HAL
open science

A Partial Read Barrier for Efficient Support of Live Object-oriented Programming

Eliot Miranda, Clément Béra

To cite this version:

Eliot Miranda, Clément Béra. A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. International Symposium on Memory Management, Jun 2015, Portland, United States. pp.93-104 10.1145/2754169.2754186 . hal-01152610

HAL Id: hal-01152610

https://inria.hal.science/hal-01152610v1

Submitted on 26 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Partial Read Barrier for Efficient Support of Live Object-Oriented Programming

Eliot Miranda
Cadence Design Systems, USA
eliot.miranda@gmail.com

Clément Béra
RMOD - INRIA Lille Nord Europe, France
clement.bera@inria.fr

Abstract

Live programming, originally introduced by Smalltalk and Lisp, and now gaining popularity in contemporary systems such as Swift, requires on-the-fly support for object schema migration, such that the layout of objects may be changed while the program is at one and the same time being run and developed. In Smalltalk schema migration is supported by two primitives, one that answers a collection of all instances of a class, and one that exchanges the identities of pairs of objects, called the become primitive. Existing instances are collected, copies using the new schema created, state copied from old to new, and the two exchanged with become, effecting the schema migration.

Historically the implementation of become has either required an extra level of indirection between an object's address and its body, slowing down slot access, or has required a sweep of all objects, a very slow operation on large heaps. Spur, a new object representation and memory manager for Smalltalk-like languages, has neither of these deficiencies. It uses direct pointers but still provides a fast become operation in large heaps, thanks to forwarding objects that when read conceptually answer another object and a partial read barrier that avoids the cost of explicitly checking for forwarding objects on the vast majority of object accesses.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.6 [*Software Engineering*]: Programming Environments—Interactive environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management; D.3.4 [*Programming Languages*]: Processors—Incremental compilers[Run-time environments]; D.4.2 [*Programming Languages*]: Language Constructs and Features—Garbage collection; E.2 [*Data*]: Data Storage Representations—Object representation

Keywords Live programming, schema migration, object representation, garbage collection, language virtual machine, forwarding pointer, read barrier.

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Smalltalk has provided incremental development since the late '70s. A Smalltalk system is a self-describing set of objects running above a minimal virtual machine. Classes, methods, threads, the compiler, the debugger and the programming tools are all represented as Smalltalk objects, described by classes in the system. The virtual machine therefore contains only an execution engine, a memory manager and a set of primitives. Smalltalk reifies even activation records (usually represented as stack frames in most language implementations) as objects (Deutsch and Schiffman 1984; Miranda 1999). Therefore the debugger is written entirely in Smalltalk, displaying objects representing stack frames, which are themselves capable of interpreting the bytecoded instruction set generated by the compiler and efficiently executed by the virtual machine.

The programmer may at any time, for example when in the debugger, want to add or remove methods and add or remove instance variables in class definitions. The live system of objects allows the programmer to do so.[1] In the case where the number of instance variables of a class is modified, the system will update all the class instances accordingly.[2] The virtual machine must provide a suitable set of primitives upon which to implement this schema migration.

Schema migration is performed with two main primitives. The first one, `allInstances`, answers a collec-

[1] Adding or removing instance variables is not possible on a small number of classes depended on by the virtual machine

[2] It is now common to program in the debugger. One may start by writing a test as a method, referencing not-yet-defined classes, execute it and in the resulting debugger fill in a form to define the class, continue execution until some unimplemented message is sent to an instance of the class, and then define that method given a template generated from the MessageNotUnderstood exception that results from sending the unimplemented message.

tion of all the instances of a given class. The second one, become, conceptually changes all the references to an object a into references to another object b. The implementation of become in the original Smalltalk implementations, still used by VisualWorks Smalltalk (Cincom), involves adding a level of indirection to every object. An object pointer points at a header which contains a pointer to the object's body. This makes implementing become trivial by exchanging the pointers in their headers. This incurs the cost of an extra indirection to access the fields of an object and the space overhead of an indirection pointer in every header.

An alternative approach is to use "direct pointers" where the object header and body are always adjacent. Our previous implementation of become using direct pointers requires a full heap scan, looking for all references to the object a in each object in the system and replacing them with references to object b, and vice verse. This implementation of the become primitive has an important flaw: its performance is linearly proportional to the heap size.

This paper describes the design and implementation of efficient schema migration using direct pointers, which consists mainly in hiding the cost of checking for forwarding pointers behind other checking operations that the system performs as part of its normal processing. This new implementation uses *forwarding objects*, which conceptually when read answer another object in memory. Any object may be converted into a forwarding pointer to another object. Become is therefore implemented lazily; copies of the pair of objects are created, and each original is forwarded to the matching copy; the forwarding pointer is followed when the object is encountered. Forwarding objects allow us to avoid the object table, and allow us to implement a become primitive with performance independent of heap size. Theoretically, adding an extra indirection to access objects through forwarding objects introduces a read barrier; it forces the virtual machine to check each time a reference to an object is read to determine if the object is a forwarding object or not. Such checks would however cost performance, and a significant increase in the size of code for slot access, which is important in a JIT. However, using the properties of the Smalltalk runtime, the virtual machine avoids the read barrier in most cases and needs to explicitly verify if a reference read is a forwarding object only in uncommon cases. Forwarding objects remain uncommon in the heap because the become operation is uncommon and because several aspects of the memory manager, such as the garbage collector, are aware of forwarding objects and remove them incrementally.

2. Problem: User Pauses in Schema Migration

To redefine a class in Smalltalk, the following tasks are performed. Firstly, for the class and all its subclasses, a copy of the class is created reflecting the new definition. Methods are recompiled from the original class into the corresponding copy, with their bytecodes reflecting the layout of instance variables in the new definition. None of this requires special VM support; classes and methods are objects and the compiler is a set of Smalltalk classes. Once the new schema has been created, for each pair of classes in the old and new schema, all instances of the old class are enumerated and a corresponding instance of the new class is created. Instance variable state is copied from each instance to its copy, new instance variables being initialized with nil. Finally all references to the old instances are replaced with references to the new instances using a become primitive.

