TAIL CALL ELIMINATION IN THE OPENSMALLTALK VIRTUAL MACHINE

by

Matthew Ralston

Bachelor of Computer Science (Honours)

University of Windsor, 2004

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the program of

Computer Science

Toronto, Ontario, Canada, 2019

Author's Declaration

Abstract

Tail Call Elimination in the OpenSmalltalk Virtual Machine

Matthew Ralston
Master of Science, Computer Science
Ryerson University, 2019

Tail call elimination is used by languages and compilers to optimize the activation of methods in tail position. While this optimization has been the source of much research, it has not previously been implemented in the OpenSmalltalk virtual machine - the open-source Smalltalk virtual machine used by Smalltalk environments such as Pharo and Squeak. There are many approaches described in the literature to implement tail call elimination, such as removing stack frames on method activation instead of method return. Two implementations of tail call elimination using a stack frame removal approach are presented for Opensmalltalk VM. One implementation is presented for the Interpreter and one for the Cog JIT compiler. These implementations are tested with both ideal and real world scenarios and show improvements in execution time and memory usage.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

My thesis is that tail call elimination is a useful optimization for an object-oriented language such as Smalltalk. This is demonstrated in the form of two implementations using the open-source Smalltalk virtual machine OpenSmalltalk. One implementation is done in the Stack Interpreter, which is the bytecode interpreter component of the VM. The second implementation uses the just-in-time compiler (occasionally known as the Cog JIT compiler, or Cog VM) to implement tail call elimination in the generated machine code.

1.1 Problem Background

Before exploring the details of these implementations, it is worthwhile to fully explain what tail calls are, and why elimination of tail calls would be considered an optimization. This section will define what exactly a tail call is with examples drawn from Smalltalk. An introduction to Smalltalk syntax is included to explain the concept of tail call elimination. Next, we describe a special case of tail call elimination: tail recursion. Finally, we explain why tail call elimination is generally considered a valuable optimization and, in some circumstances, a necessary one.

1.1.1 Introduction to Smalltalk Syntax

Smalltalk is different enough from other programming languages to warrant an introduction to the syntax. A key difference between Smalltalk and other object oriented languages like Java is that, in Smalltalk, everything is an object, including integers, floats, booleans and characters. Individual objects are instances of classes and, like most object oriented languages, there are instance methods and class methods that an object has associated to it. In Smalltalk, methods are invoked by sending messages to objects. A message consists of a selector, which specifies which method the message is meant to invoke, and zero or more arguments. The object sending the message is referred to as the sender, and the object receiving the message is referred to as the receiver. If the receiver implements or inherits a method with the selector in the message, and the message includes the correct number of arguments, the corresponding method is executed. Like many other object oriented programming languages, objects can have instance methods invoked on them from their own class, or any class that they inherit from. Sending messages corresponds to calling methods or functions in other languages with some semantic differences.[1] However, when the term call is used when discussing other languages, it can be useful to consider them as equivalent to sends in Smalltalk. For a concrete example of message sending in Smalltalk, consider the following method:

```
Integer>>factorial
  self = 0 ifTrue:[↑1] ifFalse:[↑self * (self -1) factorial]
```

This method is an instance method for objects of class Integer that calculates the factorial for a number and returns the result. An example of how this method would be invoked would be as follows: `10 factorial`, where 10 is an instance of Integer. Note that this is a recursive implementation of the factorial calculation, as this method contains another send of the `factorial` message. The selector for this method, `factorial`, is an example of a unary message - it has a selector with no arguments. Walking through the method, the first message encountered is `self = 0`. This is a message send, as it sends a message with the selector `=` with one argument, 0, to `self`. The word `self` is a reserved word in Smalltalk and means a reference to the same object that is executing this method. The selector `=` is an example of a binary message. Binary messages take one argument and the selectors are typically arithmetic operators such as `+` and `-`. In this specific example, the binary message will be evaluated and a boolean will be returned. Next, the boolean result will have a keyword message sent to it, `ifTrue:ifFalse:`, with two blocks of code as arguments. Keyword messages are messages that have one or more keywords as a selector, with an argument for each keyword. Selectors for keyword messages are typically plain English descriptions of the purpose of the method, and each keyword is ended by a colon and followed by the argument for that keyword. In this case, the message `ifTrue:ifFalse:` is sent to the boolean result of the previous send. If the boolean is true, the first block of code will be evaluated, and, if false, the second

[1]For example, sending an unimplemented message to an object can be a recoverable situation in a Smalltalk program

block of code will be evaluated. Note that Smalltalk does not have a typical if/then/else statement. In fact, all control structures in Smalltalk such as conditional statements and loops are implemented by message passing, typically by providing blocks as arguments. Blocks are one or more lines of code that are surrounded by square brackets and are not immediately evaluated. Blocks also function as closures, and can reference variables from the enclosing method as well as have their own temporary variables. Blocks are also objects in Smalltalk, and can be passed into methods as arguments. Examining the first block of code, we see the value of 1 being returned. The Smalltalk syntax to indicate a return is typically a caret, and is stylized as "↑" in the listing. In the second block of the code, we see more messages being passed - the binary messages with selectors `-` and `*`, and a recursive send of the unary message `factorial`. It is necessary to understand the order of operations for message sends in a Smalltalk expression - unary messages are evaluated first, then binary messages and, finally, keyword messages. In the `ifFalse:` block, after evaluating `self - 1` in brackets, the `factorial` unary message send will be executed before the * binary message. Note that both blocks return a result. If there is no explicit return, a reference to `self` is returned instead.

1.1.2 Definition of a Tail Call

A tail call is a call that is the last instruction before a return of the top of the call stack - no further instructions within that block of code will be executed after the return. Another way to describe a tail call is to say the call is in tail position. Consider the following method in Smalltalk:

```
LinkedList>>at: index
  ↑(self linkAt: index) value
```

The above listing is a method with the keyword selector `at:` with one argument. It is a method for objects with the class `LinkedList` and is used to retrieve an element at a specific index within a `LinkedList` instance. The method contains two message sends - a message with the keyword selector `linkAt:` and a message with the unary selector `value`, which are executed in that order. At the end, the method returns the result of the `value` send. In order to clearly see the tail call pattern displayed in this example, it is useful to see the compiled bytecode for this method. Smalltalk was designed to be a platform-independent language which compiles to bytecode, and the bytecode then runs on a platform specific virtual machine.[2] For the purpose of this example, a typical call stack will be used. The following listing is the compiled bytecode for the `at:` method

[2]an idea adopted by Java 15 years later

presented above, with added comments to describe what each bytecode does and the operations performed on the call stack.

```
17 <70> self           "Pushes self, the current object, onto the stack"
18 <10> pushTemp: 0    "Pushes the argument onto the stack"
19 <E0> send: linkAt:  "Sends a message with the selector linkAt:"
20 <C9> send: value    "Sends a message with the selector value"
21 <7C> returnTop      "Returns the top of the stack"
```

First, a note about the format - the first column is the position of the bytecode in the compiled method - in this example, the first bytecode is at position 17, preceded by a header and an array of object references called literals which are not shown here. The second column is the actual hexadecimal representation of the bytecode, followed by the English description of the bytecode. Looking at the bytecode makes the order of message sends clear - first, a reference to `self` is pushed to the top of the stack. Next, the argument for `linkAt:` is pushed onto the stack. In this case, the argument is actually the argument passed with `at:`. In Smalltalk bytecode, arguments and local variables are referenced as part of the same pool of temporary variables - in this case, the argument is the first (and only) temporary variable at index 0. Next, a message with the selector `linkAt:` is sent, with the top of the stack as the argument, and the receiver of the message as the top of the stack minus the number of arguments (if the selector had no arguments, the receiver would be on top of the stack). Note that when `linkAt:` returns, the argument and the receiver will be replaced with the return result of `linkAt:`. Next, a message with the selector `value` is sent to the return result of `linkAt:` which, as mentioned, will either be an explicit return result or a reference to `linkAt:`'s `self` (in this case, the same `self` as this method). Finally, the top of the stack is returned, which in this case is whatever the return result of `value` is.

With the syntax of Smalltalk in mind, the definition of a tail call can be revisited which, as stated above, is a call that is the last instruction of a block of code, such as a method or function, before a return of the top of the stack. Recall that, for the purposes of this discussion, calls and sends will be treated as equivalent, and consider the send of `linkAt:`. This send is not in tail position as it is not followed by a return - when `linkAt:` returns, the result of that return is on the top of the stack. The second send of the message `value` is in tail position - it is followed by a return of the top of the stack (the return value of `value`). This can be more clearly demonstrated by showing the state of a hypothetical call stack - the actual implementation of Smalltalk call stack in the Cog VM will be explored later in Chapter 3. Unlike the actual implementation of the Smalltalk call stack, this hypothetical stack will grow upwards, so the top of the stack refers to the top-most entry in the visualizations. Another note about the visualizations is that a distinction will be made between the values pushed at method activation, and values pushed as part of method execution. The values pushed at activation will be called the stack frame, and will only be shown if necessary. The values pushed as part of method execution will be called the call stack. Both the stack frame and call stack for a specific method will be grouped in a bold outline. The state of the stack prior to calling the method `linkAt:` is represented in Figure 1.1. At the base of this stack is the stack frame for an unspecified method that is sending the `at:` message. Next is a reference to the object that will be receiving the `at:` message. On the top of the stack is an argument being passed with the `at:` message.

