs

VMs have an impact as well in language kernel evolution: the VM defines and enforces the language's execution model. Not so surprisingly, evolving a VM presents usually the same problems as evolving a language kernel: high amounts of low-level code which leads to lack of abstraction, and mixed and scattered concerns. *Metacircular VMs* propose the use of high-level low-level programming for VM evolution [FBC+09]. A metacircular VM is a VM written in the same language it provides in the end. Such VMs use high-level language VM frameworks to provide the VM developer with prefabricated components *e.g.,* it is possible to simply parametrize a premade GC to reduce development effort. The main goal of this research is to shield the VM developer from complexity and irrelevant detail, and so, improve his efficiency.

Several Metacircular VM projects were developed in the last years. Maxine [WHVDV+13] and Jikes [AAB+00] are Metacircular VMs for the Java programming language. Pinocchio [VBLN11] and the Squeak VM [IKM+97] present efforts in the Smalltalk side. Klein [USA05] does the same for the Self programming language with a particularity: there is no separation between VM and language. PyPy [RP06] is a Python-based high-level VM framework that is mainly used to develop a Python VM, however it has been successfully used to build VMs for other languages such as Smalltalk [BLM+08].

All the works above succeed to provide the developer with low-level abstractions written in a high-level language, with the exception of some explicit low-level code for its startup/*bootstrap*. By using high-level languages to describe a VM, we can build better abstractions and tools to work on VM related concerns. For example, PyPy[RP06] presents already implemented garbage collectors (GCs) and Just In Time (JIT) compilers in complete isolation to the code interpreter. The Squeak VM provides with a VM simulator that allows one to debug the VM code with the tools meant for Squeak itself. Still such works do not focus on bootstrapping a language runtime but the VM runtime, focus more on the VM execution part (GC, speed, JIT), and are thus complementary.

9.4 Clarifying the VM and Language interface

Regarding a clear separation between VM and language, we find in the JVM Tool Interface (JVMTI) [JVM] and the Klein VM [USA05] two approaches that are related to the object spaces first-class runtime.

JVMTI [JVM] is the Java VM interface used by development and monitoring tools that appeared as an evolution of the Java Debug Interface (JDI). JVMTI provides a programming interface to manipulate the VM internals through functions: memory management, thread control and manipulation, stack frame and heap access, object and class manipulation. It succeeds to be used with debugging, dynamic analysis, code coverage or profiling purposes. Regarding bootstrapping, the JVMTI has not been used, to our knowledge, to manipulate the Java language kernel at initialisation time (or any of the other languages on top of it).

The Klein VM [USA05], already named amongst the metacircular VMs, exposes mirror objects to enable object manipulation. It exposes as well the VM internals to the language through low-level mirrors. These low level mirrors make explicit the interface between the language and the VM elements. Particularly, Klein explores the usage of low-

level mirrors in the context of debugging a remote Virtual Machine.

10. Discussion and Future Work

10.1 Infrastructure requirements

Our solution indeed makes Pharo and languages alike easier to modify. Our bootstrapping infrastructure solves the bootstrapping circular problems and provides two main points to extend a language. This flexibility comes however at the cost of a more complex infrastructure: the introduction of a bootstrapping interpreter and a first-class runtime.

Regardless of the simplicity of building an AST interpreter, it means more code to maintain for the language developers. The ideas behind Metacircular VMs [WHVDV+13, AAB+00, VBLN11, IKM+97] could reduce this burden by making reusable the VM interpreter for bootstrapping and provide a bootstrap-time object space meta-object. Alternatively, the interpretation steps can be replaced by a Just-In-Time compiler such as the one we used for optimizing the Pharo's bootstrapping process.

10.2 Reflection and stub classes

The introduction of stubs in our bootstrap process comes with one main drawback. Using reflective operations on a class is to be avoided before the real class is created. Otherwise the reflective operations will try to act on the class stub and miss-behave. This happens since stub classes do not contain all the reflective data to answer to reflective operations properly. Our implementation fills stubs only with the information necessary for instantiation.

A possible alternative to solve this problem in the future is to detect reflective operations on stubs and apply them transparently on the elements of the language definition. The language definition ASTs have all the information required to answer such operations (*e.g.,* the class' superclass, list of subclasses, names of instance variables, etc.). However, such information is not encoded as objects from the bootstrapped language and would require transformations introducing more bootstrapping issues.