The become primitive exists in essentially four varieties. The first form is two-way become, which exchanges the references to two objects a and b such that all references to a become references to b, and all references to b become references to a as shown on Figure 1. The second form is one-way become which changes the references to two objects a and b such that all references to a become references to b, and references to b remain unchanged. The third and fourth forms are bulk two-way and one-way becomes that take two arrays of objects as arguments, performing single becomes on matching pairs from each array. Depending on the implementation a bulk become may be much cheaper than several becomes, depending on whether the become primitive requires a full heap scan or not.

There are two extant implementations of become in modern Smalltalk runtimes.

The first one, used for example by Cincom Smalltalk (Cincom; Miranda 1999), requires every object to be split in memory between a fixed sized object header which contains a pointer to its body, and a variable-sized body. Every object-oriented pointer always targets the fixed-sized header of the object, and its fields are accessed through the pointer to its body. In this case, become only requires exchanging the pointers from headers to bodies in the two objects. This implementation has a flaw: accessing the fields of an object requires an extra indirection through the header.

The second one, used in the Cog virtual machine (Miranda 2008) we work on, relies on a full heap scan, swapping every pointer to objects being becomed to their new pointer. This implementation allows the virtual machine to store each object as a contiguous sequence of memory locations, but the become primitive performance decreases as the heap grows (the time spent to execute the primitive is directly proportional to the heap size, as shown in Section 5). Although this implementation looks expensive, it is still judged by the Self and Squeak VM implementors as preferable to object table indirection. Both implementations have an optimization if the objects to be becomed are in new space, only new space and the objects in the remembered table are scanned.

We wanted to design a solution for the become primitive that has none of these deficiencies: an object should be represented as a single contiguous sequence of memory locations

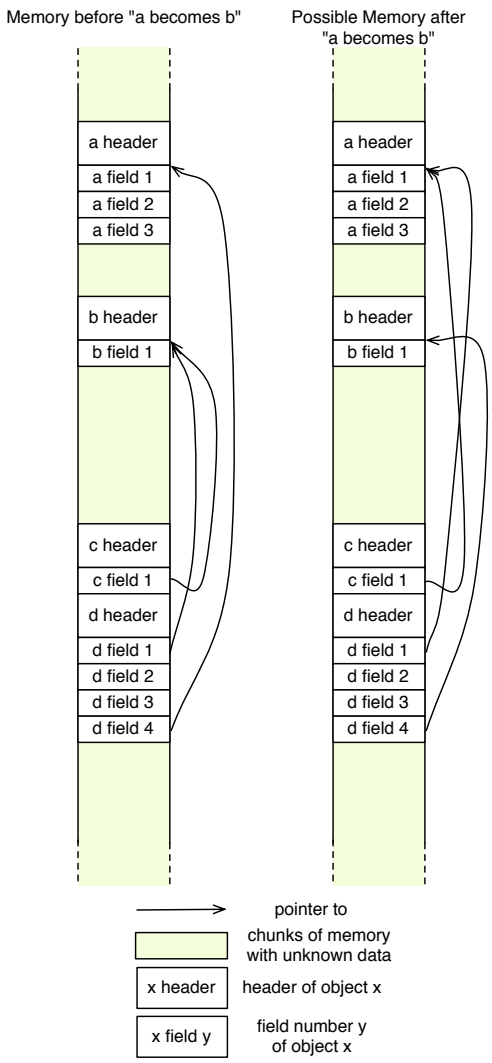

Figure 1. Two way become

using direct pointers, and the become primitive should not incur pauses noticeable to the user. With this goal in mind, we designed a new implementation of the become primitive, using forwarding objects, an old idea (Weinreb and Moon 1981), and the *partial read barrier* scheme, a new idea, which lazily follows indirections on objects when needed. We detail how we avoid checking at each object-oriented pointer read if the object read is a forwarding object, checking instead only on uncommon cases. Forwarding objects are lazily removed by the garbage collector.

3. Context: Smalltalk Object Representation

In this section we describe some aspects of the Spur memory manager that aid in understanding how the virtual machine avoids explicitly checking for a forwarding pointer each time it reads an object's slot.

Spur is an object representation and garbage collector for Smalltalk-like languages, currently being used in the Squeak (Guzdial and Rose 2001) and Pharo (Black et al. 2009) Smalltalk dialects and the Newspeak language (Bracha et al. 2010). It is a memory manager for the Cog VM, an evolution of the original Squeak VM (Ingalls et al. 1997) adding context-to-stack mapping (Deutsch and Schiffman 1984; Miranda 1999) and a JIT. Its main design goals are

- a common object representation between 32 and 64-bit implementations
- support of direct pointers and fast `become`
- support for object pinning (pinned objects are not moved by the garbage collector)
- more immediate (tagged) data types
- support for a segmented memory (the heap can grow and shrink a segment at a time)
- support for ephemerons (Hayes 1997)

The key innovation here is the efficient implementation of `become` using direct pointers while avoiding an explicit read barrier in the common case. In the following subsections we present the details of the object representation and instance enumeration, detailing how it impacts our implementation of the become primitive. The next section discusses how the VM as a whole implements the partial read barrier.

3.1 Object Representation in Spur

In Spur both 32- and 64-bit implementations share a common 64-bit object header. All objects are aligned on a 64-bit boundary. Following the object header are at least two 32-bit slots, in 32-bits, or at least one 64-bit slot, in 64-bits. Ensuring that all objects have at least one slot makes converting any object into a forwarding object possible, because a forwarding object requires a single field to record the object it forwards to. Unfortunately in our Smalltalk system there are many zero-sized objects, mostly Arrays. We measured that the size of the default Squeak Smalltalk 4.5 runtime grew by 4% due to the overhead of the slot reserved for the forwarding pointer.