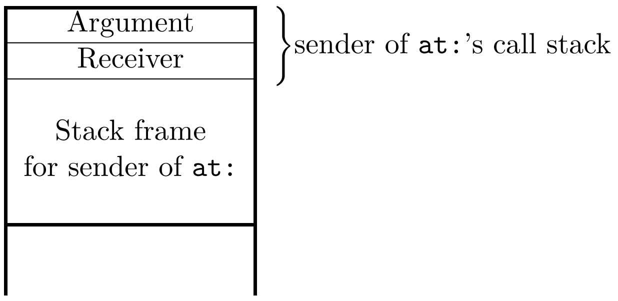


Figure 1.1: Stack before sending `at:`

When the method `at:` is called, the current instruction pointer is saved by being pushed onto the stack, and a new stack frame is built for the new method, which is identified through a combination of the selector and receiver. Skipping ahead to the state of the stack immediately before sending the `linkAt:` method, the receiver and argument for `linkAt:` have been pushed onto the stack. Refer to Figure 1.2. The details of the stack frame creation will be explored later.

When `linkAt:` is called, a stack frame will be created for `linkAt:` and the `linkAt:` method will be executed. Skipping ahead to when the method `linkAt:` returns, the receiver and argument for `linkAt:` are replaced with the return result and execution continues - see Figure 1.3. The call to `linkAt:` is not considered to be in tail position because execution of the method continues after `linkAt:` returns - the result of `linkAt:` is not immediately returned.

Now consider the send of `value`. There are no arguments for `value`, and the message is being sent to the return result of `linkAt:`, so there are no stack changes to demonstrate. Once again, a stack frame is built for `value` and, upon completion, the receiver of the `value` message is replaced with the return result of `value` - see Figure 1.4.

The next bytecode to execute in `at:` is the return of the top of the stack. No

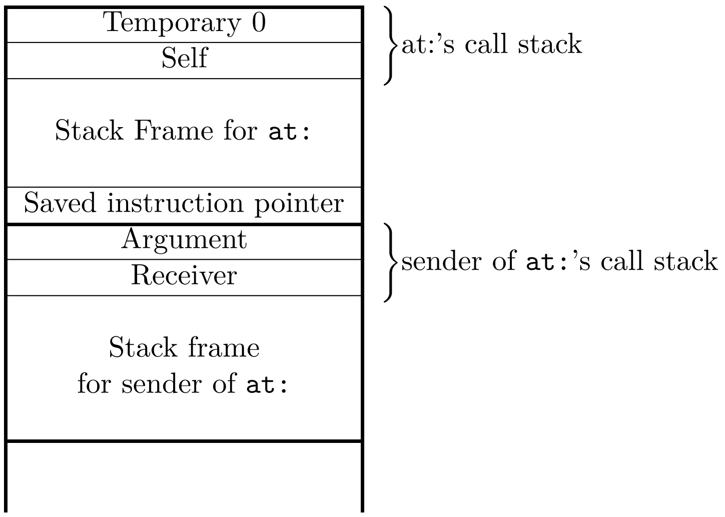

Temporary 0
Self
Stack Frame for at:
Saved instruction pointer
Argument
Receiver
Stack frame for sender of at:
at:'s call stack
sender of at:'s call stack

Figure 1.2: Stack before sending `linkAt:`

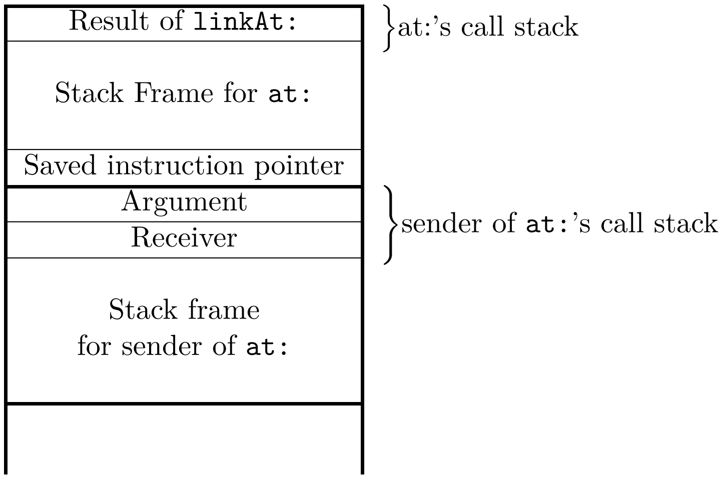

Result of linkAt:
Stack Frame for at:
Saved instruction pointer
Argument
Receiver
Stack frame for sender of at:
at:'s call stack
sender of at:'s call stack

Figure 1.3: Stack after sending `linkAt:` and before sending `value`

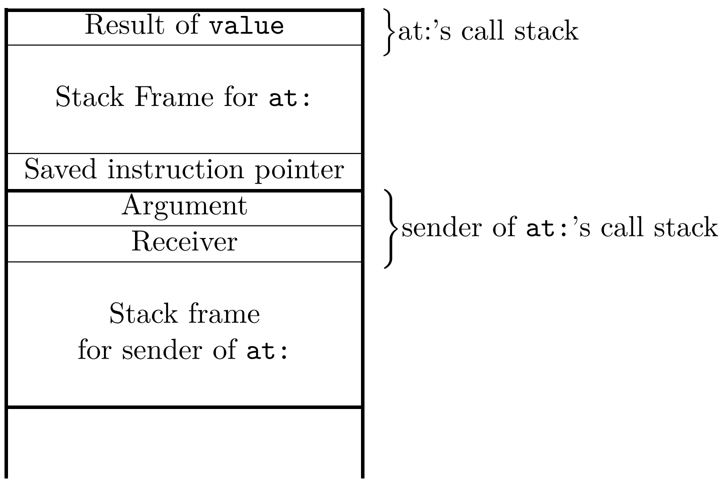


Figure 1.4: Stack after sending `value`

further manipulation of the return value of `value` will occur within the method `at:`, and no further use of anything local to `at:` is needed. The stack frame of `at:` can be dismantled and the top of the stack, in this case, the return result of `value`, can be returned as the result of `at:` and the saved instruction pointer is used to continue execution - see Figure 1.5.

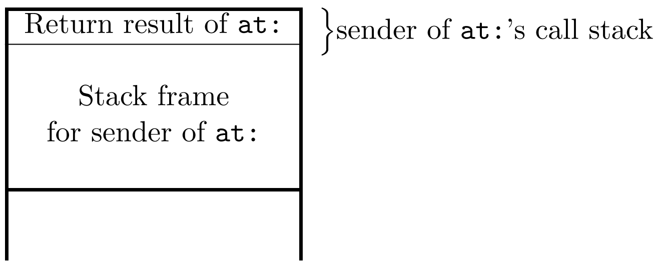


Figure 1.5: Stack after returning `at:`

Based on the above definition, the call to `value` was a tail call. It was last instruction of a method or function before a return of the top of the stack. When the call to `value` was made, no further use of anything local to `at:`'s stack was needed, and the return result of `value` was simply returned again by `at:`. This presents an opportunity for an optimization - could the stack of `at:` be dismantled before calling `value`, or even reused, preventing stack growth when tail calls are present? And could the return in `value` be directly to the caller of `at:`, skipping the execution of redundant instructions? Also, are there any common patterns in programming that naturally produce a large amount of tail calls? Attempts to answer these questions, and the optimizations that come from those answers, are known as Tail Call Elimination. Tail Call Elimination can therefore be defined as a research area that explores how to effectively change the behaviour of a call in tail position to eliminate unnecessary stack growth and return instructions.

1.1.3 Tail Recursion: A Special Case

A special case of a tail call that deserves mention is a tail recursive call. As the name indicates, this is a recursive call in a function that is in tail position, that is, it's a call within a function that calls itself. Note that not all recursive calls are tail recursive - we will demonstrate this by revisiting the implementation of factorial shown earlier.

```
Integer>>factorial
    self = 0 ifTrue:[↑1] ifFalse:[↑self * (self -1) factorial]
```

As a reminder, this is a method on an integer object, which would be called in the following fashion, where 10 is an Integer: `10 factorial`. Now, with the understanding of both Smalltalk syntax, and tail calls, it is clear that the send of the `factorial` message in the false case is not a tail call, as this send is followed by another operation (in this case, the send of *). The send of * is, in fact, a tail call and would be a potential candidate for tail call elimination if the send of * was handled like a typical method invocation[3]. Going back to the send of `factorial`, this means that, for each send of `factorial`, a new stack frame needs to be created for each recursive loop. Essentially, each new send of `factorial` sends the `factorial` message to itself after subtracting one, and waits for the return to actually multiply the results. What if there was a way to implement factorial without waiting for the return?

