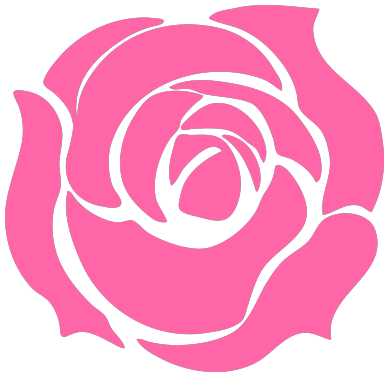


Utena VM
Virtual Machine for a dynamic-interactive object system

Gnuxie Lulamoon
Gnuxie@applied-langua.ge

June 18, 2021

Contents

Chapter 1

Introduction

1.1 Project Overview

The aim of this project was to prototype a Virtual Machine (VM) that allows for dynamic loading of objects that would be capable of hosting a dynamic-interactive object system, the purpose of this is to explore and learn about prototyping a VM with new semantics (that are interesting to us) in an allotted timescale.

The semantics we are interested in are object transportation and the Object Capability Model, however the project needed to be achievable in the time frame and we are only providing a framework (in terms of approach and the implementation of the virtual machine itself) that could be used to explore these ideas.

I had no previous experience with implementing virtual machines before I started this project or any real knowledge of how they work other than word of mouth (Which granted can actually be very powerful) so this was a learning exercise for me as well.

The prototype is implemented as a series of Common Lisp systems due to the support the Common Lisp ecosystem has for rapid prototyping, developer interaction and the ability to reuse complex components from the hosting implementation such as the garbage collector when developing new languages and their runtimes.

1.2 Project Aims

- Investigate existing dynamic-interactive object systems and virtual machines.
- Design a virtual machine by selecting an initial set of opcodes and primitive objects.
- Implement an adaptable prototype of the virtual machine.
- Evaluate the design and implementation of the virtual machine.

While these aims are somewhat ambiguous I must remind the reader that while we analyze and even criticise a number of technologies in order to inform the design of the VM and cover a lot of ground in the following report, ultimately we are developing a VM for the first time and making it interesting, having fun, not solving every problem that is discussed or exists.

Chapter 2

Background Research

2.1 Virtual Machines

We identify a number of desirable properties for our VM particularly in the context of allowing methods to be transportable. The following properties must be understood in order to understand the analysis that follows. They are enumerated only so they can be referenced, they are all required, there is no precedence.

1. byte-code operations should work to enable the message send and any additional operations should only be concerned with behaviour that is impractical to develop via primitive objects.

2. Message transparency: Any machine byte-code concerning the implementation of the message send must not depend on the presence of any primitive object in the machine.

3. Fully transparent proxies: It should be possible to proxy any object transparently and any means of detecting a proxy in a guest must be encapsulated with mirrors which could also be proxies.

2.1.1 Java Virtual Machine

The JVM is the most universally supported virtual machine with oracle famously claiming that it runs on '3 billion devices'. The object model in the JVM makes a distinction between methods and fields [ora, 2015, Section 4] and access to either is a distinct operation. This leaks the implementation of Java objects to both business methods and internal methods in the byte-code form, which is something we want to eliminate for message transparency. Byte-code in the JVM also contains information about the types that are being used, including primitives, which can make it impossible to provide proxies to in place of expected arguments to a given object. This is particularly important in the context of transporting byte-code.

Work is underway to add support for cooperative multi-threading to the JVM in the form of delimited continuations and fibres [Pressler, 2019]. This is a good model for threads and asynchronous programming as it allows for programs to be written in direct style. The only cause for concern is how Java's exception system interacts with fibres and the scheduler [Pressler, 2020]. This is a model for concurrent & asynchronous operations we wish to replicate should we consider adding threads to the VM.

2.1.2 Termite Scheme

Termite scheme [Germain, 2006] is a variant of scheme providing support for distributed computing, inspired heavily by Erlang [Armstrong, 2003]. What is interesting about Termite scheme is that it is able to serialise any object for transport and any objects that cannot be serialised are replaced with a proxy [Germain, 2006]. Both Erlang and Termite Scheme emphasise immutability in both programs and primitives which makes transporting them significantly easier as it reduces the number of cases were identity needs to be preserved.

2.1.3 JavaScript

JavaScript implementations also use a similar prototype based object system with the 'map' storage layout from the Self VM, however some implementations have also introduced a 'transition tree' to reduce the amount of 'map splitting' that would occur when adding the same slots to objects that share the same map [Artoul, 2015].

2.1.4 Smalltalk

While Smalltalk has been largely superseded by the other environments we are discussing, it has provided a number of resources to help with designing and implementing a virtual machine with message oriented object systems. A particular example is the *Blue Book* [Goldberg and Robson, 1983], which details not only the Smalltalk-80 System but also how to create an implementation and 'A little Smalltalk' [Budd, 1987] which details how to create a *LittleSmalltalk* system.

2.1.5 Newspeak

In Newspeak, all names are late bound and there is no global environment. Modules are defined as top level classes of which there can be many instances with different versions of their dependencies. The dependencies of modules are provided to a *factory method* on the class in order to instantiate the module [Bracha et al., 2010]. This model allows dependencies and capabilities to managed transparently, removing a shared global environment and any ability to depend upon it from the language. This is a model of modularity we would like to see from any guest.

Newspeak also has support for the actor model of concurrency and its' implementation is influenced heavily be the E programming language [Botev, 2012]. This model provided by E relies heavily on a distinction between an immediate and eventual message-send, providing special support for the latter [Miller, 2006].

2.1.6 Self Virtual Machine

The Self VM is referred to as an implementation of the Self language and is not a specification for a Virtual Machine and object system in its own right. To represent Self's prototype object model, the Self VM has a model using *maps* for instance vectors [Chambers et al., 1989]. Maps are used to match messages to instance vector locations, replacing the role of a class in a traditional Smalltalk virtual machine. An interesting property of this map model is that it can be used to express both prototype and class based object systems relatively efficiently, both in terms of performance and development time [Wolczko et al., 1999].

In the Self VM bytecode objects contain a literal array, the indexes of which are referred to by the instruction operands. In the Self language, these literals are constructed at parse time by evaluating slot initialisation forms within the context of an object called *the lobby* [Allen et al., 2017], the lobby here is somewhat similar to an instance of a *top level class* in Newspeak [Bracha et al., 2010] and provides the only environment that that an object can access other than the enclosing lexical environment.

In the Self VM, all messages are looked up dynamically by using message names from the literal array provided with code objects, this makes it possible to create transparent proxies.

2.2 Development Environment

2.2.1 Systems Language

Typically systems programming languages are used to create a VM as they can produce high performance applications by following the semantics of conventional hardware.

However, it is notoriously difficult to write and debug complex programs in systems languages, particularly within the context of creating virtual machines for dynamic object systems [Ingalls et al., 1997, Wolczko et al., 1999, Miranda et al., 2018].

As we are creating an experimental VM and performance is not of concern, there does not appear to be any benefit from using such a language for developing the VM and it would hinder the rapid changes that we need to make to the system as we begin to gain more understanding about the domain. Even though we have considered an argument in favour of better performance, it must be stated this argument remains unconvincing, especially when faced with the concept of meta-circular implementations (See 2.2.2) bringing their own compilers written entirely in a guest language.

2.2.2 Meta-circular implementation

If one has already designed a guest language for the virtual machine, then this language can be used to implement the virtual machine itself by using a meta-circular compiler. Java has had the most success with meta-circular implementations, including Maxine [Wimmer et al., 2013] and GraalVM [Würthinger et al., 2013], followed by Smalltalk [Béra, 2017]. There are at least two other instances where this technique has been explored for other environments (Klein [Ungar et al., 2005] and more recently SICL [Strandh, 2013]).

As this is an experimental system, we don't know what the guest language will look like yet, so we cannot create a meta-circular implementation. However if we did, we think this would be the best approach as not only are all of the components that are expressed in the guest language are portable and can be re-used by other implementations. It would also be possible to change all behaviour of an implementation from within it.

2.2.3 Existing Dynamic-Interactive System

Using an existing dynamic system would eliminate a number of bugs and safety pitfalls associated with systems programming languages and reduce the complexity of the prototype as a number of components from the system can be reused such as the garbage collector. Using an existing dynamic system also means we can use tools and workflows that are native to a introspective system to aid rapid incremental development such as live coding and interactive debugging [Pitman, 1994, Miranda et al., 2018].

2.3 Development Process

The development process needs to integrate closely with the aims of the project and the technologies involved. any strategy needs to consider that are objectives are ambiguous and it is not clear how the aims are going to be achieved without investigation and experimentation.

2.3.1 Rapid-Prototyping

Rapid-Prototyping is a methodology that fits well for experimental projects with ambiguous requirements. With rapid prototyping it is possible to learn about our objectives through experimentation and incremental improvements [Taylor and Standish, 1982] without having to commit resources to a design that may be flawed or unfit for requirements that are not well understood. While it is useful to think with prototypes, there are some risks associated with rapid prototyping to be aware of that can lead to premature the abandonment of a prototype, including loss of maintainability, regression and lack of documentation [Gordon and Bieman, 1995], these are risks that need to be mitigated via a combination of other practices.