The object's header holds information about the object, as shown in Figure 2:

- its size encoded in a single byte field: objects larger than 254 slots have a prepended 64-bit overflow size field; a space overhead of at most 1.6% per large object in the 32-bit system)
- 5 bit "format" field: indexable, non indexable with inst vars, weak, ephemeron, forwarding object, 8-bit, 16-bit, 32-bit or 64-bit indexable bits, compiled method
- its class index, encoded in 22 bits, allowing 4 million classes
- its hash, encoded in 22 bits

- bits for the GC: isPinned, isImmutable, isGrey, isRemembered, isImmutable
- Two unused bits

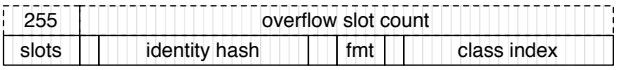

Figure 2. Spur object's header

The class index refers to the location of the class object in the class table. In addition to class indices for regular classes, a class index is explicitly reserved for forwarding objects, and among the different formats possible for an object, a distinct format is also used for forwarding objects.

3.2 Class Table

The common header format between 32- and 64-bit versions is made possible by the use of class indices in object headers instead of pointers, saving 10 and 42 bits respectively. Class indices index a sparse array of class table pages, grown on demand. To allow a class index to be quickly determined, the system is modified to arrange that the identityHash of a class is its index in the class table. So when a class is entered into the table (e.g. when it is first instantiated), or when it gains an identityHash (by being sent the hash message) an unused entry in the table is found and the class's identityHash is set to that index. To instantiate a class therefore, the class's identityHash is copied to the instance's class index, avoiding having to search the table. The few classes known to the VM such as Array, BlockClosure, MethodContext, Message, ByteString etc are assigned known class indices in the first class table page. Therefore instantiating these objects is simple; the header is computed using the relevant known class index, and is often a constant. In a system with a moving garbage collector and direct class pointers in headers, instantiation has to lookup a class in some table to obtain its current address and then copy this into the header. Spur's allocation of known objects, and checking for particular classes, is in contrast simpler and faster.

Class table pages are co-resident on the heap, but hidden from normal objects by themselves having a class index unused by normal classes; hence they will never show up in allInstances or allObjects primitives.

3.3 All Instances

`Allinstances`, critical for schema migration, is done simply by scanning all objects looking for those with a class index that matches that of the receiver class. Although it requires a full heap scan, it needs to check only the header of each object and doesn't need to look into all its object-oriented pointer fields.

3.3.1 Object Enumeration

As described above, the Spur object header has room for an 8-bit slot count. Objects up to 254 slots have a single header. Objects with 255 or more slots have an extra 64-bit overflow size word prepended, holding a 56-bit slot count. Both the prepended and the normal word have their slot count set to 255. To enumerate objects in the heap, the memory manager starts at the lowest heap address, and fetches the slot count from the 64-bit word at that address. If the slot count is 254 or less the object is a normal object, and the address following the object can be computed from the slot count. If the slot count is 255, then this address is that of a prepended overflow size word, the header follows that, and the number of slots is determined from the overflow size word. Object enumeration proceeds from the lowest object in memory continuing up to the limit of the heap.[3]

Old space is composed of segments, and each segment ends with a two word object header that claims its contents are bits, not pointers, and lies about its length so that it spans the gap to the next segment (**?**). To the rest of the VM, old space appears to be one contiguous sequence of objects.

3.4 Smalltalk Execution Model

The Smalltalk execution model (**?**) is that of methods executing bytecodes for a stack machine. A method is invokes by the machine first pushing the receiver, followed by the arguments and then sending a message, which is looked up in the class of the receiver. The only access to the state of an object is within methods of that object.

A method has a sequence of literals and a sequence of bytecodes. A method activation is represented by a context object which has a receiver, a method, local temporaries, stack state and a reference to a sender activation. Push and store instance variable bytecodes directly access the receiver. Push literal references literals. Push and store global variable access the value fields of globals[4] in the literals. Branch tests a boolean on top of stack and conditionally branches.

Unlike, for example, Java there are no byte codes that describe operations other than the closure creation bytecode. Every operation other than closure creation requires a send, and hence sends are extremely common in Smalltalk. Two aspects of sends require optimization. One is lookup, the other activation.

Notionally sending a message involves searching the class hierarchy, starting at the receiver's class for a method that matches the send's selector. Once the method is found it is examined to see if it has a primitive. If it has a primitive the VM performs the primitive, and if it succeeds, the receiver and arguments are replaced by the primitive's result

[3] This is an over simplification. The lowest part of the heap is in fact a conventional three-region eden plus two survivor spaces, managed by a generation scavenger; it is immediately followed by the first old space heap segment.

[4] A global is a two slot object, a key, value pair

and execution continues. If the method has no primitive, or if the primitive fails, then conceptually a new context object is created, receiver and arguments are moved from the sender context to the new context, and execution of the new method continues. Both operations must be heavily optimized to achieve good performance.

Sends themselves are sped up using a simple associative cache that stores methods and primitives keyed by message selectors and classes, called the first-level method lookup cache. Of course the JIT uses inline caching techniques to speed up sends (Deutsch and Schiffman 1984; Hölzle et al. 1991), but in this paper, for simplicity we consider only interpretation. Context creation must also be optimized to yield good performance, and is done so by mapping activations to stack frames and lazily creating context objects, so called context-to-stack mapping. We discuss later in the next section how method lookup caches and context-to-stack mapping are used to handle forwarding objects.

4. Solution: The Partial Read Barrier

In this section we detail how the become primitive is performed in the Spur memory manager, and how message sending and primitive execution is used to lazily follow and eliminate forwarders. The become implementation in most cases creates forwarding objects in the heap, so we then discuss how forwarding objects impact the runtime system performance and lastly we discuss the static analysis that was required for our implementation.

4.1 Become Primitive Implementation

The become primitive does two things:

- edits objects so they forward to the objects they have become,
- ensures that no receiver and no method in method activations are forwarding objects.

Editing objects to forward. To forward an existing object `a` to another object `b`, the virtual machine:

- sets the class index and the format field of the object `a` to that for forwarding objects,
- sets the first slot of the object to point to object `b`.

One-way become is essentially the forwarding operation. When `a` becomes forward `b`, the primitive fails if `a` is an immediate object, otherwise `a` is forwarded to `b`.