Consider the following factorial implemention:

```
Integer>>factorial
   ↑self factorialWithProduct:1.

Integer>>factorialWithProduct: product
    (self = 0)
     ifTrue:[↑product]
     ifFalse:[↑(self -1) factorialWithProduct:(product * self)]
```

This implementation does a couple of things differently from the previous. For one, the `factorial` method sends a different message, `factorialWithProduct:` with an argument of one. The method `factorialWithProduct:` is implemented slightly differently - instead of waiting for the return result of the recursive call to multiply to the total,

[3]Basic arithmetic operations tend to be optimized in Smalltalk implementations already and would not need to have a stack frame created

it does the multiplication before the recursive call and then sends the new total as an argument with the recursive send of `factorialWithProduct:`. In this implementation, the recursive call is in tail position, as no further operations are needed on the result returned by each recursive call. Tail call elimination could potentially optimize these calls, preventing a new stack frame from being created for each call.

1.1.4 Rationale for Tail Call Elimination

The rationale for optimizing calls through tail call elimination is simple. Eliminating tail calls opens up the possibility of reducing the number of instructions necessary to perform a call, and can prevent the growth of call stacks, which hypothetically can lead to reduced execution time and memory usage. In addition, many functional programming languages implement iteration via recursion as opposed to loops. Eliminating tail calls for, at least, tail recursive calls in those language implementation prevent unlimited stack growth. Despite being an object oriented language, Smalltalk also does not provide built-in loop constructs - loops are implemented in Smalltalk as messages that are passed blocks of code. While some simple loops are implemented by the compiler as typical loops in bytecode as an optimization, other loops are implemented as message sends. With tail call elimination implemented in Smalltalk, it may provide a benefit for these methods if they currently use tail calls, or provide an incentive to adopt a programming style that uses tail calls.

1.2 Objectives and Proposed Methodology

The objectives of this dissertation are as follows:

- Introduce tail call elimination by providing an explanation of what tail call elimination is and the research that has been done on implementing it.
- Introduce the OpenSmalltalk Virtual Machine by exploring the history of development in Smalltalk that lead to the OpenSmalltalk VM and discussing past and ongoing research in OpenSmalltalk.
- Having thoroughly explored the context of this research, present two implementations of tail call elimination in OpenSmalltalk - one in the interpreter and one in the JIT compiler

The proposed methodology is to build the two implementations presented in this dissertation, and then perform tests that compare execution time and memory usage with the existing implementation of OpenSmalltalk.

1.3 Contributions

The main contributions in this dissertation are:

- Designing and implementing an approach for tail call elimination in the Stack Interpreter;
- Providing experimental evidence that there are significant benefits to this implementation;
- Designing and implementing an approach for monomorphic tail call elimination in the Cog VM;
- Providing experimental evidence that there are significant benefits even with this limited implementation.

1.4 Dissertation Outline

This dissertation is organized as follows:

- chapter 2 presents background on the OpenSmalltalk VM, as well as previous work in tail call elimination.
- chapter 3 discusses two implementations of tail call elimination in OpenSmalltalk in detail.
- chapter 4 analyzes the performance of each implementation of tail call elimination with statistics comparing execution between tail call eliminating and non-tail call eliminating versions of the VM.
- chapter 5 presents the conclusions from the above and suggests future work.

Chapter 2

Related Work

This chapter will provide an overview of research in two different bodies of work. First, an overview of the OpenSmalltalk virtual machine will be given. This will cover historical developments in Smalltalk leading to the VM, as well as some recent and ongoing work. Secondly, tail call elimination will be discussed in two sections. First, a look will be taken at what sort of support there is for tail call elimination in other programming environments. Next, a look will be taken at what research has been done in tail call elimination - this will cover early influential work and what sort of techniques have been applied to implement tail call elimination in environments which do not support it.

2.1 The OpenSmalltalk Virtual Machine

The OpenSmalltalk Virtual Machine is an open source virtual machine for Smalltalk which is available for Smalltalk environments such as Squeak and Pharo. OpenSmalltalk uses features such as context-to-stack mapping, inline message caching, and JIT compilation for efficient message sending. This section covers the history of the Smalltalk VM with an eye towards message sending, development leading up to the OpenSmalltalk VM, and a detailed look at message sending in OpenSmalltalk to build an understanding of how tail call elimination may be introduced to the VM.

2.1.1 Early Development

The Smalltalk programming language was designed by Alan Kay at Xerox PARC. After several early revisions, the specification for the Smalltalk language, interpreter and a Smalltalk implementation of the interpreter were made available as Smalltalk-80 in what is colloquially called the Blue Book [15]. Method activation is implemented in this implementation of Smalltalk-80 using heap-allocated context objects - when a message is sent and a new method is activated, a new context object is allocated on the heap. Refer to Figure 2.1 for details of the context object. While this allowed Smalltalk-80 to achieve dynamic behaviour such as live debugging, this comes at the cost of performance and space, as context objects must be allocated and later garbage collected.

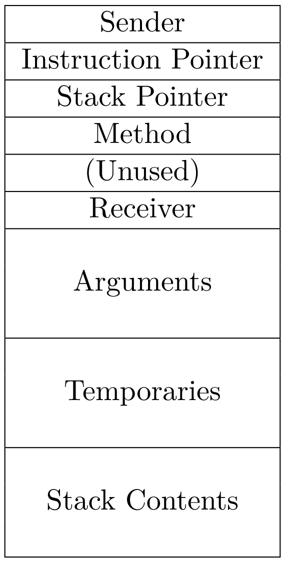


Figure 2.1: Smalltalk-80 Context Object

Deutsch and Shiffman presented a more efficient implementation of message sending in Smalltalk-80, with the goal of improving performance on conventional hardware while maintaining compatibility with existing implementations [12]. Rather than heap allocating all contexts, contexts are instead allocated onto a stack first. If no reference is made to the context as an object, the context only ever exists as a stack frame - this is called a volatile context. If a pointer is generated to a volatile context, a block of memory reserved for a context is allocated and its address is stored in the stack frame - this is called a hybrid context, as it exists on the stack but also has an (as of yet unpopulated) object representation on the heap. Once a message is sent to the hybrid context, it is converted to a full context object, where it can respond to the sent message - this is a stable context.

Deutsch and Shiffman also implemented inline caching of method addresses. While Smalltalk is a dynamically typed language, the observation has been that message sends often call the same method on objects of the same type, so the address of the method is cached at send sites. They also presented a JIT compiler, or, in their terminology, dynamic translation of virtual code to native code at runtime. These optimizations serve the basis of optimizations seen in the OpenSmalltalk VM [28, 23].

Squeak is an open source implementation of Smalltalk-80, which was released in 1996. The creators of Squeak were looking for a portable, open development environment for non-technical people and, finding no open Smalltalk environments, chose to build it themselves and share it with the community [17]. The design of Squeak differs in a few ways from the original Blue Book implementation of Smalltalk-80, as a new object and image format were designed for Squeak. The object memory was redesigned for a 32 bit address space and to use direct pointers instead of indirect pointers and an object table. The headers for objects themselves were redesigned to be variable length, where information about the object was encoded in either one, two or three 32-bit words. Garbage collection had to also be redesigned for the new object format.

One of the goals of the developers of Squeak was to be able to develop and simulate the virtual machine using Smalltalk, but also be able to build for a target platform using C. This required the creation of a translator from Smalltalk to C. To avoid having to emulate all of Smalltalk, only a subset of Smalltalk was chosen to be translated (later called Slang), so blocks, message sending, and objects are not directly translated. Methods can also be inlined when translated to C - this allows them to be kept as separate methods in Smalltalk for ease of development without adding overhead of procedure calls in C.

2.1.2 The Cog VM

The Squeak framework for building VMs, VMMaker, was extended by Miranda into the OpenSmalltalk VM, which added, amongst other improvements, an extended bytecode set, full closures, context-to-stack mapping, a JIT compiler, and inline caching [27]. Full closures were implemented first, replacing the implementation that was currently in Squeak and based on the Blue Book Smalltalk-80 implementation [24]. Closures, or blocks in Smalltalk syntax, were originally not implemented as proper closures, as they lacked their full activation record to store their own arguments and temporary variables - attempting to create blocks with recursive calls would fail due to this implementation. Miranda corrected this implementation by ensuring each activation of a block would have its own activation record, and ensuring that values of temporary variables defined in the enclosing method but referenced within a block would be independent from references within subsequent activations of the block. Bytecodes for the new closure implementation were added to the existing Squeak bytecode set [25].

Context to stack mapping was implemented next as a precursor to adding a JIT compiler - this class of the OpenSmalltalk project is called the Stack Interpreter [28].