10.3 VM-Runtime Bootstrap

Our solution focuses on the language runtime bootstrap and gives us a degree of freedom to modify our languages. We showed how we can easily add mirror-based reflection, traits, first-class instance variables. It does not, however, allow us to change the language execution semantics, which still resides in the VM (often encoded in bytecode or assembly generation). Changing the VM execution semantics poses a big challenge as it affects many components: the language runtime definition, the interpreter, the JIT compiler, amongst others. We would like to explore in the future the co-evolution of VM and language.

11. Conclusion

Bootstrapping is commonly known by its usage on compiler building, where a compiler can compile itself. A bootstrap process allows us to easily change this system as it is expressed in terms of itself, taking advantage of its abstractions and tools.

This paper explores an infrastructure to ease the circular bootstrap of reflective object-oriented language runtimes from the Pharo family. Our infrastructure is based on a bootstrapping interpreter that solves the bootstrapping issues by transparently introducing class stubs and providing an alternative method lookup. This bootstrapping interpreter manipulates the language runtime under bootstrapping through a first-class runtime. This first-class runtime offers a MOP that presents a clear VM-language separation.

We validated our bootstrap process by bootstrapping three object-oriented languages of the Pharo family. These three languages have key differences between them: Candle is a minimal Smalltalk with implicit metaclasses, the core of the Pharo language is defined by traits and first-class slots, and Metatalk decomposes reflection from the base-level and stores meta-information in the meta-level of the language. These three languages run on top of the same Pharo Virtual Machine. We showed also that a fast startup can still be achieved through caching.

References

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BLDP10] Tristan Bourgois, Jannik Laval, Stéphane Ducasse, and Damien Pollet. Bloc: a trait-based collections library - a preliminary experience report. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'10)*, Barcelona, Spain, 2010.

[BLM+08] Carl Friedrich Bolz, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. Back to the future in one week — implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*, volume 5142 of *LNCS*, pages 123–139. Springer, 2008.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA'04,*

ACM SIGPLAN Notices, pages 331–344, New York, NY, USA, 2004. ACM Press.

[CD01] G. Czajkowski and L. Daynés. Multitasking without comprimise: a virtual machine evolution. *ACM SIGPLAN Notices*, 36(11):125–138, 2001.

[CKL96] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

[DSD08] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008.

[FBC+09] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, Eliot, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In *VEE'09, ACM SIGPLAN/SIGOPS*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[IKM+97] 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 *OOPSLA'97, ACM SIGPLAN*, pages 318–326. ACM Press, November 1997.

[JVM] Sun microsystems, inc. JVM tool interface (JVMTI). http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA '98*, pages 36–44, 1998.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[Mal] John Maloney. Microsqueak. http://web.media.mit.edu/ jmaloney/microsqueak/.

[PBD+11] Nikolaos Papoulias, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Towards structural decomposition of reflection with mirrors. In *Proceedings of International Workshop on Smalltalk Technologies (IWST'11)*, Edingburgh, United Kingdom, 2011.

[PDFB13] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. Virtual smalltalk im-

ages: Model and applications. In *IWST - International Workshop on Smalltalk Technology, Colocated within the 21th International Smalltalk Conference - 2013*, 2013.

[Piu06] Ian Piumarta. Accessible language-based environments of recursive theories (a white paper advocating widespread unreasonable behavior). Technical report, Viewpoints Research Institute, 2006. VPRI Research Note RN-2006-001-a.

[PW06] Ian Piumarta and Alessandro Warth. Open reusable object models. Technical report, Viewpoints Research Institute, 2006. VPRI Research Note RN-2006-003-a.

[Rho08] Christophe Rhodes. Sbcl: A sanely-bootstrappable common lisp. In *International Workshop on Self Sustainable Systems (S3)*, pages 74–86, 2008.

[RP06] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *DLS'06, companion to OOPSLA'06*, pages 944–953, New York, NY, USA, 2006. ACM.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *POPL '84*, pages 23–3, 1984.

[Str14] Robert Strandh. Resolving metastability issues during bootstrapping. In *International Lisp Conference*, 2014.

[USA05] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05, ACM SIGPLAN*, pages 11–20, New York, NY, USA, 2005. ACM.

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, pages 959–972, New York, NY, USA, 2011. ACM.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.