Two-way become is a little more complicated. When `a` becomes `b`, which is symmetric, the primitive fails if `a` or `b` are immediate and then creates two copies of `a` and `b`, `a'` and `b'`, and forwards `a` to `b'` and `b` to `a'`, while copying the identityHash of `a` to `b'` and the identityHash of `b` to `a'`,[5] as shown on Figure 3.

[5] Effectively identityHashes are the property of object references, not objects. Any reference to an object `a` in some hash table still exists at a location derived from the hash of `a` after `a` has becommed `b`.

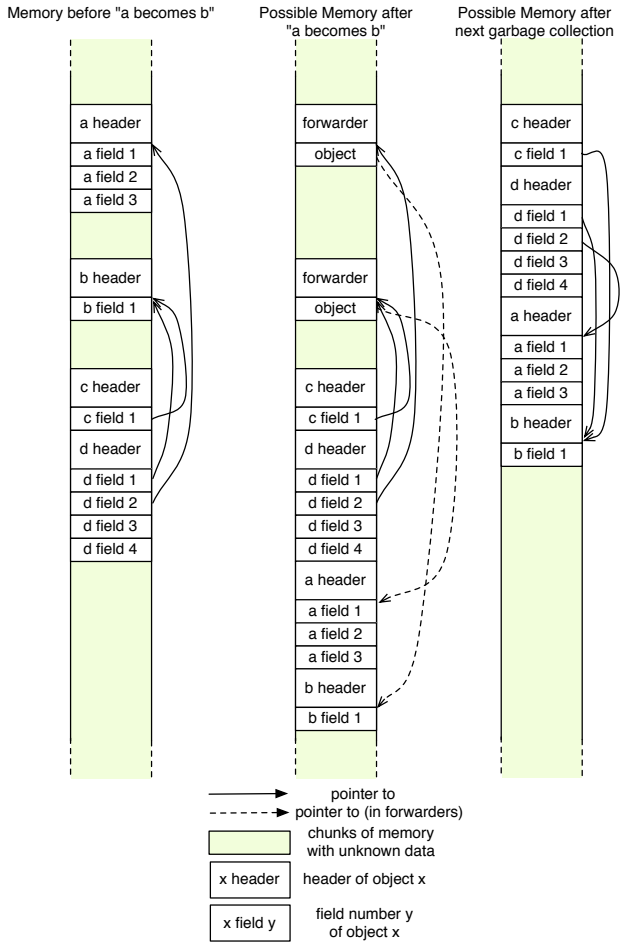

Figure 3. Two way become with forwarding objects

There are complications with one-way become and the class table. Since a class's identityHash is its index in the class table, we must arrange that if a class `classA` is now a forwarding object to the class `classB`, the class table has two entries for `classB`, one at `classB`'s identityHash and one at the forwarding object for `classB`'s identityHash. This is resolved by the garbage collector and by the allInstances primitive. To do so, both scan the class table looking for duplicate entries, and use a slower algorithm that dereferences class indices if a receiver occurs at multiple entries in the class table. This slower algorithm also updates the class indices of instances so that at the end the duplicate entries can be eliminated.

Replacing forwarding objects in the stack. A reference to a forwarding object is *unforwarded* when the reference is replaced by the object the forwarding object forwards to. It does not necessarily mean that the reference is replaced by the forwarding object's first slot because after multiple becomes one can end up with a linked list of forwarding objects. The unforwarding operation is therefore iterative,

reading each forwarding object's first slot until it finds a non forwarding object.

One of the main aspects of the solution consists of ensuring that the receiver and the method referenced by method activations being executed are not forwarding objects. To avoid performance overhead in common virtual machine operations (e.g. instance variable access), the become operation must ensure that each method activation has a direct pointer, and not a pointer to a forwarding object, to its receiver and its method. Essentially, the virtual machine needs to scan all method activations, and for each stack frame unforward its receiver and method if they are forwarded.

To understand how we can efficiently iterate over method activations, we'll describe briefly how the stack is managed in our virtual machine. Our implementation improves upon the Deutsch-Schiffman virtual machine (Deutsch and Schiffman 1984) implementation as described in (Miranda 1999).

The virtual machine maintains a small region for the stack organized as pages (by default 160 2kb pages). A new process starts its stack in an available stack page. Frame-building sends perform a stack check. When the end of a stack page is reached, the process asks for a new stack page and keeps growing its stack there. If no more stack pages are available, the method activations on the least recently used stack page are converted to context objects and written into the heap, freeing the stack page for the process to use it. When execution returns from the stack frame at the top of a stack page, a specific routine either resumes execution in the stack page with the the top stack frame's caller, or if it does not exist, converts the heap context object back into a method activation on an available stack page to resume the execution. Of course, the implementation has many details unrelated to this paper to ensure that calls and returns across stack pages are uncommon to avoid overhead. Stack page management may be a bit complex, but this is because it is not only used to handle stack page overflow but also to handle language-side stack manipulation, allowing for example continuations and exceptions to be implemented on the language-side and not in the virtual machine.

We call the area composed of all the stack pages the *stack zone*. The stack zone effectively holds the most recent method activations. This means that the VM can find recent stack activations by scanning the stack zone. The virtual machine does so after every become operation and unforwards the receiver and the current method in each stack frame when needed.

Now, some method activations may have been moved to the heap because a free stack page was required and no pages were available. Such activations are difficult to find because they could be anywhere in the heap. The become primitive ignores such activations, avoiding a full heap scan. Therefore an explicit read barrier is required when faulting context objects back into the stack zone. At this point the receiver and method are examined and unforwarded if required. But such activity is rare; only the presence of deep recursion or of many active processes can cause frequent stack page faulting, and the size of the stack zone can be set at start-up to make room for more processes.

Cases where both objects have the same size. For two-way become, our solution optimizes the case where a and b have the same size in memory by merely swapping the contents, excepting the identityHashes. This optimization is important because an existing database proxy framework using the become primitive was designed to become only objects of the same size. This framework was implemented in such way to avoid the full heap scan in the existing implementation of `become`, also by swapping the objects'contents, except identityHashes. Although we could eliminate the stack scan in this case we have yet to optimize this case.