The Stack Interpreter is a reimplementation of method activation that attempts to make the process more efficient by mapping heap allocated context objects to a call stack. As mentioned earlier, Blue Book Smalltalk-80 implemented activation records as first class heap allocated context objects which allowed Smalltalk-80 systems to have advanced features like live debugging and exception handling. Each context contains a pointer to the sender, a pointer to the current instruction, a stack pointer, a reference to the compiled method, arguments, temporary variables and a call stack for that context. Refer again to Figure 2.1 for details. Bluebook Smalltalk-80 presented an inefficient implementation of method activation where the new context object is allocated on every send. This is doubly inefficient as, not only is there a context object allocated on each send, but eventually that context needs to be garbage collected as well. Squeak improved the implementation by allowing unreferenced context objects to be reclaimed when a new method is activated. As mentioned before, Deutsch and Schiffman presented a scheme for mapping context objects to a call stack and Miranda developed a variation of that scheme for the Stack Interpreter.

In the Stack Interpreter, a call stack is implemented in a manner similar to a language like C. When a message is sent the receiver and arguments are pushed onto the stack and then a stack frame is built by pushing the current frame pointer on to the stack, and setting the stack pointer to be the new frame pointer. When a method returns, the stack pointer reverts to the value of the frame pointer, and the frame pointer pushed up earlier is set as the new frame pointer. The difference in implementation between the stack interpreter and a language like C is that there needs to be a reference to a heap-allocated context object - these context objects are allocated only when needed, such as when a block closure is created and needs a reference to its enclosing method context, or when engaging in some higher level activity like debugging. When a method is activated on the call stack, an empty slot is pushed onto the call stack. This is replaced with a context pointer, if the context object is created. The Stack Interpreter virtual machine only allows for one context object to exist per stack frame. Unlike Blue Book context objects, in the Stack Interpreter there is a single shared call stack and there needs to be a mechanism for managing pages of stack frames. Because Smalltalk context objects are heap allocated, Smalltalk implementations typically have been immune to stack overflow - there can be as many method activations as there is memory available for the heap. In the Stack Interpreter there is a set stack limit, and on each method activation that stack limit is checked to see if it has been reached and a new stack page has to be built. One interesting aspect of implementing this is support for non-local returns in Smalltalk; this was challenging to implement as a non-local return needs to return to the activation record of the method in which it was defined, and which may need to be returned across various different stack pages, stack allocated activation records, and heap allocated context objects.

Having laid the groundwork with the Stack Interpreter, the Cog JIT compiler was the next piece added to OpenSmalltalk [23]. The OpenSmalltalk VM with JIT compilation is sometimes referred to as the Cog VM. The Cog VM consists of two major components - first, there is a version of the Stack Interpreter (CoInterpreter) that functions like the Stack Interpreter but is also able to invoke the JIT compiler and transfer execution from the interpreter to JIT compiled code and back again as needed. Second, there is the Cog JIT compiler, which is responsible for generating platform specific assembly code. The JIT compilation process itself first translates Smalltalk bytecode to an intermediate representation using a generic assembly language, before being compiled to a platform specific assembly language with real memory addresses. Execution jumps back and forth between interpreted code and JIT compiled code using a series of trampolines (jumping from the interpreter to JIT compiled) - interpreted code and JIT compiled code actually share the same call stack, and when execution jumps back and forth, the stack pointer and frame pointers are copied from registers used by the JIT code to variables used by the interpreter and back again as the case may be. There are also three levels of inline caching of message sends that are used in the Cog VM - monomorphic, polymorphic and megamorphic. Monomorphic inline caches are caches of sends to a specific method of a specific class of object - caches are implemented by caching a class tag at the send site. Polymorphic sends are sends to methods of different classes with the same selector - caches are implemented by sending to a lookup routine with the class tags and call addresses of each method. There is an upper limit of how many methods can be in a polymorphic inline cache, and, if broken, the send site is considered megamorphic and the cache goes from a closed polymorphic inline cache to an open polymorphic inline cache. A detailed look at method invocation and inline caching in the JIT will be given later on, as the implementation presented in this thesis will involve changing the JIT compiler.

As mentioned above, one of the important requirements of development of this series of VMs since Squeak was to be able to develop and simulate the VM within Smalltalk. With a JIT compiler, this means that generated assembly code for a specific target platform also needs to be simulated within Smalltalk [26]. While the hardware of the development platform could be used, it would mean that development would be tied to the CPU platform of that hardware, and that incorrectly generated JIT code would be running on the development machine, potentially causing crashes. The solution taken by Miranda was to simulate the generated assembly code within Smalltalk using existing emulators for the target platform and building a Smalltalk FFI (foreign function interface) to communicate with the emulator. For x86 and x64, the emulator Bochs was used [1].

When discussing Squeak, it was mentioned that the Blue Book Smalltalk object format was replaced for Squeak with a new 32 bit format which used a variable length object header made up of one to three 32 bit words, depending on the class. There is a known set of compact classes, for whom objects use one 32 bit header word, otherwise another 32 bit word is added for a class pointer. If an object has more than 255 fields, another 32 bit word is added to the header. This adds complexity to inline caching and object instantiation in Cog, so the object header format was revisited with a project called Spur [22]. The object header format is now one consistent 64 bit format, for both 32 bit and 64 bit implementations [6]. To reduce the size of the header, the header no longer contains a pointer to the class as this would need to be 64 bits in a 64 bit environment. Instead, the header contains a class index, and a class table is maintained to look up classes. The saved space allows objects to have a bigger identity hash, reducing collisions, and the consistent header size saves complexity. The class indices can also be used when inline caching, which will be looked at in more detail later.

The Spur object model also allows for better support of live programming in an object oriented environment where instances of objects can exist at the same time as the code for the object is being modified [29]. Smalltalk implements this feature with two primitives - one, called `allInstances`, to collect all instances of a class, and one, called `become:`, that exchanges the identities of a pair of objects. Become has four different forms: one way, in which `a` becomes `b` means references to `a` become references to `b`, two way, in which `a` becomes `b` means references to `a` become references to `b` and references to `b` become references to `a`, and bulk versions of one way and two way that operate on arrays. Blue Book Smalltalk-80 has indirect access to objects - an object pointer points to a header which then points to the object, so become is simply a matter of changing which object the header references at the cost of having a pointer to an object in every object header. However, Squeak's original object format used direct pointers, so become required a scan of all references to the object through the heap to replace references. With Spur, which uses direct pointers, a scheme to lazily implement become was presented by Miranda and Béra [29]. Existing instances of objects are edited to become forwarding objects - their class index is changed to that of a forwarding object and the first slot in the object is a pointer to the object it becomes. In a two way become,

[1]http://bochs.sourceforge.net/

copies need to be created and the forwarding object forwards to a copy of the object it became, as the original object it should have become also turned into a forwarding object. References to forwarding objects are replaced with references to the actual object lazily, as they are referenced. Checks for forwarding objects are minimized to certain cases to avoid unecessary checking for forwarding objects (a read barrier) by preventing their use in certain cases - for example, they can't be the receiver or the method of an activation to prevent a read barrier on instance variables. Forwarding objects also cannot be saved in a method lookup cache, so they will fail lookup and be unforwarded.

Béra also introduced read-only objects into the Cog VM [4]. Introducing read only object could involve an extra check, or write barrier, on each store into an object - if the object is read only, the store would fail, and one of the requirements was that failure should be handled at the language level. However, this would add overhead on each store to an object, as each store would need to check for read access. The read only status of an object can change as well, which is why the terminology write barrier was chosen as opposed to immutability. Support for this was added to the virtual machine, as many common operations in Smalltalk are implemented as primitive operations, which are handled by the VM and not exposed to the Smalltalk image. Changing the behaviour of these to enforce read only behaviour would be difficult to implement just in the image. Also, some objects are not allowed to become read only, such as context objects, as execution may rely on being able to change these objects. Implementing this change required three modification to the VM: the object representation, the interpreter and the JIT compiler. In the object representation under Spur, a flag is added to the header of the object to indicate that an object is read only - this flag is modified by calling a specific operation from Smalltalk code that invokes a primitive operation. Fortunately, object headers under Spur left a few bits unused, so modifications like this can be done. The interpreter was modified to handle the case where access to a read only object could fail, including any primitives that access objects. Checks needed to be added in primitives and in stores to instance variables. Similar modifications needed to be made to JIT compiled primitives and instance variable stores. The overhead of checking at every store is mitigated by adding this check to existing checks on behalf of the garbage collector, which allowed this to be implemented with minimal overhead.