2.3.2 Continuous Integration

It is possible to mitigate some of these concerns with continuous integration [Fowler and Foemmel, 2006]. By consistently running tests, building documentation automatically in a central repository with each commit produced by a developer, providing feedback to the developer with notification of failing tests or otherwise, regression and other concerns have more of a chance to be spotted early [Fitzgerald and Stol, 2014].

2.3.3 Continuous Improvement

Having the tools to spot errors or a developing situation can only do so much; the ability to act is as important. Some things cannot be spotted by testing and automation, we need to be constantly aware of the design of the system and its relation to our goals but also we need to be aware of how practice is conducted both in relation to the implementation and every aspect of development [Fitzgerald and Stol, 2014]. Continuous improvement is usually facilitated with methods such as Scrum, Kanban and Keizen each with their own trade-offs that need to be evaluated and chosen by the team.

Chapter 3

Design

3.1 Overview

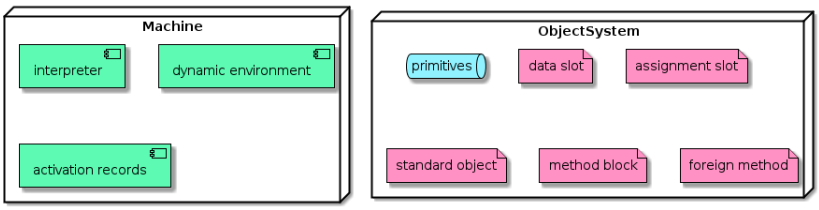


Figure 3.1: Overview of the machine

We begin with an overview of the system which is composed fundamentally of two interdependent components each mediated by a protocol. The machine is concerned with modelling the execution of the object system, how messages are evaluated and the behaviour of methods, whereas the object system is concerned with describing those methods and providing the underlying data structures and representations for different objects.

3.2 Objects & Messages

3.2.1 Objects

To keep inline with the simplicity of Self, we have decided to adopt the prototype-based object model. This has several advantages over a class based model in terms of implementation, mainly that each object is in control of its' own allocation and properties and there is no requirement to calculate the compatibility of super classes and sub classes [Ungar et al., 1991]. It should be noted that the description of objects in our interface (Figure 3.2) is abstract enough that a guest is not tied to using a prototype-based object system, they could sub-class the object interface without needing to use the the translations described by Self includes Smalltalk [Wolczko, 1996].

Fundamentally all that is required for a message oriented object system is a protocol to send messages to an object [Ingalls, 1981]. Therefore, it was decided to separate a message send in our VM into two stages, first a message lookup, which uses a message selector to find a matching slot on an object and then message evaluation, where the machine implementation uses the object and associated slot to produce the behaviour of the message (this distinction is also implied to exist in the Self VM [Allen et al., 2017]).

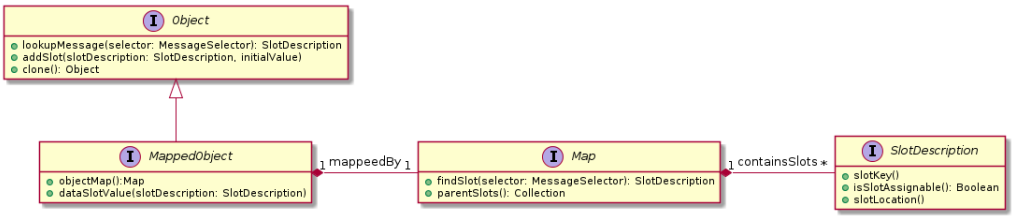
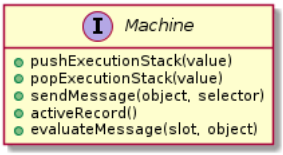


Figure 3.2: UML Class Diagram showing the design for the object interface

This is a useful distinction to make as we can dispatch on the type of a slot received from message-lookup in the machine to control the behaviour of evaluation in a maintainable and extensible way.

3.2.2 Machine




```
(defclass machine () ())

(defgeneric push-execution-stack (machine value))
(defgeneric send-message (object message-key machine &rest arguments)
)
(defgeneric evaluate-message (lookup-value object machine))
(defgeneric pop-execution-stack (machine))
(defgeneric active-activation-object (machine))
```

Figure 3.3: The machine interface

The machine itself is a simple stack machine with only one stack, the execution stack, that the operations described in section 3.3.1 pop and push from for their inputs and outputs. This is intentionally simple, as the addition of any register to the machine allows for a situation where ambient state is left inside the register which can lead to a bug or exploit if an author does not realise that they are leaving state there. The pushing and popping of objects from the execution stack are exposed in the interface 3.3 so that the implementation of the machine's operations can use the stack. We also allow clients to access the current activation record that is on the stack, to make it easier to implement flow control. The machine interface is not only useful to the implementation of the machine's operations, but also situations where primitive methods written in Common Lisp need to send messages to guest objects without relying on the machine implementation, an example of this being the *send-message* generic function.

Slot Evaluation

Now the basics of the machine are understood it is possible for us to continue from the end of 3.2.1 and discuss how messages are evaluated once they have been looked up. To create a new type of slot, all that is required of the slot is to implement the

slot-description interface shown in Figure 3.2 and also define an associated method on the *evaluate-message* generic function shown in Figure 3.3. The implementation of the associated *evaluate-message* method can then use the rest of the machine interface in much the same way as a new operation to create new evaluation semantics. Through the use of these generic functions, it is possible for a user of the machine to add new types of slots and semantics which are not provided or anticipated with the standard object system and the user can modify both of these arbitrarily without needing to directly patch the original source code or access machine internals.

3.3 Operations

Arguably the simplest feature of the Self VM is the way the methods are expressed once they have been parsed, rather than coupling instructions to primitives, registers and other machine specific facilities (which was typical of Smalltalk implementations and still is typical of VMs such as the JVM to date), the representation is concerned only with enabling one fundamental semantic of Self programs, which is a message send to an object [Chambers et al., 1989].

This is something we wish to replicate in our VM as all 'primitive' operations can be implemented as message sends to primitive objects that are capabilities to be requested by a guest environment, essentially removing any *ambient authority* from bytecode, which could allow for the possibility of trustless program transportation in future experiments.

3.3.1 Methods and Instructions

Methods in our VM are represented as an array of instructions and an array of objects (usually literals and symbols that would be read by a parser).

The instructions in our VM are composed of a 4 byte opcode and 12 byte operand that is usually used to index a literal object on a method's literal array.

Please note that some of these instructions are exactly the same as those described in [Chambers et al., 1989].

- Self

  Push the activation record's self onto the execution stack.

- Literal (literal-index)

  Push the literal at the index in the active method's literal array onto the execution stack.

- Send (selector-index)

  Send a message, using the index to get the message selector from the active method's literal array popping the receiver and arguments off the execution stack.

- Implicit Self Send (selector-index)

  Send a message to the current activation record's implicit self, popping the arguments from the execution stack.

- Set Argument Count Register (count)

  Set the argument count register to the count specified, this register used by the machine when a method has a varying number of arguments.

- Activation

  Push the current activation record onto the execution stack.

- Reset

  Pop the target activation record from the execution stack, reset its instruction pointer to 0, unwind the stack until we reach the record and reestablish control to it.

- Escape

  Pop the target activation record from the execution stack, increment its instruction pointer by 1, unwind the stack until we reach the record and reestablish control to it.

3.4 Primitives

We have listed the significant primitives we want to include with the VM, we may provide more or split the primitives into further components in the implementation.

- Block: Blocks are essentially our control flow primitive.
- Boolean: Booleans provide a mechanism to activate a block if a condition is met and so provide conditionals.
- Number: Numbers are usually involved with any useful computation, so need to be included for tests and example programs.
- Method: Methods provide us with a way to define behaviour.
- Object: Objects because this is an object system.
- Symbol: Symbols are used as message selectors.

3.4.1 Boxed primitives

We have decided to box objects (both built-in and objects specific to this VM) available from the host, Common Lisp, for a few reasons. We could have chosen to instead present primitives as a foreign function interface to Common Lisp but this would be a problematic as it would be tying the semantics of our primitives directly to the single host environment we have. We could have also chosen to wrap primitives using the *evaluate-message* generic function in the machine, however this restricts arbitrary modification of primitive objects, which is something we want to allow for the purpose of annotation (for those who are interested in the details of this, please see 6.3.3). So we are left with the need for a mechanism by which all primitive objects can be modified individually much like standard objects, to annotate them with metadata or to add new slots.