4.2 Impacts on the Runtime

Theoretically, allowing forwarding objects forces the virtual machine to use a read barrier, which checks each time an object-oriented pointer (oop) is accessed if it is a forwarding object, following the forwarding pointer if so. Obviously, doing so at each oop read ends up being very expensive and a virtual machine would lose significant performance implementing such an operation. Therefore, we designed the read barrier in such a way that common oop reads do not need to check if the object being accessed is a forwarding object, and only performs the check in uncommon cases. The read barrier is partial. This subsection discusses how we avoid a read barrier in the common case.

Message sends (Virtual calls). Notionally sending a message involves fetching the object beneath the arguments, and searching the class hierarchy, starting at the receiver's class for a method that matches the send's selector. Once the method is found it is examined to see if it has a primitive. If it has a primitive the VM performs the primitive, and if it succeeds, the receiver and arguments are replaced by the primitive's result and execution continues. If the method has no primitive, or if the primitive fails, then notionally a new context object is created, receiver and arguments are moved from the sender context to the new context, and execution of the new method continues. Both operations must be heavily optimized to achieve good performance. We have sketched above how method activation is optimized; context allocation is eliminated by mapping activations to stack frames; contexts being allocated when activations overflow the stack zone, or when a block needs an outer scope, or when a stack frame is accessed as an object, for example by the debugger.

Sends themselves are sped up using a simple associative cache that stores methods and primitives keyed by message selectors and classes, called the first-level method lookup cache. In Spur class keys are replaced with classIndex keys. Of course the JIT uses inline caching techniques to speed up sends, but for the purposes of this discussion we'll consider only the first-level method lookup cache in a Smalltalk in-

terpreter. The handling of forwarders is essentially the same in each case.

Sends to forwarding objects will always fail the cache lookup because the VM never enters the class index for forwarding objects in the cache. Hence the read barrier is moved to the failure side of the first level method lookup cache probe. On failure, both the receiver's class index and the selector are tested to check if they are forwarding objects. If the selector is a forwarding object,[6] the virtual machine checks all literals in the current method and unforwards them if they're forwarding objects, including the selector. If the receiver is a forwarding object, the virtual machine checks all the literals of the method (since the receiver could have been a literal) and the contents of the current stack frame, and unforwards every forwarding object met. If any forwarding object is found the first level method lookup cache probe is repeated. [7]

Since the effectiveness of the method cache is good, for example worst case miss rates for a 1k entry method cache are around 4% in the interpreter and 1.2% in the JIT with inline caches, checks for forwarding pointers are relatively rare.[8]. Hence folding the read barrier into message lookup make it cheap and affordable.

Instance variable access and bytecode execution. The receiver and the method of the method activation being executed cannot be forwarding objects. We make sure of that because:

- When the become primitive finishes executing, the receiver and method field of stack frames in the stack zone are unforwarded if needed.
- If a method activation is on the heap when the become primitive is executed, then the virtual machine unforwards the receiver and the method when faulting back the method activation from the heap to the stack zone.
- Message lookup unforwards the receiver and the activated method if they are forwarding objects.

Therefore, since in Smalltalk only a method's activation can access instance variables of the receiver, instance variable access is performed without any read barrier.

In addition, bytecode execution (either through an interpreter or JIT compiled machine code) does not need to check if the method is a forwarding object.

[6] In Smalltalk, everything is an object including selectors, so selectors can be becommed too.

[7] Smalltalk experts will know that the language includes "super" sends that are potentially executed without checking the receiver, and will know that these have both low static and dynamic frequencies. Although the receiver of a super send is always the receiver, and therefore unforwarded when pushed on the stack when marshalling a supersend, the object could potentially be becommed later on during marshalling but before the actual send. We handle super sends with an explicit read barrier to unforward the receiver.

[8] These miss rates were measured during startup of a Squeak system, during which code is being freshly executed.

Global variable access. Global variables *do* require a read barrier. This could only be avoided if all machine code were scanned after become and if methods themselves where scanned when faulting objects back into the stack zone. This is far too expensive. But since global variables are represented by two slot objects,[9] it is very likely that the global's object header and its value share the same processor cache line. Therefore on current hardware the fetch of the class index to check for a forwarding object likely faults in the value slot, and so the cost of the read barrier is mitigated. In addition, even with the forwarding object check, the number of native instructions generated for global variable access is low therefore the overall cost of global variable access is low on production applications. Fortunately global variable access has much lower dynamic and static frequency than instance variable access and message send. [10]

Primitive operations. In Smalltalk a primitive operation is an operation implemented by the virtual machine, and is either an operation essential to the language's execution that the language itself cannot implement, or implemented to provide higher performance than the Smalltalk implementation. [11] Primitives are always associated with methods. A method has information in its header to inform the virtual machine if it has a primitive operation or not. When activating a method with a primitive, the primitive function is executed before the method's bytecode. If the primitive succeeds, the primitive returns a result, as if it were the result of the message send. If the primitive fails, it does so without side-effects and execution continues executing the method's bytecode, as if the primitive had not been present.

To fail without side-effects a primitive must validate any arguments and any state fetched from the receiver, before changing execution state by performing its operations. Validation involves any of testing for a specific class, testing for bit vs pointer objects, bounds checks, and recursively applying these tests to substructure of the receiver and/or arguments. For example, the primitive that installs a cursor examines the receiver to check that it represents a valid cursor object, comprised of two bitmaps, one for the image and one for the shape, plus a point to specify the cursor's hotspot. Provided that the sense of the validation tests are correct, they will fail for forwarding objects because forwarding objects have a class index and a format that are different to any other object. Hence, the addition of forwarding pointers does not add additional checks to primitive execution; it

[9] An association with a key slot holding the global's name and a value slot holding the global itself.

[10] a forwarding object to a global variable will be in the literal frame of a method. When that forwarding object is encountered, the virtual machine merely follows the reference. Eliminating the forwarder in the literal frame is left to the garbage collector, or, but unlikely, as a side-effect of following a forwarded send.

[11] as Smalltalk execution technology improves, the number of these so-called optimized primitives is declining.

merely folds checks for forwarders within existing validation checks.