2.1.3 Other Work in OpenSmalltalk

Béra and Miranda introduced an alternative bytecode set to add support for adaptive optimization into OpenSmalltalk [7]. The adaptive optimizer being designed was to be implemented in Smalltalk above the virtual machine, and needed to extend the existing bytecode set for new operations. However, the existing bytecode set lacked room to add new operations, with only three unused bytes. In addition, it had a few issues which could be solved as well, of which a few examples will be given. The existing bytecode set had a limit on the size of arguments - for example, the jump bytecode was limited to only be able to jump 1024 instructions forward. This would actually cause compilation failures and restrict the amount of code that the compiler could inline. Another issue was how primitives were implemented - each primitive has an index, and the index of the primitive is encoded in the method header in such a way that inlining a primitive is impossible, which prevents operations implemented as primitives from being as optimized as possible. The bytecode set also had some issues related to legacy support, such as support for 16 bit Squeak versions and support for older hardware - the bytecode set could be redesigned to remove this legacy support.

The new bytecode set (Sista) introduced some features, some of which are general improvements and others of which are added to support the adaptive optimizer. An extension bytecode was added, which allows for arguments to be infinitely extended - this removes limitations on argument size such as the jump forward example given before. This also allows a Smalltalk method to have more than 255 literals (literals being things like selectors), as literal bytecodes can also be extended. Primitive indices have been moved to bytecode instead of the header, and the primitive bytecode can be inlined to improve performance. 15 bytecodes were also left available for future implementations.

To facilitate compatibility, the new bytecode set was added to the VM using the VM's support for multiple bytecode sets. The VM already had the ability to support multiple bytecode sets, as it supported bytecode for both Smalltalk and Newspeak. This allowed for execution of multiple bytecode sets in the same runtime without having to build a converter for an image compiled in one bytecode set or another. Compiled methods in Smalltalk are a special type of object with their own header format, and a flag bit is included at the end of the method header to indicate which bytecode set it uses - this bit is checked before method execution to determine which bytecode set to use for interpretation. Support for multiple bytecode sets will be briefly revisited later, as it was experimented with to implement tail call elimination in the Stack Interpreter.

Béra also introduced a potential replacement for the BlockClosure implementation, discussed earlier, called FullBlockClosure with a new bytecode in the Sista bytecode set [5]. The original implementation of BlockClosure required that an outer context object exist for the enclosing method, however, the new implementation only requires an outer context to exist when the block has a non-local return. FullBlockClosures also have their own method (instead of just referencing their enclosing method).

2.2 Tail Calls in Different Environments

Tail call elimination is a well known optimization and is implemented as a part of many languages, whether that is as a requirement or as an optimization. This section provides a breakdown of tail call elimination support in languages which have been chosen either for their relevance to the history of tail call elimination or for their popularity. Note that adding tail call elimination support for environments which do not support it is the source of much interest (including this thesis) - attempts to add tail call elimination and the methods employed will be discussed later in this chapter.

2.2.1 Scheme

Scheme, a programming language based on Lisp, is particularly noteworthy in tail call elimination research, as general tail call elimination was first presented as part of the development of the Scheme language [37]. Scheme requires implementations to be properly tail-recursive as part of its specification [43]. The definition of properly tail-recursive as presented in the specification encompasses both tail recursive calls and general tail calls as well, not just recursive, or self, tail calls [10]. As a language where tail call elimination is required, Scheme compilers are a source of much research in different strategies for tail call elimination, which will be reviewed later in this chapter.

2.2.2 Javascript

Javascript is an implementation of the ECMAScript specification. The ECMAScript 6, or 2015, specification introduced proper tail call elimination into the specification when using strict mode [13]. The language definition defines what tail calls are considered and an abstract operation is provided which handles the tail call. However, support at the time of this writing is limited, with Safari/Webkit being the most notable inclusion [14].

2.2.3 Java Virtual Machine

The Java Virtual Machine, or JVM, is the specification for virtual machines to run the Java programming language, as well as numerous other languages that target the JVM. Tail call elimination was originally not included as part of the JVM due to security methods that relied on counting stack frames - as tail call elimination may change the number of frames, this security implementation prevented tail call elimination [11]. While this security implementation has been replaced with something more robust, tail call elimination has yet to be implemented. This creates a problem for functional languages which rely on tail call elimination for, at the very least, tail recursive calls to implement looping in a functional paradigm. The approach taken by some languages is to add special syntax to indicate tail recursive calls to the compiler. Examples of languages designed for the JVM use this approach include Clojure, Scala, and Kotlin [42, 44, 20]. Compiling functional languages to the JVM is of some significant interest, and some approaches will be looked at later in this chapter.

2.2.4 Common Language Runtime

The Common Language Runtime, or CLR, is Microsoft's virtual machine for .NET applications programmed in languages such as C# and F#. The specification for the CLR is laid out in the Common Language Infrastructure (CLI) specification and supports a specific tail prefix to call instructions -this indicates that the stack frame of the current method can be eliminated before the call is executed [36]. Implementations are only required to honor the tail prefix for call instructions for which the targets are known at compile time. F# uses the tail call capabilities laid out in the CLI for tail-recursive calls, but does not perform general tail call elimination [39]. The Roslyn compiler for C#, however, never emits the tail instruction [33]. Regardless of whether the individual languages emit the tail instruction, the JIT compiler for CLR on AMD64 platforms can opportunistically eliminate tail calls [18].

2.2.5 Python

Python supports neither tail call elimination or tail recursion elimination - this is by design for several reasons discussed in the context of tail recursion elimination [34]. First, as tail recursion elimination removes stack frames, stack traces will not have the eliminated stack frames. This will make debugging more difficult. Optional tail recursion elimination to preserve debugging stack frames was also not considered due to a desire to always default to useful debugging. Second, if tail recursion elimination is introduced as a choice for implementers, programs may be written that rely on the presence of tail recursion elimination and not run on implementations that choose not to implement it. Third, Python also allows functions to be rebound at runtime, so implementing tail recursion may be difficult to accommodate in all cases.

2.2.6 Other

The LLVM compiler infrastructure, used in the clang C compiler amongst other compilers, has optional tail call elimination depending on the calling convention of the caller and the callee (typically preferring calling conventions where arguments are passed in registers). Tail call elimination is implemented by having the callee reuse the stack of the caller for calls in tail position. When the callee has more arguments than the caller, the convention is that the callee pops the arguments [21].

2.3 Research in Tail Call Elimination

Steele presented tail call elimination implemented as a compiler optimization of machine code while discussing the expensiveness of procedure calls compared to goto statements [37]. Steele was demonstrating several points in comparing procedure calls to goto statements in order to show that procedure calls were not necessarily too expensive and not expressive enough compared to goto statements. When demonstrating expensiveness, a simple Lisp method was shown compiled to a machine language with a small set of instructions: JUMP, PUSHJ, and POPJ. JUMP jumps to an address, PUSHJ pushes a location onto a call stack and then jumps to an address, and POPJ pops a location off of a call stack and jumps to that location. The machine code representation of a procedure call is shown to be a PUSHJ operation in that it pushes the current instruction and then jumps. The machine code representation of a return at the end of the procedure is shown to be a POPJ operation - it pops the previous address off the call stack and jumps. Steele shows that the final procedure call in the procedure can actually simply be executed as a JUMP without pushing the address (with a PUSHJ) and without needing the POPJ instruction for the return. Procedure calls can be optimized in this way within the branches of conditionals, looping constructs and the last component of a block, amongst others. Steele recognized this as a universal technique for optimizing procedure calls as opposed to tail recursive specific optimization done in earlier work.

Hanson builds on Steele's work from a compiler perspective [16]. Hanson presented a technique for compiling tail-recursive languages using stack allocation. Hanson uses Scheme as the example language, as it was a widely known tail recursive language - this is using the Scheme definition of tail-recursive, which refers to general tail call elimination and not just elimination of self tail calls. To set the context for Hanson's implementation, an activation record for procedure calls was presented. This activation record consists of four types of fields: arguments of the called procedure, the return address, temporary variables, and an access link which is a pointer to the activation record of the lexical parent of the called procedure. Note that a control link, or a link to the activation record that invoked the current procedure is not included - Hanson states that MIT Scheme does not require this.

First, Hanson shows a traditional non tail recursive implementation of procedure calls and one-to-one stack growth with each procedure call. To present a tail recursive implementation, Hanson recognized that some information needs to be removed from the stack. Hanson broke activation records down to two parts in order to facilitate discussion: control records, which consist of the return address, and environment records, which contain bindings and the access link. Hanson also classified procedure calls into two categories: subproblems, which are procedure calls where the calling procedure will continue execution after the subproblem returns, and reductions, where the calling procedure will not continue execution after the return. Tail recursion is then defined as avoiding the pushing of unnecessary control records and popping the environment records when they are not needed.

Reductions have unnecessary control records, as there is no need to use the return address in the reduction when there will be no further execution in the calling procedure. Control records for subproblems are needed, as execution does continue in the calling procedure. Environment records can be discarded when the variable binding and access links cannot be reached and are therefore no longer needed. These environment records are popped in two cases - when a reduction is called, and when a return sequence happens from a reduction. When a reduction is called, the environment records that are popped are ones which do not have a return address, and are not the lexical parent of the reduction (and therefore the reduction would have no need to reference its values). Returning from a reduction pops the environment record of the reduction, and all remaining environment records up until a control record is reached, at which point execution jumps to the return address of the control record.