3.4.2 Standard Object

We will provide an implementation of the *mapped-object* protocol called *standard-object*to be used as the standard object in the system. The standard object has both an *object map* and an *instance vector* (see Figure 3.4 which extends Figure 3.2 that was discussed

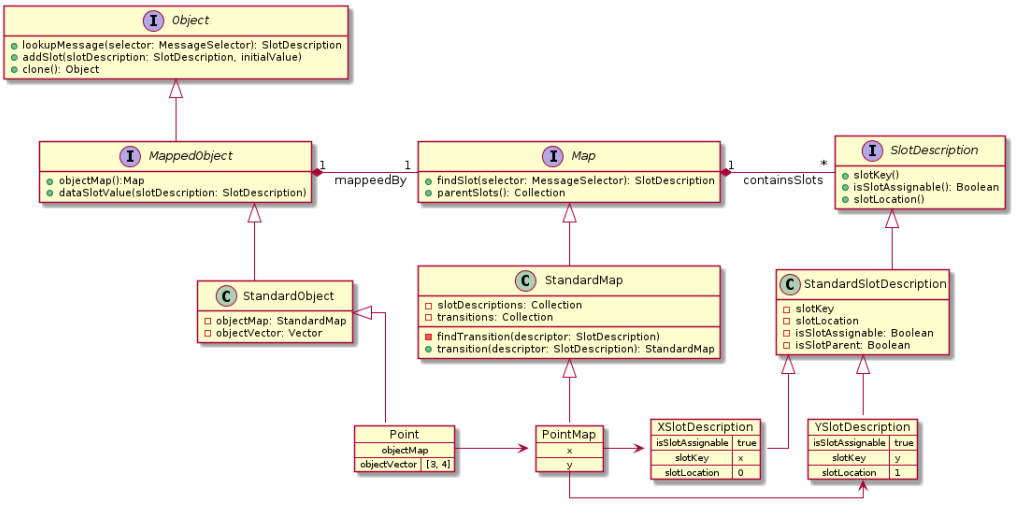


Figure 3.4: An example of a point instance showing the relation to standard object

in section 3.2.1). Figure 3.4 shows us how an assignable slot is allocated on a vector associated with an instance and that a *standard slot description* matches a message selector to a location (index) in the instance vector where the associated value for that slot is stored (e.g. x's location is 0, which in the Point's object vector is 3). This is essentially identical to the object layout that self provides for its objects [Chambers et al., 1989], however we also provide a mechanism to cache the transitions of maps which we discuss in detail in section 5.1.2.

3.4.3 Methods & Blocks

Context

Before the design of methods and blocks can be understood, it is important to understand the context of methods, blocks and activation records in a typical Smalltalk system. Its important to know that a method is a first class object with properties and in a Self style system, the properties of a method actually provide the lexical environment for the method when it is activated. This happens because when a method is called, the method object is cloned (copied) and any argument slots in the clone are instantiated with the respective method arguments [Allen et al., 2017]. A block is mostly synonymous with a *closure* and the closing over of the lexical environment that a block was created within is achieved by adding a reference to the activation record that the block was created within the context of. In Self this is achieved by adding a parent slot to the block object [Allen et al., 2017] in order to include the activation record's properties when performing a message lookup on the block. We include blocks in our system because they are a well established way to add extensible control flow such as conditionals and loops to a Smalltalk system [Goldberg and Robson, 1983] as well as having a small footprint on the number of special cases a guest will need to be aware of.

Method Blocks

The method block is central central component to methods, activation records and blocks. All of these components need arguments, the instructions to execute and the literals that the instructions refer to, and thse are all provided by the method block.

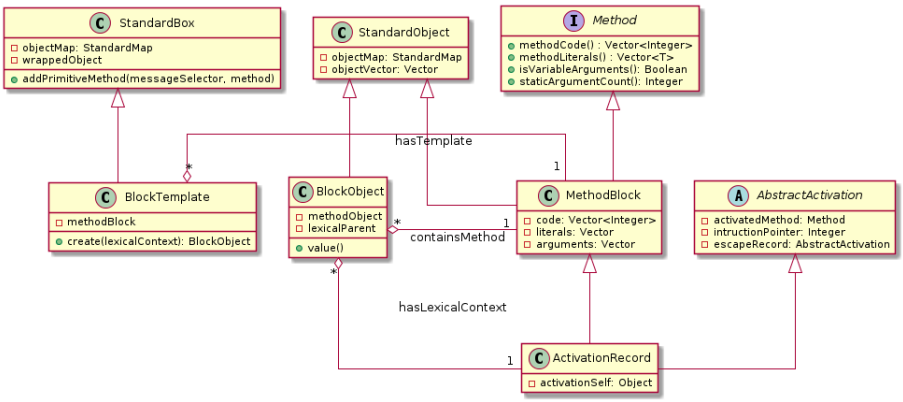


Figure 3.5: UML Class diagram of methods, blocks and method blocks
Note: Some interfaces have been omitted for clarity and the lexicalParent is depicted as being present on the *BlockObject* where in reality it would appear as a parent slot on an instance of the *MethodBlock*, however the class diagram is to helps us understand the situation from the host's perspective, and not the guest's)

In figure 3.5 we see the method block extends the *standard-object* and this is so that the method behaves much the same way to the guest as a standard object, but we also want to store some extra information for the machine to use (such as the instructions and literals). Method arguments will be created such that they are a new type of slot as discussed in 3.2.2 so that they are distinguishable from the other slots on the method object and the machine can initialize them by popping the arguments from the machine execution stack during activation.

3.5 Dynamic Environment

You may recall from section 3.3.1 in our machine that we provide a reset and escape instruction to provide the capability of non local transfer of control. In this section we detail the dynamic environment that is maintained in order to perform bookkeeping of which exit points are still able to receive a valid transfer of control. This is important because we should not allow for a situation where control is transferred to an activation record that is no longer active and has since returned normally to the preceding record (As this would allow for the equivalent of Scheme's call-with-current-continuation in which an implementation needs to take special measures to copy the effective call stack, execution stack and the contents of the dynamic-environment [Shinn et al., 2013]). A dynamic environment is going to be essential for the reasons of performing bookkeeping of exit points that are valid during non local transfer of control [Strandh, 2013] and this is all we aim to use the environment for in our prototype, however it can also be used to add interactive error handling via *conditions* and *handlers* [Herda, 2020].

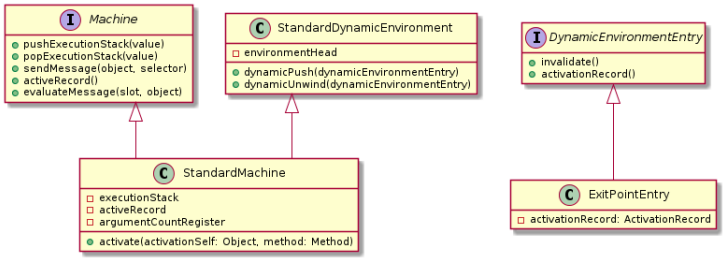


Figure 3.6: UML Class diagram showing the design of the dynamic environment

In order to book-keep which activation records control can be transferred to we only really need one entry type, which is the *exit-point-entry* and this only needs to detail which activation record it will return control to. In order to allow extension later on we provide an interface for the dynamic-environment-entries which only need to be able to tell the machine which activation record was active. We also specify the *invalidate* generic function as part of this protocol so an entry can control what happens when the machine removes the entry from the environment. In figure 3.6 we also provide a *standard-dynamic-environment* class to mix in with the *standard-machine* to provide the functionality for inserting and unwinding entries from the dynamic environment. In abstract this design is heavily inspired by the one present in SICL [Strandh, 2013, Section 29.4] but I must stress that they operate in entirely different circumstances.

3.6 Modules as Objects

In this section we discuss how a guest would implement Newspeak's 'modules as objects' [Bracha et al., 2010] using the Self inspired prototype-based object system provided with this Virtual Machine. This section will require the reader to familiar with the implementation of methods (Section 5.2.4) and the object map (Section 5.1). To add some context, in Self there exists a global environment object called *the lobby*, this is used by the parser to provide the lexical environment that slot initialization blocks run within. The slot initialization blocks in Self are executed at parse-time to construct the objects as they are read and are not retained [Allen et al., 2017, Section 3.1.9]. Therefore in order to align the object system so that we can get the equivalent functionality of top level and nested classes from Newspeak [Bracha et al., 2010], we need to instead construct an *object-template* that retains the initialization blocks, a *map* to use for the object we want to create from the template and a *factory method* to act as the lexical environment for the initialization blocks and which will produce the new object when invoked.

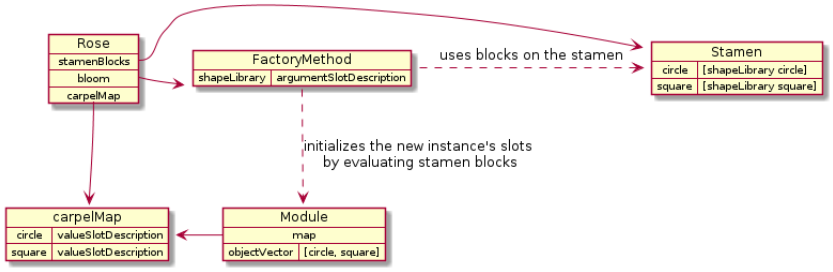


Figure 3.7: Rose with a factory method, templating a module that has a dependency on the circle and square objects from a shape library.

In Figure 3.7 we use the terms *Rose* to refer to an *object-template*, *Stamen* to refer to an object containing the initialization blocks for the data slots on the object we are templating, and *Carpel* to refer to the *object map* that will be used by the templated object when it is allocated. The process for creating an instance from the template starts with calling a factory method on the *object-template*, which then allocates the new object and initializes each *data slot* in the *carpel map* with the result produced by running the associated block in the *stamen* within the context of the factory method.