Given that primitives will fail if they access forwarders, all that is needed is for the VM to know how deeply each primitive accesses the substructure of its receiver and arguments, which we call the primitive's accessor depth. On failure, [12] the virtual machine then traverses the receiver and arguments to that depth, looking for forwarding objects and unforwarding them if found. [13] If any object is unforwarded, the primitive is retried. The primitive accessor's depth is computed statically, as described in the next subsection (subsection 4.3).[14]

Read barrier moved to uncommon cases. As we discussed, the virtual machine needs to check if an object is a forwarded object only on look-up failure, on primitive failure, when a method activation is moved from the heap to the stack zone and on global variable access. All these operations are uncommon. Therefore, we have successfully implemented a partial read barrier that checks if an object is a forwarding object only in uncommon cases. As a result we achieve both a fast become and the compactness and speed of direct pointers.

We note that become can be used for much more than schema migration. If it is cheap, it is useful in many contexts, such as interfacing with object databases, where objects functioning as indexes into the database are faulted into the system when sent messages, and become into full objects. Indeed in the original object table implementations it is used to grow collection objects. The Squeak class library has been largely rewritten, introducing explicit indirections to containers such as arrays to avoid the overhead of become on growing; Spur may allow a return to that older, more compact, style.

4.3 Static Analysis: Finding Primitive Accessor Depths

The virtual machine holds a table mapping each primitive index with the function to run if they're encountered. We extended that table to hold the accessor depth of the primitive for each entry.

The depth stored is encoded as follow:

- A negative value means none of the primitive operands must be checked for forwarding objects.
- 0 means only the primitive operands need to be checked (the arguments on the stack).
- 1 means only the primitives operands and objects directly referred to by their fields need to be checked.
- etc.

The issue is how to determine the accessor depth for each primitive. In Cog, there are about 200 core primitives and more than 650 *plugin* primitives that extend the VM with access to the outside world, or providing optimized implementations of multi-media algorithms such as zip inflate/deflate. It is therefore tedious to analyze manually each primitive to find its accessor depth. In addition, a manual analysis would require the developer to compute the new primitives accessor depth for each new primitive he implements, requiring a new level of sophistication for the plugin author.

To avoid such inconvenience, we built a simple tool which constructs and analyzes the abstract syntax tree of each primitive, computing its accessor depth automatically. This tool is run statically as the virtual machine source is being generated, so it has no impact on runtime performance. This tool is constructed out of the standard machinery used to produce the VM. The VM is written and developed as a Smalltalk program, and its production version is produced by translating the Smalltalk for the VM into C by a Smalltalk program called Slang. Slang's parse trees for the primitives are those analysed by our tool.

4.4 Garbage Collection

Spur's stop-the-world garbage collector comprises a generation scavenger managing an eden and two survivor spaces (**?**), and a non-incremental mark-sweep collector (**?**). The garbage collector is aware of the presence of forwarding objects. The scavenger, of course, uses forwarding objects as part of its copy of objects to a survivor space and so forwarding objects in new space are eliminated by a scavenge. During the mark phase, when reading the fields of an object to mark all its referents, the mark-sweep collector unforwards references to forwarding objects. At the end of the mark phase, all forwarding objects are unreferenced and unmarked, and are collected in the sweep phase.

5. Performance

There are two main use-cases of the become primitive. The first case, emphasized in this paper, is schema migration; adding or removing instance variables from objects with a fixed layout. Schema migration is mainly used by developers for agile and exploratory programming. The second case is for proxies: `become` is used to create a proxy for an object and automatically switch all references to the objects to the proxy. `Become` can then be used again to remove the proxy (Martinez Peck et al. 2013; Martinez Peck 2012). Proxies are often used to access database objects transparently, loading them lazily from the database value when needed. [15]

In both cases, our main concern was system responsiveness as observed by the user when performing the become

[12] which the virtual machine must already check for to see if the method should be activated or the result returned,

[13] Primitives are never mutually recursive, therefore failures only have to consider a single primitive.

[14] Note that circular structures as primitive arguments do not pose a problem; given that traversal is to a fixed depth circular structures imply only that a given object may be visited more than once during the failure scan.

[15] Proxies are also used to implement distributed computation, catching and forwarding messages to remote machines; but unless objects migrate, become will not be used.

operation. The previous implementation, inherited from the original Squeak VM (Ingalls et al. 1997) requires a full heap scan, which is very costly in case of large heaps. Starting from heaps over a few hundred megabytes, users notice the pauses on modern machines.

The garbage collector theoretically also requires a full heap scan to garbage collect old objects creating an important pause. However, garbage collection is incremental and the heap scan is split into several tasks so users do not notice the pauses. Making `become` incremental is difficult as once it has been performed the variables that have been become are directly used by the program.

Our new solution, requiring only a stack zone, method cache and class table scan instead of a full heap scan, greatly improves the system's responsiveness while executing `become` on large heaps. The scavenger removes forwarding objects, without creating noticeable pauses. The and global mark-sweep garbage collector eliminates forwarding pointers, but *does* create noticeable pauses. But it is not run often and we intend to implement an incremental global mark-sweep garbage collector that will not create noticeable pauses as soon as possible.

Become performance according to heap size. For this first evaluation we created two objects of different size (one with no fields and another one with two fields). Evaluating become between objects of the same size is not as interesting because the object's data are just swapped in memory. Each object was referenced 10 times from 10 different random object in the heap. We evaluated the user pause on a MacBook pro with Mac OS 10.8, a 2.5Ghz processor Intel Core 5, 8 Gb 1600MHz DDR3 of RAM. We ran the operation 10 times and computed an average value to get an accurate result.

Heap size (Mb)	User pause with a full heap scan (ms)	User pause with forwarders and the partial read barrier (ms)
48	27.03	0.21
93	58.82	0.20
160	106.38	0. 22
210	138.88	0.21
283	188.68	0.20
335	222.22	0.19
521	344.83	0.21
816	542.03	0.20
1027	678.27	0.22
1433	949.49	0.21
1632	1081.04	0.22

Figure 4. User pauses according to become implementation

As we can see on Figure 4, the old implementation, requiring a full heap scan, had a performance proportional to the heap size. We stop the measurement below 2GB because the measurement were run in 32-bit mode. With a 500 Mb heap, the user pause for `become` is already at 345 ms. Nowadays, most servers have a larger heap than 500 Mb whereas they cannot allow themselves to answer a request after pauses greater than 345 ms due to their database proxies, so the performance was obviously not acceptable. The new implementation is not anymore proportional to the heap size, and requires only 0.2ms in any case. This does not delay too much a server response time any more.