Hanson recognized the challenge in implementing this, as the runtime needs to be able to distinguish between environment records and control records, and there needs to be added instructions for determining which records to pop during reduction calls and return sequences. Building on earlier work, Hanson presents an implementation where a control link is reintroduced, implemented as a register, as a control link points to a control record in the previous activation record. In the tail recursive implementation, the control link points to the topmost control record, thus providing a quick reference. When calling a subproblem, the contents of the current control link register is pushed onto the stack, and the stack pointer is copied into the control link register before the rest of the activation record is pushed. When a reduction is called, the stack pointer is set to either the control link register, or the access link, which pops any environment record without a control record and which the reduction has no reference to. When returning, all of the activation records above the topmost control record can be discarded by replacing the stack pointer register with the control link register. This simplifies the execution of proper tail call recursion while adding the cost of a control link register, and pushing the value of the control link register on subproblem activation. Added efficiency can be gained by predicting the value of the control links and access links as offsets of the stack pointer, which Hanson presents a few strategies for. Hanson leaves any actual performance evaluation of these techniques to future work.

Kelsey presents several methods for implementing proper tail recursion in a stack-based interpreter [19]. Kelsey first presents the issue as to why a traditional call stack doesn't provide proper tail recursion. As discussed elsewhere, when a function calls another function in tail position, the environment (variables, access link) for the calling function is left on the stack despite never needing to be referenced again. Kelsey then considers three general approaches to dealing with this issue. The first is to allocate all environments to a heap, and then later garbage collect any unused environments. The second is to overwrite the existing environment with the new environment, replacing existing arguments with new arguments. Third, the stack could be allowed to fill with environments, and then later be garbage collected. Three garbage collection strategies for the stack were looked at as well - compacting the existing stack, copying the stack to a new stack and copying it to a heap. Looking at the three different approaches for implementing tail recursion, as well as not using a stack at all, Kelsey looked at the costs of making continuations, making procedure calls and garbage collection. Kelsey found that the second approach worked best in general for languages except Scheme.

Benton, Kennedy and Russell presented a compiler for Standard ML to Java bytecode which, amongst other things, implemented tail recursion elimination [3]. The compiler converts tail recursive calls into goto instructions, but does not perform general tail call elimination, as the authors believed that general tail call elimination would be added to JVMs (as mentioned before, this has yet to happen). The compiler first converts an ML program into one large term in a typed intermediate language, called the Monadic Intermediate Language (MIL), before being translated into a low level code called Basic Block Code (BBC). This is then converted into Java class files. For tail recursive calls in the sense of self tail calls, the tail call is ultimately compiled as a goto bytecode for the JVM. The authors suggested two possible techniques for general tail call elimination if tail call elimination was never included in the JVM - one was to place tail calling and callee functions in the same method (so that tail calls could still be replaced with gotos), and the other would be to use the tiny interpreter (essentially a trampoline) in the Glasgow Haskell compiler [31].

Bothner presented a compiler for dynamic languages such as Scheme on the JVM, based on the Kawa Scheme interpreter [8]. The implementation that the author presented supported almost all features of Scheme at the time. The author considered writing a Scheme interpreter directly in Java, or writing a compiler to translate the language into Java source, but abandoned these ideas in favour of compiling Scheme to Java bytecode. The overhead of an interpreter is avoided by compiling to bytecode first, and JVM bytecode was considered by the author to be more expressive than the Java language itself, such as having a goto instruction (which Java does not). The author only implemented optimiztation of self-tail calls by replacing the calls with gotos for the two standard Scheme looping forms - do and named-let. Without having direct access to stack frames to implement general tail call elimination, the author presented a proposed implementation of a framework for implementing continuation passing style in Java, which would put tail calls inside of a switch statement.

Peyton Jones, Ramsey and Reig designed an intermediate assembly language named C– that provided tail call elimination [32]. The motivation for writing this language was to produce a portable high quality assembly language. The authors reject C as an intermediate language for various reasons - relevant to here is the lack of easy implementation of tail call elimination. In C–, tail calls are treated as jumps from which control does not return. An implementation would be required to disassemble the caller's stack frame before the jump.

As mentioned earlier, one of the reasons that the JVM specification originally did not include tail call elimination was because of a stack checking security mechanism. Clements and Felleisen presented an abstract machine with general tail call elimination that also has security stack inspection, which they claim invalidates the belief that tail call elimination and stack inspection are incompatible [9].

Tauber et al. presented FCore, which is a JVM implementation of System F, a typed lambda calculus, that has full tail call elimination [41]. The authors state that the JVM is not designed for functional programming idioms such as recursion for iteration, and first class functions, and state that functional languages in the JVM often work around such issues by providing alternatives to functional programming style. Two typical approaches on how to represent functions are discarded. Representing functions as Java methods is one approach, but has limitations as JVM methods don't support currying. Representing functions as Java objects is another approach, which languages like Scala and Clojure use, which allow for more flexibility. Functions as objects (FAO) can be represented as a FAO interface, with an apply method that takes an argument. Neither option provided a good solution to the problem of general tail call elimination. The authors presented a different approach to representing first class functions called imperative function objects

(IFO). IFOs are represented as an abstract class with argument and result fields, and an apply function. are distinct from the previous approach for representing functions as objects in that setting an argument and invoking a function are two different parts. In tail calls, the IFO has its arguments set, but invocation is delayed - instead, an auxillary structure saves the IFO. The saved IFO is executed at the original call site, looping until the auxillary structure is null.

2.3.1 Continuation Passing Style and Trampolines

One strategy for implementing tail call elimination is for the compiler to convert a program into continuation-passing-style (CPS). The term continuation passing style was first used when describing iteration implemented with recursion in early Scheme [38]. It is a style of programming where functions are supplied an extra argument in the form of a continuation and, rather than returning, the function calls the continuation. CPS is often used as an intermediate representation by compilers, and compiling tail calls with CPS can provide tail call elimination [1].

Another method of implementing tail call elimination is to use a trampoline, occasionally referred to as a dispatch loop. Assume a function f with a tail call to a function g - f is called from the trampoline and returns g - the trampoline then calls g and so forth as long as a function reference is returned by the function it calls. Because f returns, the stack frame for f is popped before calling g thus preventing unlimited stack growth. This technique isn't necessarily separate from CPS, as it may be used to implement CPS.

In looking at the problem of compiling Scheme to C, Baker presents a solution in which the Scheme program is converted into CPS [2]. Trampolines were considered, but eliminated due to performance concerns and that arguments need to be passed into global variables. The author proposed to compile Scheme by converting a program to CPS. Each lambda in CPS would be compiled as separate C functions. Continuations are passed as extra arguments - the code portion of a closure and the environment part of a closure are passed as separate arguments.

Clinger presented a formal definition of proper tail recursion for Scheme which, once again, is defined as including all tail calls, not just self tail calls [10]. Clinger presented this in response to definitions of tail recursion that were either too implementation specific, or too informal. Clinger defines tail expressions in core Scheme (after macro expansion) as being the following: the body of a lambda expression and the branches of a conditional expression, if the conditional expression itself is a tail expression. Tail calls are therefore any tail expression that is also a procedure call. Self tail calls are tail calls which calls itself recursively. Clinger defines the essence of proper tail recursion to be that a procedure can essentially return by performing a tail call. The responsibility for the return is passed on to the procedure called during the tail call. Proper tail recursion is important in Scheme as continuation passing style is a common idiom in Scheme and requires proper tail recursion to be feasible, as proper tail recursion allows for CPS in a bounded amount of storage. Because of the importance of proper tail recursion and its space efficiency, Clinger defines proper tail recursion as a model of space consumption, which serves as an asymptotic upper bound on the space consumption of the actual implementations. Clinger's work serves as the definition of proper tail recursion in the specification for Scheme [43].

Schinz and Odersky implemented general tail call elimination for a functional language called Funnel, which targeted the Java Virtual Machine [35]. Like many functional languages, Funnel does not provide built-in loop constructs - instead it relies on recursive calls to provide iteration. Furthermore, Funnel made use of visitors to provide algrebraic data types, so recursive loops do not consist only of self tail calls. This required implementation of general tail call elimination. The authors looked at several methods, including compiling the entire program into one function, trampolines and conversion into continuation passing style. Compiling the program to one function was deemed unrealistic because the JVM has a size limitation for methods of 64 kb. Trampolines were not attempted, as previous work had shown significant slowdown for C. Converting the program to CPS, and returning to the bottom of the call stack when nearing stack overflow was of interest to the authors as it guarantees execution in a bounded stack space, but tests showed that implementing a function in CPS caused significant slowdown. The authors decided that one could add a counter for tail calls, tail call limit (TCL), which, when reaching a maximum value, would trigger a return to a trampoline at the bottom of the tail call chain. Returning to trampoline can be executed by either a chain of return instructions, or by throwing an exception. Tuning the TCL is a matter of choosing stack usage over speed - a high TCL means higher stack usage, but higher speed (as less returns are needed), whereas a low TCL means the opposite.