Chapter 4

Software Engineering Approach and Testing

This section is a discussion of the *implementation* of the processes discussed in 2.3 and the different tools, conventions and practices that were used.

4.1 Style Guide

There are a few style guides relevant to Common Lisp, created by different actors for different purposes. It is important to note that this project is being developed by a single developer, not a corporation and will continue to be maintained by a very small number of developers regardless of outcome, the distinction is important as both have to manage human resources differently [Sockwell, 2021].

We will follow the 'General Common Lisp style guide' found in the SICL Specification [Strandh, 2013, Section 31], as they outline what they consider to be idiomatic and similarly have a small number of core contributors.

4.2 Issue Management & Continuous Improvement

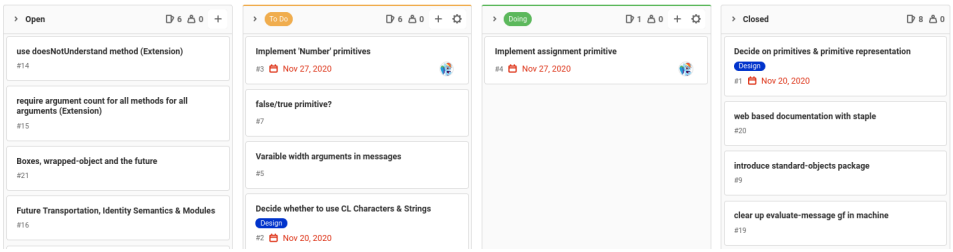


Figure 4.1: The Gitlab Kanban board

Figure 4.1 shows the Kanban board for the project with four states for an issue to lie in, any issues that are found during development are to be documented using the Gitlab issue system and placed on a Kanban board. This will ensure that problems are tracked both as they are live and once they have been dealt with we will retain a historical record of the problem (Whereas for instance TODO notes are frequently deleted and lost in an active code base). Issues should contain some information that indicate when

they can be closed, and if possible refer to any test that should be written and passing before the work can be considered 'complete'.

4.3 Testing

As the goals of the project are extremely dynamic it may seem difficult to take a test driven approach to development, however many components in this project have clear interfaces and as soon as these have been finalized we are able to write a test for their implementation. Therefor once the requirements of a component have been identified in the issue system (4.2) it has been possible to write a test enforcing those requirements and only close the issue when the associated tests have passed.

4.3.1 Continuous Integration

As per the our discussion in the background research 2.3 and recommendations of Fowler & Foemmel [Fowler and Foemmel, 2006], we will be using a central Git repository hosted on Gitlab to run automated tests and provide us with other tools such as an issue system. On each push to the master branch we ensure that all our tests run against the latest commit using Gitlab's CI system. This well notify us of any regression in the code base and eliminate possible errors caused by a desynchronisation between what is present the source control system and the development environment.

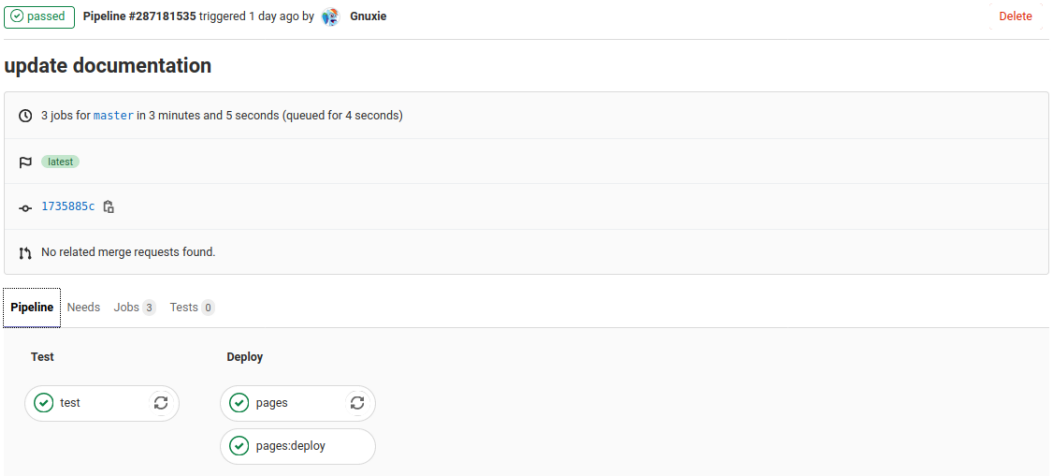


Figure 4.2: The CI pipeline

In addition to running all unit and integration tests, our CI pipeline also updates the statically generated web based documentation from the project components, this not only keeps the documentation up to date but also highlights where documentation is lacking.

4.3.2 Relation to Issue Management

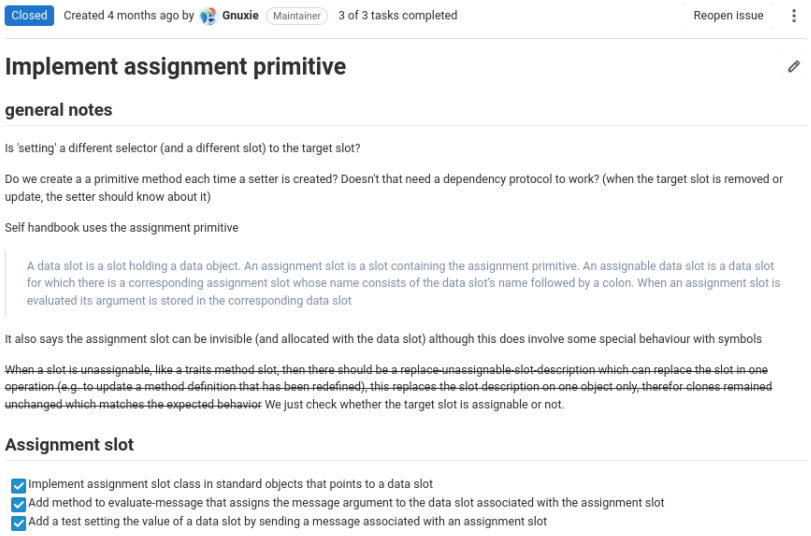


Figure 4.3: A typical issue

The general structure of issues is split into two parts, first we write anything down relevant to research or discussion about the problem and then describe what actions should be taken to resolve the issue. In order to know when an issue has been completed, one of the actions should be to develop a test for the new functionality or prevent regression. When all tests then pass in the CI system for the git commits associated with the work the issue can then be closed.

Chapter 5

Implementation

5.1 The Object Map

5.1.1 Lookup

The lookup protocol is implemented by using an association list of selector-value pairs. In order to find a slot associated with a message selector on a map, each selector of the selector-value pairs in the association list is tested by reference equality to that of the selector we are trying to find an associated slot for. As our object system is similar to that of Self, we implement the same algorithm which is described in the Self handbook [Allen et al., 2017, Section 3.3.8], but to summarise essentially we try to find a selector on an object's map and when a selector cannot be found on the current map, we query the map for any parent slots and then recursively try to find a matching slot on the object map of the parent objects.

5.1.2 Transition

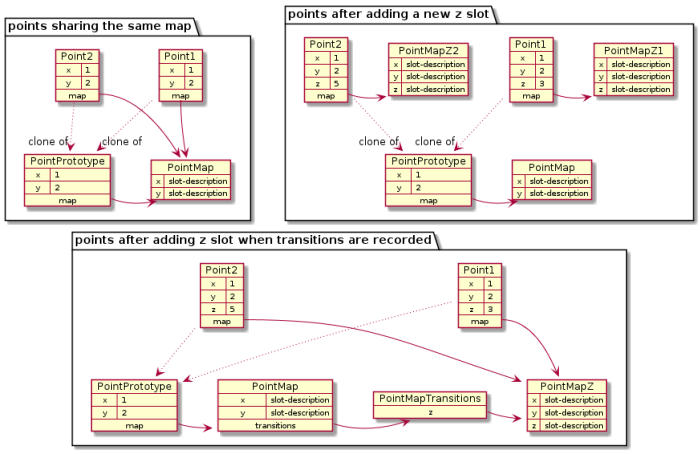


Figure 5.1: Adding a new slot 'z' to two points originating from the same point prototype

Transition is a technique developed by the implementors of Javascript engines in order to avoid the duplication of object maps when the same slots are repeatedly added to objects originating from the same prototype map [Artoul, 2015]. Each map therefor manages an association list of slot-description map pairs recording the next transition states from the current map. When a slot is added to an object, first we look at the

next transition states to see if the description has been added to an object with this map previously, and if so, we can use the associated pair to get the next map. If no previous transition is found, we can just create a new map and add it to the transition tree of the current map.

```
(defmethod find-transition ((transition-node standard-map) descriptor)
    (cdr (assoc descriptor (transitions transition-node))))

(defmethod transition ((map standard-map) descriptor)
    (let ((existing-map (find-transition map descriptor)))
      (or existing-map
          (let ((next-map
                 (make-instance 'standard-map
                                :slot-descriptions
                                (cons descriptor
                                      (slot-descriptions map)))))
            (setf (transitions map)
                  (acons descriptor next-map (transitions map)))
            next-map))))
```