By comparison, the VisualWorks Smalltalk implementation, which uses object table indirection, can perform become in about 1 microsecond on the same hardware. However, the object table indirection adds extra overhead to all object access. [16]

Become performance on objects with same size. As discussed in the previous section, in the case of a two-way become involving two operands with the same size in memory, no forwarding objects are created and the become primitive does not need to scan the stack zone. Using our new implementation of `become`, we measured the performance difference between the two cases.

Again, each object to be become was referenced 10 times from 10 different random object in the heap. Again, we evaluated the performance on a MacBook pro with Mac OS 10.8, a 2.5Ghz processor Intel Core 5, 8 Gb 1600MHz DDR3 of RAM. Again, we ran the operation 10 times and computed an average value to get an accurate result.

Become with operands having the same size	Become with operands having different sizes
0.0347 ms	0.1980 ms

Figure 5. Spur become performance depending on operands size

It happens that in this specific case (Figure 5), the become operation is 5.7 times faster. The implementation was worthwhile due to users using a specific proxy framework which creates proxies of the same size as the proxified object.

Evaluating the slow down. The new implementation also implies a slow down on message sends and primitive operations involving forwarding objects. Therefore, even if the user pause is less important, the overall program performance may still slow down. This slow down is difficult to measure because:

- forwarding objects are uncommon. A schema migration may migrate hundreds of instances, however, it still represents a small portion of live objects. On macro benchmarks, few message sends overall are performed on forwarding objects. The proxy implementation using be-

[16] We would like to compare the cost of object table indirection in VisualWorks against the direct pointer implementation in Cog Spur, but lack of time and the differences in the implementations of the VMs that make a direct comparison difficult, have resulted in us not including such a comparison.

come has the same property: few objects are proxified and unproxifed overall.

- forwarding objects are most of the time quickly unforwarded when encountered, removing their overhead at first use.
- a portion of the incremental garbage collection algorithm is performed at each scavenge and each time all the processes are idle. Therefore, the number of forwarding objects is constantly decreasing, removing their overhead.

Our implementation was deployed months ago, and we received no customer feedback reporting slow downs. We tried to run a suite of benchmarks, the benchmarks games (Gouy and Brent) and two additional macro benchmarks we commonly use to evaluate the performance of our VM, Richards[17] and DeltaBlue (Sannella et al. 1993), with the instances of a class used in the benchmark being become before the benchmark execution (for example in DeltaBlue, instances of Strength) to verify our customer feedback, but we didn't measure any noticeable overhead compared to direct objects. Any slow downs are further masked because Spur's object representation is an improvement over its predecessor (for example in the faster allocation of execution objects such as closures) so overall Spur is significantly faster than its predecessor.

To show the performance difference between a forwarding object and a regular one, we therefore built a micro benchmark sending a message to a forwarding object and a regular object, the method being executed in each case only returning the receiver. We ensured that the virtual machine could not quickly unforward the forwarding object at first access as it can do it in some cases. In this micro benchmark, we clearly saw that a message send on a forwarder is slower, as shown on Figure 6. Again, we evaluated the performance on a MacBook pro with Mac OS 10.8, a 2.5Ghz processor Intel Core 5, 8 Gb 1600MHz DDR3 of RAM. Again, we ran the operation 10 times and computed an average value to get an accurate result.

Message send on forwarding object	Message send on object
2,460,000 per second	60,400,000 per second

Figure 6. Message send on forwarders

On this benchmark, the message send was executed 94% slower when the receiver was a forwarding object. It is difficult to conclude from this result because in real application several scavenges or process idles could have happened between two executions of the message send, executing part of the incremental garbage collection and perhaps removing the forwarding object. In addition, the performance evaluated in micro benchmarks is always shadowed by noise, for example in our case by the actual execution of the method answering the receiver.

Now the implementation and the evaluation relied on the fact that the become operation is uncommon. If a programmer starts to use really extensively the become primitive, which is not the case on any real application deployed on our virtual machine, one would probably notice the slow down. In this case, we cannot really evaluate the performance because we cannot benchmark applications that do not exist.

6. Discussion

Become and incremental garbage collection. In the end, there is a clear connection between our become implementation and incremental garbage collection. An incremental garbage collector tries to split its algorithm in several chunks in order to limit user pauses and usually at the cost of an overall slower algorithm. We solved the same issue here: forwarding objects slow down a bit the system and are incrementally resolved by the GC whereas we removed the important user-pauses.

Languages with flexible layouts. Some modern high level programming languages, such as Javascript or Python, can dynamically add or remove instance variables for any object in the system. All the objects have a flexible layout, or, another way of saying it, are hash maps. One may think that our work, focusing mainly on schema migration, is not relevant for such languages, because our solution focuses on migrating objects with a fixed layout.

If one looks into the implementation of efficient Javascript or Python VMs, one will find that the virtual machine internally creates hidden *maps* for performance critical allocation sites. Hash maps have poor performance, and therefore such VMs needs to discover fixed layouts they can used for common objects to achieve high performance. In this context, when new code is loaded, creating when executed new instance variables to objects already belonging to an existing map, the same problem occurs: how to migrate objects with a fixed layout defined in an exsting map to a new layout with additional instance variables defined in a new map. Each instance needs to be migrated, hence our work may be relevant for such languages too.

Become and multithreading. There are few Smalltalk systems that are fully multithreaded. The common solution for a Smalltalk runtime is to provide green threads above a single threaded virtual machine and have a non-blocking foreign function interface (basically, being multithreaded with global interpreter lock), as in mainstream implementation of Ruby or NodeJS. To use several threads, a user has to call another runtime (which can also be a Smalltalk runtime) which uses another thread. The virtual machine we work on implements such a thread model, therefore we haven't faced concurrency problems in the become implementation.