Tarditi, Lee and Acharya presented a compiler for Standard ML to C, which needed to maintain proper tail recursion [40]. The authors cited previous Scheme to C compilers which failed to maintain proper tail recursive behavior, and sought to modify an existing compiler, Standard ML of New Jersey, to produce C code that handled the language in a faithful way. The existing compiler uses CPS as an intermediate representation before code generation, and the decision was made to keep the same intermediate representation and change the code generator to produce C. A trampoline (referred to by the authors as a dispatch loop), is used to prevent stack growth for the C implementation of calls.

Chapter 3

Experimental Design

This chapter will discuss in depth the design and implementation of tail call elimination in two different variations in the OpenSmalltalk VM, and will discuss the implementation of tests for these variations. Relevant details about the implementation of OpenSmalltalk-VM will also be discussed.

3.1 OpenSmalltalk Architecture

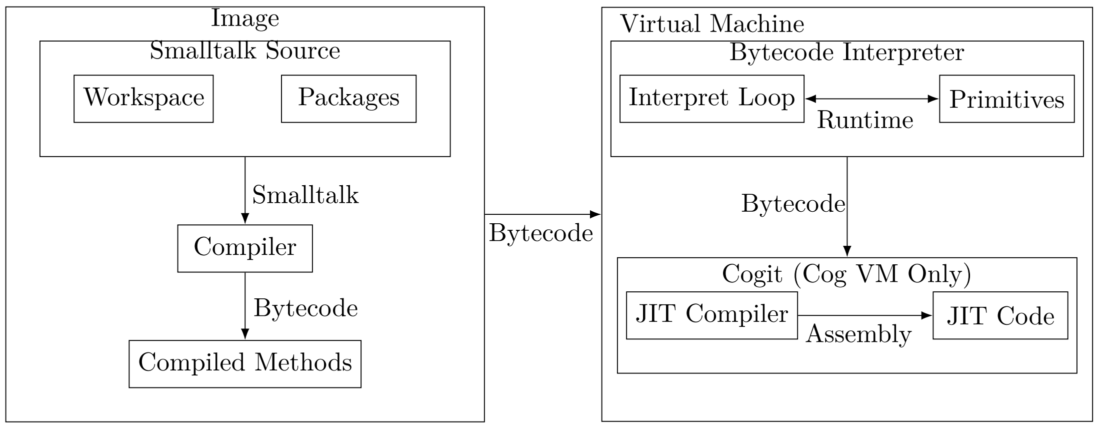


Figure 3.1: OpenSmalltalk Architecture

The overall architecture of the OpenSmalltalk VM is detailed in Figure 3.1. Broadly speaking, a Smalltalk system consists of a Smalltalk image and a virtual machine. A Smalltalk image contains all of the Smalltalk source code, the compiled bytecode, and the state of any instantiated objects. The image is also where the code for the Smalltalk to bytecode compiler exists. The virtual machine is the component of the system responsible for the interpretation and execution of the compiled bytecode. In addition to executing bytecode, it also executes runtime procedures called primitives. For the Stack Interpreter, the bytecode interpreter is the only relevant component of the virtual machine. For the Cog VM, there is also a JIT compiler component of the virtual machine, which compiles bytecode to assembly. Note that the virtual machine never compiles or translates actual Smalltalk source code, and that the Smalltalk image never executes, compiles or stores any JIT compiled assembly code.

3.2 Stack Interpreter

The first section will discuss the implementation of several tail call elimination strategies within the Stack Interpreter. The Stack Interpreter was chosen for this test for three reasons. One, as mentioned earlier, the Stack Interpreter uses a traditional call stack and only creates method context objects as needed, which makes it possible to implement stack based strategies for tail call elimination. Two, the Stack Interpreter is a pure interpreter - the subclass of the Stack Interpreter used for the JIT, the CoInterpreter, would not be capable of testing a purely interpreter-based implementation of tail call elimination. Third, from an implementation perspective, the Stack Interpreter is much less complicated than the Cog VM and provided an ideal environment for experimenting and learning about tail call elimination. In this section, the overall flow of execution in the Stack Interpreter will be discussed. Next, the exact execution of a method send and a return will be discussed in terms of the call stack. Finally, an implementation of tail call elimination is discussed, as well as some approaches that were discarded.

3.2.1 Execution Flow in the Stack Interpreter

First, an overview of the flow of execution in the Stack Interpreter is needed. Figure 3.2 shows a simplified flowchart of the Stack Interpreter, with details relevant to message sending. The interpreter runs as a loop, executing a bytecode and then fetching the next bytecode. Table 3.1 shows a subset of the variables that the Stack Interpreter uses to keep track of its state - this subset is made up of the variables relevant to this discussion.

Note that both Figure 3.2 and Table 3.1 make reference to internal and external facets of the Stack Interpreter. In the virtual machine, the interpret loop and bytecode execution are implemented as an interpret procedure containing a loop with a switch statement, with bytecodes implemented as cases in this switch statement. The internal variables are implemented as local variables to the interpret procedure as an optimization. When a function is called from the switch statement (ie. an operation is needed outside

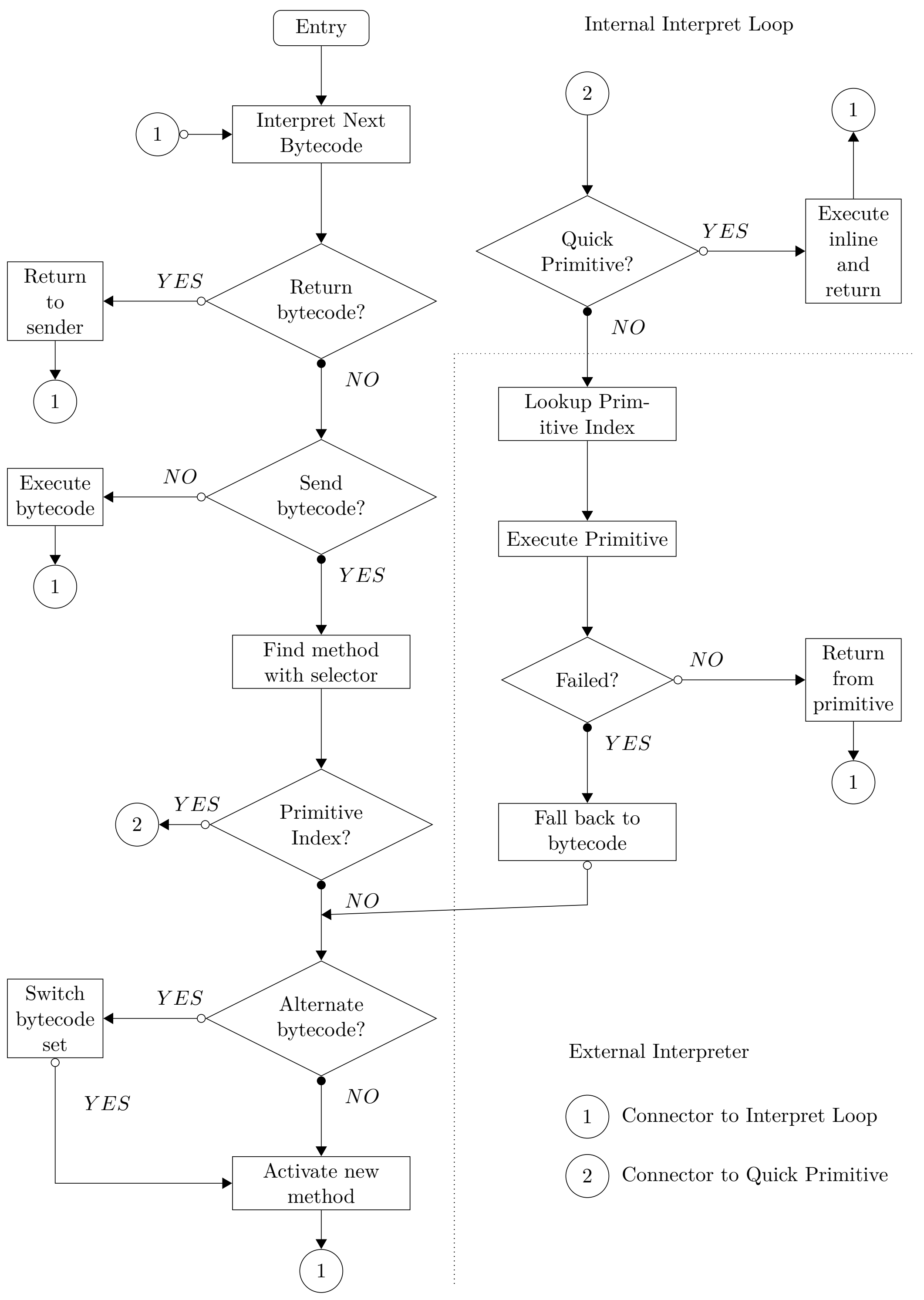


Figure 3.2: Method Execution in StackInterpreter

Internal Variable	External Var.	Purpose
localFP	framePointer	Stores current frame pointer
localSP	stackPointer	Stores current stack pointer
localIP	instructionPointer	Stores current instruction pointer
currentBytecode	n/a	Stores current bytecode
method	n/a	Pointer to current method
bytecodeSetSelector	n/a	Offset for multiple bytecode sets

Table 3.1: Execution variables in StackInterpreter

of the bytecode case statements), the values of the internal variables are copied to the corresponding external variables, which are implemented in C as globals. Any variable without a corresponding external variable is either not needed outside of the interpret procedure, or is calculated another way. Table 3.2 details what bytecodes and bytecode ranges will be relevant to this discussion. Note that literal refers to a reference to a string or symbol encoded in a method - in this case, the literals referenced will be method selectors.