Listing 5.1: Maintenance of the transition tree

5.2 Primitive Boxes

5.2.1 Shared Behavior

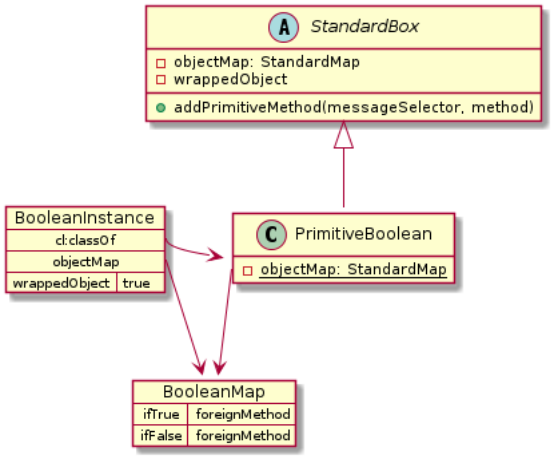


Figure 5.2: The relation between primitive instances and their object map

```
(define-primitive primitive-boolean
    (:wrapped-slot-type boolean
     :wrapped-initform nil))

(defmethod o:box-primitive ((true-value (eql t)))
  (make-instance 'primitive-boolean :wrapped-object true-value))

(defmethod o:box-primitive ((false-value (eql nil)))
  (make-instance 'primitive-boolean :wrapped-object false-value))
```

Listing 5.2: Defining the Boolean primitive

When defining a new primitive type, we want all instances of that type to share the same behaviour and messages. In order to achieve this all we need to happen is for our instances to share the same object map. So, to get shared behaviour between primitives of the same type, we create a subclass of *standard-box* and provide a new definition for the class allocated object-map slot in order to provide an object-map that all instances of our new primitive type will use, however in order to encapsulate that complexity the convenience macro *define-primitive* should be used, as in Listing 5.2.

5.2.2 Boxing

The *wrapped-object* slot detailed in Figure 5.2 and Listing 5.2 is used to encapsulate what is usually an instance of a Common Lisp built in class we want to use to help provide the behaviour of a primitive. This is because it would be impossible to *mix* the class *utena-vm.standard-objects:standard-object* with a Common Lisp *built-in-class*. This technique is known as boxing and the reason it was decided to *box* primitives was to enable annotation of primitives and arbitrary modification of primitive behavior.

```
(defmethod o:upgrade-box ((box standard-box))
  (let ((wrapped-object (o:wrapped-object box)))
    (change-class box 'standard-object)
    (let ((forwarding-parent-slot
            (o:make-slot :slot-parent-p t
                         :assignable-p nil
                         :slot-key +hidden-parent-selector+)))
      (o:add-slot box forwarding-parent-slot
                  (o:box-primitive wrapped-object))))
  box)

(defmethod o:add-slot ((object standard-box) slot-description value)
  (o:upgrade-box object)
  (o:add-slot object slot-description value))
```

Listing 5.3: Upgrading Boxes to Standard Objects

When an attempt is made to add a new slot to an instance of a primitive in the guest environment, we cannot add the slot directly to the object map because (as described in section 5.2.1) this would add the slot to all instances of the same primitive type because share the same object map. The next step therefore would be to create an instance of standard-object to add the new slot, and also add a parent slot pointing to the original primitive. The problem with this is that any references to the primitive instance would still be pointing to the same instance, not the new one, and the side effect of adding the slot would not be propagated. To avoid this, we can instead upgrade the box using *change-class*, preserving the object's identity while still propagating the side effect of adding the new slot, then add a parent slot with a copy of primitive instance (Which in the context of the guest environment would be identical and not just because we have made it invisible). The methods which implement this behaviour are shown in Listing 5.3.

5.2.3 New Behavior

```
(define-box-method (name (box-name &key (machine-var 'machine)
                                        (self-var 'self))
                         (&rest arguments)
                         &body body))
```

Listing 5.4: The macro arguments to add-box-method

```
(define-box-method "if-true" (primitive-boolean) (true-block)
  (when (o:wrapped-object self)
    (%activate-block true-block machine))
  nil)
```

Listing 5.5: Adding a primitive method to a box

To add methods to our primitives that can be used in the guest environment, we can use the convenience macro define-box-method (Listing 5.4), this will create a slot on the primitive's object map associated with the foreign-method. To demonstrate this, in listing 5.5 we add a box method to our primitive-boolean, with the message selector "if-true", this method is the equivalent to the typical Smalltalk message *ifTrue:* which will activate the block provided as the only argument when the primitive is of the value *true*. All instances of a box of the same type share the same object map and methods.

5.2.4 Methods and Blocks

Method Blocks

Method blocks extend standard-object to provide the methods that are created and used by the guest system. There are three parts to a method-block, the method's code, which are the instructions the machine will execute, the literals, which are object literals the instructions refer to, and the method's arguments which are slots the block has that are to be initialized by popping from the machine's execution stack when the method is called.

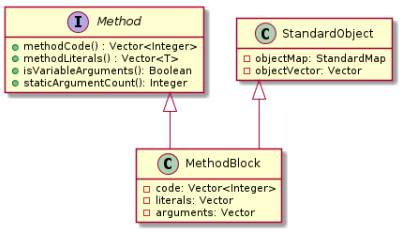


Figure 5.3: The method-block class and its inheritance of standard object.

Essentially the code, literals and arguments should be treated as a cache to be used by the machine in a format suited for the machine, in the case of arguments in particular, these will also be slot descriptions found in the method block's object map however they are grouped together in an extra slot to make it convenient for the machine to find them and initiate them when cloning the method block to create an activation record.

```
(defclass method-block (o:method standard-object)
  ((%code :initarg :method-code :reader o:method-code)
   (%literals :initarg :method-literals :reader o:method-literals)
   (%arguments :initarg :method-arguments :reader o:method-arguments
               :initform (make-array 3 :adjustable t :fill-pointer 0)
               :documentation "A vector of argument slots in the order
they should be removed from the execution stack.")))
```

Listing 5.6: The class definition for the method-block.

Blocks

Blocks are unusual and harder to implement than all the other primitives discussed in this report as they cannot be created by a client program outside of runtime because they need to refer to an activation record to provide their enclosing lexical scope.

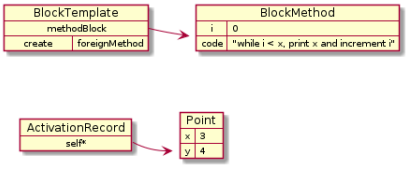


Figure 5.4: Block template and method before invoking the create method

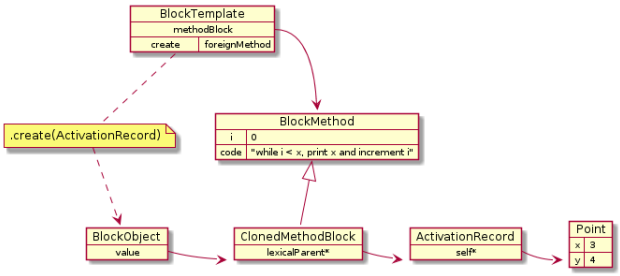


Figure 5.5: Block template and method after invoking the create method
Note: The link between BlockMethod and ClonedBlockMethod is only showing that the ClonedMethodBlock is a clone, there is in fact no live relation between the two

Blocks are implemented with the concept of a *block template* which is how blocks are implemented in the early Smalltalk interpreters [Goldberg and Robson, 1983] [Budd, 1987]. These are objects that can receive a message to create a block with the associated lexical environment as an argument in order to produce a block for that context. Figure 5.4 shows a block template referring to a simple method and an activation record (presumably the current one) that we want to use lexical context, in figure 5.5 we see the result of sending the create message with that activation record as the argument, producing a new block object with a method slot referring to a cloned method containing the activation record as its lexical context.

Methods

Method blocks essentially can already be used as methods but we subclass the method blocks to make for good organisation of the project as it is likely there will be a need for method specific functionality, however as of writing no such functionality has been added.

5.3 Machine & Interpreter

Method Activation & Evaluation

The machine is essentially started by activating a method and evaluating the instructions associated with it. When a method is activated in the machine, the following steps occur

1. The method is cloned, creating a new activation record identical to the method.
2. We hook into the behaviour of the clone method to add a program-counter slot to the activation record, initialized at 0.
3. We hook into the clone method to add a slot to contain the escape continuation (The method to return to once execution has completed) if there is one.
4. The arguments for the method are queried using the *argument-slots* generic function, which tells the machine how many arguments to pop from the stack.
5. The arguments are popped from the stack in order, initializing the slots associated with each argument on the activation record.
6. Evaluation can begin by looking for the instruction that the index in program-counter on the activation record refers to.

This is roughly identical to what is described in the Self VM [Chambers, 1992], however we also allow for methods to be given a varying number of arguments through the use of a *rest-argument-slot* and the *argument-count-register*. When a method can accept a varying number of arguments, the activation process is exactly the same apart from Step 5. Instead any normal arguments are popped from the stack and then the *argument-count-register* is read to determine how many arguments to pop from the stack and initialize the *rest-argument-slot* with on the activation record.