[17] An operating system simulation benchmark (400 lines). The benchmark schedules the execution of four different kinds of tasks. It contains a frequently executed polymorphic send (the scheduler sends the `runTask` message to the next task)

As our implementation relies on a full stack zone scan, which can hardly be an atomic operation on mainstream hardware, the question of the become primitive in a multi-threaded environment is still very relevant. Current multi-threaded Smalltalks such as Smalltalk/X (Vraný 2010; AG) or the Roar VM (IBM Research) choose to stop all the threads in order to perform the become primitive.

7. Related Work

7.1 Implementations of Become in Other Smalltalks

We described in Section 2 the two other representations used in modern Smalltalk systems.

7.2 Hardware Support for Forwarding Pointers

Providing hardware support for virtual machine features was quite popular decades ago but it has now lost steam. The work we describe in this subsection informed us so we discuss it, even if this is old work and that such hardware support for virtual machine features is less relevant nowadays.

MIT, Symbolics, and TI Lisp machines (Weinreb and Moon 1981) had invisible forwarding pointers in hardware, similarly to our forwarding objects, so that a copying collector could move objects around while the program was running. That way, they didn't have to stop the running program to collect garbage. At that time, they thought that operations around forwarding pointers would be very expensive without hardware support. This research direction have been abandoned when non copying incremental garbage collection was invented (Baker 1992), avoiding entirely the problem.

Soar and SPUR (the Lisp-oriented RISC that followed SOAR) incorporated hardware support for the detection of intergenerational pointers. (That is, pointers from old data, which don't get garbage collected often, into new data, which do.) Previous work (Wilson and Moher 1989) showed why that's probably not the right thing to do, and why just keeping dirty bits is better. David Ungar has then improved this scheme by using bytes as dirty bits, because byte stores are cheap and fast on most architectures (Ungar et al. 1984).

In any case, these related works are not relevant to Spur, because they require hardware support which is unavailable on the standard processors that users of our virtual machine deploy application upon.

7.3 Future Work

Currently, global mark-sweep is stop-the-world. We want to add a Dijkstra-style tri-colour marking incremental global mark-sweep collector. Forwarding objects should allow us to compact incrementally too.

Now that the virtual machine is running with the implementation described, we'd like to combine it with an adaptively optimizing JIT. Guards, a common technique used in JIT compilers to ensure that an object has a certain class at a given point in the method execution or dynamically deoptimizing the active stack frame (described for the first time in (Hölzle et al. 1992) but the technique was named *guard* later), are compatible with the forwarding objects implementation: a guard testing a forwarding object will always fail as it has a specific class index.

The JIT compiler relies on the low frequency of `become` and expects none of the optimized methods' temporary variables to be becommed. Therefore, when one performs the become primitive, in addition to scanning the stack zone for receivers and methods that might have been forwarded, the virtual machine will need to scan all the stack fields of stack frames corresponding to optimized methods. If one of the temporaries has been becommed, the method has then to be deoptimized as if a guard had failed.

8. Conclusion

In Smalltalk, the become primitive exchanges pointer identity. It is used to implement schema migration in interactive development. We successfully implemented this primitive on a contiguous object representation using direct pointers without creating significant user pauses when the heap is large. The implementation relies on forwarding objects that when read conceptually answer the value of another object. Theoretically, forwarding objects require a read barrier each time an object is read in memory, but our implementation of forwarding objects makes it possible to check if an object is a forwarding object only in uncommon cases, avoiding the performance overhead of a full read barrier.

Acknowledgements

We thank Stéphane Ducasse, Stefan Marr and Gaël Thomas for their reviews of early drafts of this article. We thank Cadence Design Systems for supporting the work of the first author.

We thank Cadence Design Systems and Yaron Kashai for providing support for this project.

The writing of this paper was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the' Contrat de Projets Etat Region (CPER) 2007-2013' and the MEALS Marie Curie Actions program FP7-PEOPLE-2011-IRSES MEALS.

References

E. S. AG. Smalltalk/x official website. http://www.exept.de/en/products/smalltalkx.

H. G. Baker. The treadmill: Real-time garbage collection without motion sickness. *SIGPLAN Not.*, 1992.

A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in newspeak. In *European Conference on Object-oriented Programming*, ECOOP'10, 2010.

S. Cincom. Cincom smalltalk official website. http://www.cincomsmalltalk.com.

L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation*. Addison Wesley, 1989.

I. Gouy and F. Brent. The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/.

M. Guzdial and K. Rose. *Squeak — Open Personal Computing and Multimedia*. Prentice-Hall, 2001.

B. Hayes. Ephemerons: A new finalization mechanism. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, 1997.

U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*, ECOOP '91, London, UK, UK, 1991.

U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. .

V. U. B. IBM Research, Portland State University. Project renaissance: Harness emergence. http://stefan-marr.de/renaissance/.

D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 318–326, New York, NY, USA, 1997.

M. Martinez Peck. *Application-Level Virtual Memory for Object-Oriented Systems*. Theses, Université des Sciences et Technologie de Lille - Lille I, 2012.

M. Martinez Peck, N. Bouraqadi, S. Ducasse, L. Fabresse, and M. Denker. Ghost: A Uniform and General-Purpose Proxy Implementation, 2013.

E. Miranda. Context management in visualworks 5i. In *OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, Denver, CO, 1999.

E. Miranda. Cog blog. speeding up croquet and squeak with a new open-source vm from qwaq, 2008. http://www.mirandabanda.org/cogblog/.

M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Softw. Pract. Exper.*, 1993.

D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of soar: Smalltalk on a risc. In *International Symposium on Computer Architecture*, ISCA '84, 1984.

J. Vraný. *Supporting Multiple Languages in Virtual Machines*. Theses, Faculty of Information Technologies, Czech Technical University in Prague, 2010.

D. Weinreb and D. Moon. The lisp machine manual. *SIGART Bull.*, 1981.

P. R. Wilson and T. G. Moher. Design of the opportunistic garbage collector. In *Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, 1989.