Interpreter loop

```
(defun %evaluate-activation-record (machine activation code)
  (let ((old-record (active-record machine)))
    (setf (active-record machine) activation)
    (loop :while (< (program-counter activation) (length code))
          :for instruction := (aref code (program-counter activation))
          :do (evaluate-instruction machine activation instruction))
    (setf (active-record machine) old-record))  )
```

Listing 5.7: Evaluating an activation record.

Evaluating activation records is fairly simple, each instruction in the method's bytecode is evaluated until we reach the end of the array where we return the the calling method (Listing 5.7). Each instruction's behaviour is implemented separately and is controlled by a typical dispatch function (Listing 5.8).

```
(defun evaluate-instruction (machine activation instruction)
  (let ((opcode (opcode instruction))
        (operand (operand instruction)))
    (ecase opcode
      (#.+op-self+ (op-self machine activation))
      (#.+op-literal+ (op-literal machine activation operand))
      (#.+op-send+ (op-send machine activation operand))
      (#.+op-implicit-self-send+ (op-implicit-self-send machine
   activation operand))
      (#.+op-arg-count+ (op-arg-count machine operand))
      (#.+op-activation+ (op-activation machine))
      (#.+op-escape+ (op-escape machine))
      (#.+op-reset+ (op-reset machine)))
    (incf (program-counter activation))))
```

Listing 5.8: The dispatch function at the heart of the interpreter.

From within the *op-send* & *op-implicit-self-send* functions we call the *lookup-message* & *evaluate-message* generic functions described in section 3.2.2.

So far we have described how the interpreter flows normally, but as we have the *escape* and *reset* instructions available to methods, it is possible for methods to perform non local transfers of control and this requires maintenance of both the stack the interpreter is running on in the host, Common Lisp, and in the the guest environment which includes invalidating any activation records which a guest program may keep live references to.

You may recall from the design section 3.5 that we intend to use a dynamic-environment in order to book-keep this information. This is implemented as a list of entries, each containing a reference to the activation record that was active when they were introduced. This allows us to *pop* and invalidate entries from the environment easily when unwinding as we can stop unwinding as soon as we reach an entry referencing the activation record that we are returning control to.

```
(defmethod dynamic-unwind ((machine standard-machine) exit-point-entry)
  (loop :for record := (first (environment-head machine))
        :until (or (null record)
                   (eq (activation-record record)
                       (escape-record exit-point-entry)))
        :when (typep record 'exit-point-entry)
        :do (invalidate record)
        :do (pop (environment-head machine)))))
```

Listing 5.9: Unwinding the guest.

When a non-local-transfer of control is executed via either the *reset* or *escape* instructions, first the guest environment is unwound using the *dynamic-unwind* generic function 5.9 and then a condition of type *non-local-transfer-of-control* is signalled to unwind the host interpreter's state to the target activation record 5.10.

```
(defun transfer-control (activation-record)
  (signal 'non-local-transfer-of-control
          :activation-record activation-record))

(defun op-escape (machine)
  (let ((record (m:pop-execution-stack machine)))
    (assert (validp record))
    (dynamic-unwind machine record)
    (let ((escape-record (escape-record record)))
      (assert (not (null escape-record)))
      (incf (program-counter escape-record))
      (transfer-control escape-record))))

(defun evaluate-activation-record (machine activation code)
  (let ((activation activation)
        (code code))
    (tagbody :start
       (handler-bind ((non-local-transfer-of-control
                       (lambda (c)
                         (when (eq activation (activation-record c))
                           (setf (active-record machine)
                                 (activation-record c))
                           (go :start)))))
         (%evaluate-activation-record machine activation code)))))
```

Listing 5.10: Unwinding the interpreter stack.

Chapter 6

Evaluation

6.1 Overview

The project aims have been met, we selected a set of opcodes and primitive objects and have also implemented them. We're now left with evaluating what has been produced.

Now with such ambiguous aims it might concern the reader how it was decided when to stop developing the project and this is something that was considered from the start. We listed the byte-code operations and primitives that were selected in the design section (3.3.1 & 3.4), essentially enough to implement conditional flow control and loops. In order to test the condition that these had all been implemented we developed an integration test written in the machine byte-code essentially running the following *psuedocode* program. (Note: This test is in addition to all unit tests for these individual components, we are not suggesting this is the only test or even integration test.)

```
  i = 0
loop-start:
  if i > 5:
    goto end
  i = i + 1
  goto loop-start
end:
```

Listing 6.1: The final test.

Now this would be trivial in normal circumstances, but it does require that all of the instructions and primitives that were selected function correctly, the full program can be found in the appendix A.1 (it is 60 lines long).

6.2 Discussion of the prototype

The principle behind having a small interpreter setting up the *message send* has served well. The interpreter core is roughly under 200 lines of Common Lisp and most of the functionality is provided by the initial set of primitives and methods of the lookup-message & evaluate-message pair of generic functions we presented in section 3.2.2. One could use the analogy of the *message send* as the underlying semantic behind the virtual machine and the initial set of primitives as a library. In order to extend the functionality of the VM one shouldn't need to modify or patch the interpreter core, only add new objects and methods to the lookup-message & evaluate-message generic functions.

The tools used were instrumental to the success of the project, being able to use Parachute's interactive test mode and thus being able to examine the state of the ma-

chine at a point of failure. using Self inspired small number of bytecodes was good, resources were great, small core was good

probably talk about how hard it is to write a program without a guest language and reference that section

6.3 Technical Improvements

6.3.1 Instructions

The number of instructions that the machine requires could be reduced, for instance, the reset and escape instructions can be implemented as messages to activation records instead. This would align more with the 'desirable properties' we stated in the background research 2.1 by increasing the transparency of the activation record primitive, it is entirely reasonable for someone to want to proxy an activation record despite how essential it is to this model of computation.

In addition to the above, the *argument-count-register* could also be eliminated by requiring that the *implicit-self-send* and *send* instructions give the number of arguments they are sending the message with as the operand (and popping the message selector from the stack first).

The reader may notice that the Self instructions *resend* & *directee* [Chambers, 1992] are absent from this machine, this is because we did not need to implement the *directed-resend*, however when it is needed to it would also align with our 'desirable properties' if we were able to implement them as messages available on *standard-object* (or the activation record).

6.3.2 Guest Language

Although there are no current technical limitations by not having a guest language to aid the development of the machine, having a guest language available would greatly improve the maintainability of the project when adding new methods to the intrinsic objects in the system as the guest language would be designed around the semantics of the machine and associated object system (whereas the host environment is not). A guest language would also help when writing tests that require new methods to be written as it is tedious writing byte-code programs by hand as evident in listing 6.1 & appendix A.1.

Therefor a good first step upon taking the project further is to design a suitable guest language and start the steps to bootstrap the guest system for the purpose of maintaining the machine itself. From there the guest language can support any further experimentation and the addition of any new primitives or tests.

6.3.3 Boxed Primitives

As we have discussed we used *primitive boxes* to provide the primitives to the VM and this provided the ability to arbitrarily modify instances of primitives just like any other *standard object* in the system. It still remains unclear if there is a benefit to providing this capability to every primitive and we believe the capability should be discussed on a case by case basis. For instance, boxing numeric types can be problematic for efficiency (they require more space and make it more difficult to use platform specific operations) and it's unlikely one would annotate a number that is going to be very short lived and is in no way involved in the abstraction or structure of a program. Instances of *slot*

description and *standard object* generally are longer lived and are used as structural components in the program that aid the developer, so we reason it is more likely that one would want to annotate them arbitrarily. A more balanced case would be data structures containing *places*, these could be long lived or short lived depending on the circumstance of their use and we further reason that primitives containing *places* would be strong candidates for permitting arbitrary annotation in order to enable the use of *pluggable type systems* [Bracha, 2004].

6.4 Personal & Professional development

While this project has the resemblance of a professional one, it is the work of a passionate amateur, the author has received no income in relation to the development of dynamic-interactive systems at the time of writing. On a personal note my main professional concern is to use this experience to pursue a career, academic or otherwise, in the development of dynamic-interactive systems. Even if I am to pursue a career as a traditional *software developer*, this experience will still have provided a unique insight into the internals of similar systems. This is a perspective that I would likely be missing if I had produced any other project and my investigation is not going to stop irrespective of any outcome.

6.5 Ethical Concerns

Although it may seem there are a limited number of ethical concerns as we are not currently dealing with any third party, it is still important to address concerns as though we are directly or indirectly dealing with a third party, as the project is likely to in future even if that is in the form of a derivative work. We use an article by Alan J. Thomson and Daniel L. Schmoldt [Thomson and Schmoldt, 2001] written for the United States department of Agriculture to help identify some ethical concerns for this project. Foremost it would be important be to ensure that clients are informed about the experimental nature of the project, that is to say that it is subject to dramatic change, and inform them of the details of these changes as they occur. This not only includes technical changes but changes in leadership and the maintenance of the project so that they are able to take informed action. Equally it is important to inform the clients of any limitations with regards to the accuracy of any numeric primitive provided with the VM so that they can be accounted for in any calculations that are made. We also have concerns for the future of research of this nature in general, we were aided significantly by the research of those who preceded us in exploring Smalltalk virtual machines and it was fortunate that not only were we able to discover the works of the Self team and in particular the PhD thesis of Craig Chambers [Chambers, 1992] early on, but also fortunate that their work was funded and published in the first place. It is important for it to remain obvious to both academia & industry that innovation comes from ideas that directly oppose the *status quo* [Kay, 1997], many of today's popular programming languages incorporate innovations appropriated from a dynamic-interactive origin that are now far removed from this context [Bracha, 2013].

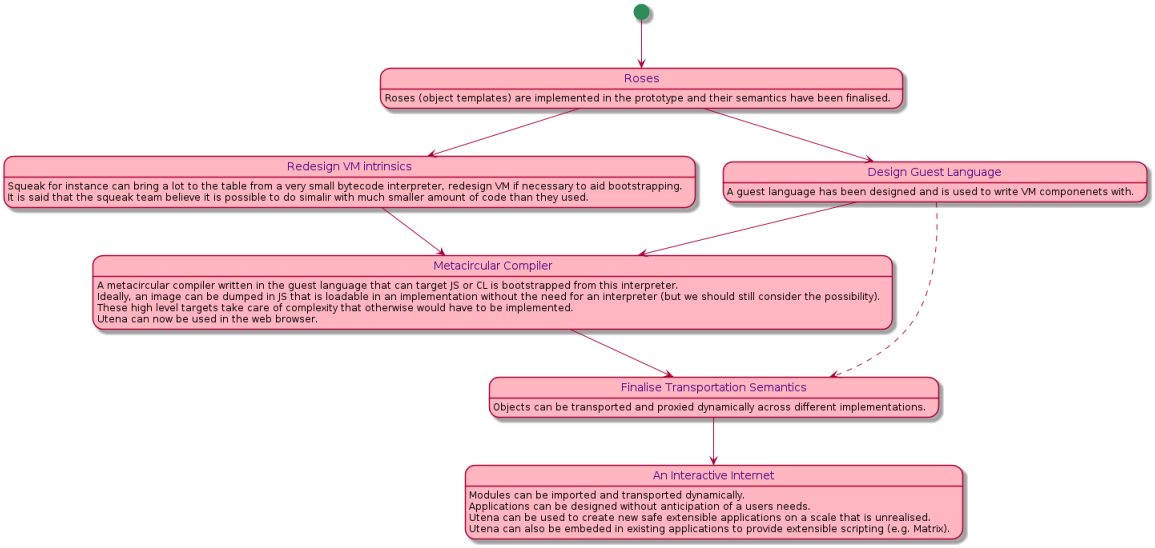


Figure 6.1: A road map for the future of Utena VM

6.6 Further work

Continuing on from section 6.3, we have produced a *road map* (Figure 6.1) to visualize the next pieces of work that could be produced. The clear next logical step is to make progress towards designing and create a guest language using the VM, this will require some light refactoring of the VM intrinsics as the guest language's needs are understood. Once we have something working, steps should be made towards the creation of a meta-circular compiler that can target a high level interactive system such as those provided by Common Lisp or JavaScript. JavaScript would be an ideal choice as us to develop applications in the web browser and this likely where most use cases for Utena lie. From then on the transportation system and semantics can be realised fully. As a final note, it is possible to create a visual programming environment that encapsulates the fundamental semantics in the core interpreter that will be able to inter-operate transparently with other programs on the machine. The design of this virtual machine is not too dissimilar to that the Squeak, Squeak Etoys [Kay, 2005] being a major project in this domain and the precursor to Scratch, which also originated as a Squeak Etoys application [Resnick et al., 2009]. Unified widespread adoption of environments such as these could be vital for the normalization of end-user programming and achievement of a consistent context for computer literacy.

Appendix A

Programs

A.1 'The final test'

```
(define-test-method *escaping-block* (o:make-block-template)
    ("escape-record")
  (sm:+op-implicit-self-send+ 0)
  (sm:+op-escape+ 0))

(define-test-method *simple-reset-block* (o:make-block-template)
    ("i" "i:" ">" 5 "if-true" "escape-block" 1 "+" "escape-record:")
  ;; put the AR onto the stack
  (sm:+op-activation+ 0)
  ;; give the AR to escape-record
  (sm:+op-implicit-self-send+ 8)
  ;; put the escape block onto the stack
  (sm:+op-implicit-self-send+ 5)
  ;; put the value 5 on the stack
  (sm:+op-literal+ 3)
  ;; put the value of x on the stack
  (sm:+op-implicit-self-send+ 0)
  ;; set arg count register to 1
  (sm:+op-arg-count+ 1)
  ;; send > to the value of i and compare against 5
  ;; if i > 5:
  (sm:+op-send+ 2)
  ;; ifTrue, use the escape block
  (sm:+op-send+ 4)
  ;; set i to i + 1 by putting i down first
  (sm:+op-implicit-self-send+ 0)
  ;; put the 1 on the stack
  (sm:+op-literal+ 6)
  ;; send + to the 1 and i
  (sm:+op-send+ 7)
  ;; set i to the result
  (sm:+op-implicit-self-send+ 1)
  ;; put the AR onto the atck
  (sm:+op-activation+ 0)
  ;; reset
  (sm:+op-reset+ 0))

(define-test-method *simple-reset-method* ()
    (*catch-prototype* 0 5 "i" "i:" *escaping-block* "create" "
    escape-block:"
     *simple-reset-block* "value")
```

```
;; push 0 onto the stack
(sm:+op-literal+ 1)
;; set i to 0
(sm:+op-implicit-self-send+ 4)
;; put the lexical context that we want the escaping block to have
;; onto the stack
(sm:+op-activation+ 0)
;; get the escaping block literal and create it
(sm:+op-literal+ 5)
;; send the create method with the block and the lexical context
(sm:+op-send+ 6)
;; set up escape-block to hold the created escpae block
(sm:+op-implicit-self-send+ 7)
;; setup the simple reset block
(sm:+op-activation+ 0)
;; put the simple reset block template onto the stack
(sm:+op-literal+ 8)
;; send create to the block template
(sm:+op-send+ 6)
;; send value to the created block template
(sm:+op-send+ 9)
;; put i back onto the execution stack
(sm:+op-implicit-self-send+ 3))
```

Listing A.1: The final test when written by hand.

Glossary

Activation Record An activation record is the Smalltalk machine equivalent to a *stack frame*. When a method is called we say that the method is 'activated' and the values for the arguments are stored in an activation record. An activation record is a first class object, so these values are usually provided with slots much the same way as any other object. In our VM, the activation record provides the lexical environment for an active method. 9, 11, 12, 14

Annotatable (also annotation, annotating) An object which is annotatable allows for arbirtrary metadata to be associated with it through the use of a mirror [Bracha and Ungar, 2004]. 32

Annotate *See* annotatable. 11

Annotation *See* annotatable. 11, 21

Box (also boxing, boxed) Boxing primitives is a technique used by implementations to present a foreign data type to the guest environment in a way that is understood by it, usually by implementing some established protocol. The way this is done in our Virtual Machine (VM) is by essentially wrapping the host object to make it appear as a natural guest object in the guest environment. See section 5.2.2. 11, 32

Boxing *See* box. 21

Guest (also Guest Environment, Host, Host Environment) The guest is the environment that we produce as a result of the program that exists in the host, which in our case is a Common Lisp system. Essentially we use the term guest to refer to the environment and object system provided by this virtual machine and the host to refer to the environment that the guest exists within. This does not imply that the host is accessible from the guest, but it is often that the host and the guest can inter-operate with each other in specific situations. 4–6, 32

Host *See* guest. 11, 13, 32

Intrinsic (also **primitive)** These are objects or messages that a guest language can expect to be implemented by the virtual machine and be ready for use, usually intrinsics are selected on the basis that the guest language may have difficulty providing an implementation for them as they require access to hardware or operating system functions that are not exposed in the guest environment. 33

JVM Java Virtual Machine. 4

Lexical Environment The lexical environment refers to the range of messages that are accessible from the context of an activation record by using an *implicit self send*. For example, a method's arguments are in the lexical environment from the context of that method as they are accessible via an *implicit self send*. This definition applies to both Self and the object system we provide with this virtual machine.. 12, 15, 32

Map This refers to an object that matches the available message selectors on an object to slot descriptions. The slot descriptions may also then contain information on how to read the value of the slot, hence the name 'map' as they provide a map to the object's instance vector, we discuss our implementation of maps in section 5.1. A map is said to *split* when two or more objects exist in the system referring to two distinct but structurally identical maps. 5, 19–22

Message Selector A message selector is easiest to describe as a key that is associated with a slot on an object. It's called this because essentially it's the key that will select which message will be evaluated from the messages an object has available. We usually refer to the *message selector* when dissecting the components of a message (the receiver, the arguments, the message selector). 8, 10–12, 19, 22

Object Capabilities (also **Object Capability Model)** The Object Capability model is a way of managing any privilege such as file system access by stating how objects can gain access to them. Very simply we can say that an object can only have access to the objects it was created with or the objects it has been given by another object but it is not as simple as this. The earliest instance of 'capabilities' in this sense comes from Jack B. Dennis and Earl C. Van Horn's 'Programming Semantics for Multiprogrammed Computations' [Dennis and Van Horn, 1966] but this is a very old resource and more work has been done since, instead to gain a better understanding I suggest Mark S. Miller's PhD thesis [Miller, 2006] and an attempt to bring the principles of object capabilities to JavaScript [Miller et al., 2008]. 33

Object Capability Model *See* Object Capabilities (also Object Capability Model). 3

Place A place essentially is a location where a reference to an object can be accessed, for example, the value of a *value slot* is a place and so is an element of a generic array type. This term is borrowed from the ANSI Common Lisp specification [ans, 1994, Chapter 5]. 28

Primitive *See* intrinsic. 10, 11

Prototype-Based A prototype-based object system put very simply is an object system without classes and instead new instances are created by *cloning* existing objects. However, it really is not as simple as this, and we use the term liberally in the sense that without classes the object system is prototype-based. it should also be noted it is completely within reason that objects can be created with factories and other patterns common in class based systems. Behaviour without classes can also be added by objects referencing *traits* objects or *mixins* with a *parent slot* [Ungar et al., 1991]. 8, 15

Virtual Machine We use the term virtual machine to mean a system that can be implemented in order to portably provide all the intrinsics for a single or specific set of guest languages. This is in contrast to a virtual machine that is built for the purpose of emulating a physical machine, we're instead developing a virtual machine to explore a new model of computation. 3, 15, 32, 34

VM Virtual Machine. 3, 4, 6, 11, 24, 26, 28, 32

Bibliography

[ans, 1994] (1994). *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp*. American National Standards Institute.

[ora, 2015] (2015). *The Java Virtual Machine Specification*. Oracle. `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

[Allen et al., 2017] Allen, R., Agesen, O., Bak, L., Chambers, C., Chang, B.-W., Hölzle, U., Maloney, J., Pape, T., Smith, R. B., Ungar, D., and Wolczko, M. (2017). Self handbook documentation. `https://handbook.selflanguage.org`.

[Armstrong, 2003] Armstrong, J. (2003). *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden.

[Artoul, 2015] Artoul, R. (2015). Javascript Hidden Classes and Inline Caching in V8. `https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html`, Accessed: 2020-10-28.

[Botev, 2012] Botev, N. (2012). Actor-based concurrency in newspeak 4. `https://scholarworks.sjsu.edu/etd_projects/231/`, Accessed: 2020-11-04.

[Bracha, 2004] Bracha, G. (2004). Pluggable type systems. In *OOPSLA Workshopon Revival of Dynamic Languages.*, OOPSLA '04.

[Bracha, 2013] Bracha, G. (2013). Does thought crime pay? In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, page 7–8, New York, NY, USA. Association for Computing Machinery.

[Bracha and Ungar, 2004] Bracha, G. and Ungar, D. (2004). Mirrors: Design principles for meta-level facilities of object-oriented programming languages. *SIGPLAN Not.*, 39(10):331–344.

[Bracha et al., 2010] Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., and Miranda, E. (2010). Modules as objects in newspeak. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, page 405–428, Berlin, Heidelberg. Springer-Verlag. `http://bracha.org/newspeak-modules.pdf`.

[Budd, 1987] Budd, T. (1987). *A little Smalltalk*. Addison-Wesley Longman Publishing Co., Inc.

[Béra, 2017] Béra, C. (2017). *Sista: a Metacircular Architecture for Runtime Optimisation Persistence*. PhD thesis, Université de Lille. `https://hal.inria.fr/tel-01634137/`.

[Chambers, 1992] Chambers, C. (1992). *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, The Department of Computer Science and the Committee on Graduate Studies of Stanford University. `https://www.researchgate.net/publication/2827867_The_Design_and_Implementation_of_the_SELF_Compiler_an_Optimizing_Compiler_for_Object-Oriented_Programming_Languages`, Accessed: 2020-10-15.

[Chambers et al., 1989] Chambers, C., Ungar, D., and Lee, E. (1989). An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70.

[Dennis and Van Horn, 1966] Dennis, J. B. and Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155.

[Fitzgerald and Stol, 2014] Fitzgerald, B. and Stol, K.-J. (2014). Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, page 1–9, New York, NY, USA. Association for Computing Machinery.

[Fowler and Foemmel, 2006] Fowler, M. and Foemmel, M. (2006). Continuous integration. *Thought-Works*, 122(14):1–7.

[Germain, 2006] Germain, G. (2006). Concurrency oriented programming in termite scheme. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, page 20, New York, NY, USA. Association for Computing Machinery.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., USA.

[Gordon and Bieman, 1995] Gordon, V. S. and Bieman, J. M. (1995). Rapid prototyping: Lessons learned. *IEEE Softw.*, 12(1):85–95.

[Herda, 2020] Herda, M. (2020). *The Common Lisp Condition System*. Apress, Berkeley, CA.

[Ingalls, 1981] Ingalls, D. (1981). Design principles behind smalltalk. *BYTE*, 6(8):286–299. `https://www.cs.virginia.edu/~evans/cs655/readings/smalltalk.html`, `https://archive.org/details/byte-magazine-1981-08/page/n299/mode/2up`.

[Ingalls et al., 1997] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. (1997). Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, page 318–326, New York, NY, USA. Association for Computing Machinery.

[Kay, 1997] Kay, A. (1997). The computer revolution hasn't happened yet. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, New York, NY, USA. Association for Computing Machinery.

[Kay, 2005] Kay, A. (2005). Squeak etoys, children & learning. *online article*, 2006.

[Miller, 2006] Miller, M. S. (2006). *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University.

[Miller et al., 2008] Miller, M. S., Samuel, M., Laurie, B., Awad, I., and Stay, M. (2008). Safe active content in sanitized javascript. *Google, Inc., Tech. Rep.*

[Miranda et al., 2018] Miranda, E., Béra, C., Boix, E. G., and Ingalls, D. (2018). Two decades of smalltalk vm development: Live vm development through simulation tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2018, page 57–66, New York, NY, USA. Association for Computing Machinery.

[Pitman, 1994] Pitman, K. M. (1994). Accelerating hindsight: Lisp as a vehicle for rapid prototyping. *SIGPLAN Lisp Pointers*, VII(1–2):14–21.

[Pressler, 2019] Pressler, R. (2019). Project loom: Fibers and continuations for the java virtual machine. `http://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html`. Accessed: 2020-10-15.

[Pressler, 2020] Pressler, R. (2020). State of loom. `http://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html`. Accessed: 2020-10-15.

[Resnick et al., 2009] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). Scratch: Programming for all. *Commun. ACM*, 52(11):60–67.

[Shinn et al., 2013] Shinn, A., Cowan, J., (editors, A. A. G., Ganz, S., Radul, A., Shivers, O., Hsu, A. W., Read, J. T., Snell-pym, A., Lucier, B., Rush, D., Sussman, G. J., Medernach, E., Russel, B. L., Kelsey, R., Clinger, W., Rees, J., and Sperber, M. (2013). Revised 7 report on the algorithmic language scheme. `https://small.r7rs.org/attachment/r7rs.pdf`.

[Sockwell, 2021] Sockwell, D. (2021). Imagining the ideal language for writing free software. `https://fosdem.org/2021/schedule/event/programming_lang_for_free_software/`.

[Strandh, 2013] Strandh, R. (2013). SICL Building blocks for creators of Common Lisp implementations. `http://metamodular.com/SICL/sicl-specification.pdf`, Accessed: 2020-10-15.

[Taylor and Standish, 1982] Taylor, T. and Standish, T. A. (1982). Initial thoughts on rapid prototyping techniques. In *Proceedings of the Workshop on Rapid Prototyping*, page 160–166, New York, NY, USA. Association for Computing Machinery.

[Thomson and Schmoldt, 2001] Thomson, A. J. and Schmoldt, D. L. (2001). Ethics in computer software design and development. *Computers and Electronics in Agriculture*, 30(1):85–102.

[Ungar et al., 1991] Ungar, D., Chambers, C., Chang, B.-W., and Hölzle, U. (1991). Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242.

[Ungar et al., 2005] Ungar, D., Spitz, A., and Ausch, A. (2005). Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to*

the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05, page 11–20, New York, NY, USA. Association for Computing Machinery.

[Wimmer et al., 2013] Wimmer, C., Haupt, M., Van De Vanter, M. L., Jordan, M., Daynès, L., and Simon, D. (2013). Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4).

[Wolczko, 1996] Wolczko, M. (1996). self includes: Smalltalk. In *Workshop on Prototype-Based Languages*. `https://www.researchgate.net/publication/265807184_self_includes_Smalltalk`.

[Wolczko et al., 1999] Wolczko, M., Agesen, O., and Ungar, D. (1999). Towards a universal implementation substrate for object-oriented languages. *Proceedings of the Workshop on Simplicity, Performance and Portability in Virtual Machine Design*. `https://www.researchgate.net/publication/2398949_Towards_a_Universal_Implementation_Substrate_for_Object-Oriented_Languages`, Accessed: 2020-10-15.

[Würthinger et al., 2013] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., and Wolczko, M. (2013). One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2013, page 187–204, New York, NY, USA. Association for Computing Machinery.