176 to 191	Special Arithmetic selectors
192 t0 207	Special Keyword selectors
208 to 223	Sends a 0 argument selector to the first 16 literals
224 to 239	Sends a 1 argument selector to the first 16 literals
240 to 255	Sends a 2 argument selector to the first 16 literals
131	Sends up to 7 argument selectors to the first 32 literals
134	Sends up to 3 argument selectors to the first 63 literals
132	Can send up to 31 argument selectors to 256 literals
120	Returns a reference to self.
121	Returns the boolean True.
122	Returns the boolean False.
123	Returns the Nil object.
124	Returns the top of the stack from a method.
125	Returns the top of the stack from a block

Table 3.2: Send and Return Bytecodes

Referring to Figure 3.2, execution enters the interpret loop, typically after the VM starts up and the image is loaded, as indicated by the node labelled Entry. In the interpret loop, the next bytecode is interpreted - as mentioned before, this is via a switch statement matching on the currentBytecode variable. For the purpose of this discussion, all bytecodes except send bytecodes and return bytecodes will be ignored - refer to Table 3.2 for a look at the relevant bytecodes. If the executed bytecode is a return bytecode, a return sequence will be performed and execution will continue at the next bytecode of the sender. Details of the return sequence will be given in the next section. If the executed bytecode is neither a return or a send, the bytecode will be executed and execution will continue at the next bytecode.

If the bytecode is a send bytecode, the interpreter first looks up the method using the literal provided as the selector and the class of the receiver of the method. A compiled method in the Stack Interpreter has a special method header - this has encoded information such as the number of arguments, number of temporaries, whether an alternate bytecode set was used and the number of literals. See Figure 3.3 for the exact specifications.

b	u	u	u	a	a	a	a	t	t	t	t	t	t	f	p	l	l	l	l	l	l	l	l	l	l	l	l	l	l	l	l
31			28				24						18	17	16	15														0	

Bit(s)	Meaning
31	Alternate bytecode flag
28-30	Unused
24-27	Number of Arguments
18-23	Number of Temporaries
17	Frame Size
16	Primitive Flag
0-15	Number of Literals

Figure 3.3: Spur 32 bit Method Header

In this case, the relevant field in the method header is bit 16 - if bit 16 is set to 1, this indicates that the method has a primitive implementation. If the method has a primitive implementation, the index of the primitive method is extracted from the argument of the first bytecode[1]. Below is an example of a method with a primitive implementation.

```
SmallFloat64>>+ aNumber
   <primitive: 541>
   ↑ aNumber adaptToFloat: self andSend: #+
```

The following is the bytecode for the above method. Note that the first bytecode has the primitive index as an argument.

[1]Previous implementations embedded the primitive index within the header, which limited both the number of primitives available in the system, and the size of the methods

```
<primitive: 541>
21 <8B 1D 02> callPrimitive: 541
24 <10> pushTemp: 0
25 <70> self
26 <21> pushConstant: #+
27 <F0> send: adaptToFloat:andSend:
28 <7C> returnTop
```

If the primitive index falls into a certain range, 1 to 519 in the StackInterpreter, the primitive is considered a quick primitive and is executed by the virtual machine before returning to the interpret loop to execute the next bytecode of the calling method. Quick primitives are operations that can be quickly completed without interpreting bytecode or calling a function, such as simple arithmetic operations on lower precision numbers, or simple boolean operations. If the primitive is not a quick primitive, the internal variable values are copied to the global variables (the variables referenced in Table 3.1), and the function corresponding to the primitive index is called. The primitive function is executed and, if successful, execution is returned to the interpret loop and the state of the loop is updated by copying the global variables back to the internal variables, and execution continues at the next bytecode of the calling method. Primitives can fail execution - for example, certain arithmetic operations may fail with high precision numbers. If the primitive fails, execution returns back to the interpret loop and the variables are copied back. However, instead of returning to the calling method, note that in the example listing above there was still a bytecode implementation of the method with the primitive index. Execution falls back to the bytecode implementation, skipping ahead to the first non-primitive call bytecode.

In most cases, the method will not have a primitive index, and execution will continue in the interpret loop (this is also the case if the primitive implementation fails). The method will now be activated within the interpret loop. However, first, there is a check to see if the method uses an alternate bytecode set. OpenSmalltalk has supported multiple bytecode sets - first, it supported the bytecode set for Newspeak, and then it supported the experimental Sista bytecode set. A flag in the method header is checked (see Figure 3.3) to see if an alternate bytecode set is used - if so, each bytecode is incremented by another byte (in essence, the interpret loop switch statement supports 512 bytecode operations - 2 sets of 256 operation bytecode sets). Finally, the called method is activated and the next bytecode is interpreted.

One observation can be made from following the execution flow as it pertains to tail call elimination. Implementation of tail call elimination need not be considered for primitives - quick and otherwise - as execution of quick primitives either occurs immediately or outside of the interpreter's call stack.

3.2.2 Sending on the Stack Level

Having followed the general flow of execution through the Stack Interpreter, two specific things need to be examined in greater detail from the perspective of the call stack - sending a message and returning. Recall the hypothetical call stack from the example of a tail call given in Chapter 1 - in this section, a concrete example will be given using methods implemented on an object named `TestObject`.

```
TestObject>>methodA: argA
   self methodB: argA
```

The first method is `methodA:`, which simply takes an argument, and sends the message `methodB:` with that argument to itself. The result is not explicitly returned, so, in Smalltalk syntax, a reference to self is returned.

```
17 <70> self
18 <10> pushTemp: 0
19 <E0> send: methodB:
20 <87> pop
21 <78> returnSelf
```

In the compiled bytecode, it's clear that this method is not a tail call - this method is just for producing side effects. The return result of the send to `methodB:` is discarded before returning Self.

```
TestObject>>methodB: argB
   ↑ self methodC: argB
```

The next method is `methodB:`, which takes one argument, and sends a message to `methodC:` with its own argument, and then returns the result.

```
17 <70> self
18 <10> pushTemp: 0
19 <E0> send: methodC:
20 <7C> returnTop
```

Looking at the bytecode, it is obvious that `methodB:` has a tail call, as it performs a send of `methodC;` and then returns the top of the stack. Next, here is the receiving method, `methodC:`.

```
methodC: argC
   | temp |
   temp := argC + 1.
   ↑ temp
```

In ths method, a temporary variable is created and assigned the value of the argument + 1. Then, the temporary variable is returned. Here is the compiled bytecode for `methodC:`.

```
13 <10> pushTemp: 0
14 <76> pushConstant: 1
15 <B0> send: +
16 <69> popIntoTemp: 1
17 <11> pushTemp: 1
18 <7C> returnTop
```

Looking at the bytecode, it's clear that `methodC:` does not perform a tail call. The result of the send to `+` is used after the send for other operations before returning the top of the stack. Moving on, the call stack and the execution variables will now be shown. Since the point of interest here ultimately is the tail call in `methodB:`, the activation of `methodA:` will be skipped and the details of `methodA:`'s frame will be obscured. Refer to Figure 3.4 for the result of jumping ahead to the invocation of `methodB:`.

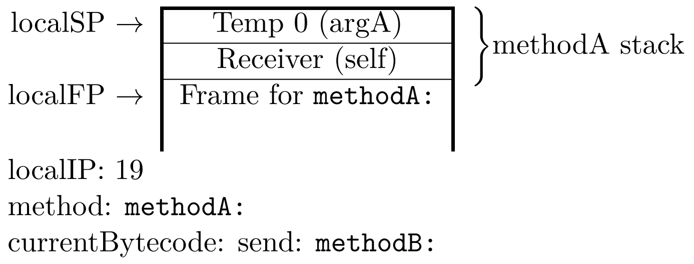


Figure 3.4: Sending `methodB:` from `methodA:`

Right before the send to `methodB:` is executed, the receiver (self) and argument (arg) are pushed onto the call stack. localFP, the frame pointer, refers back to the start of `methodA:`'s stack frame. localSP, the stack pointer, is pointing to the top of the stack. Assuming that `methodB:` does not have a primitive index, Figure 3.5 shows the result of activating `methodB:`.

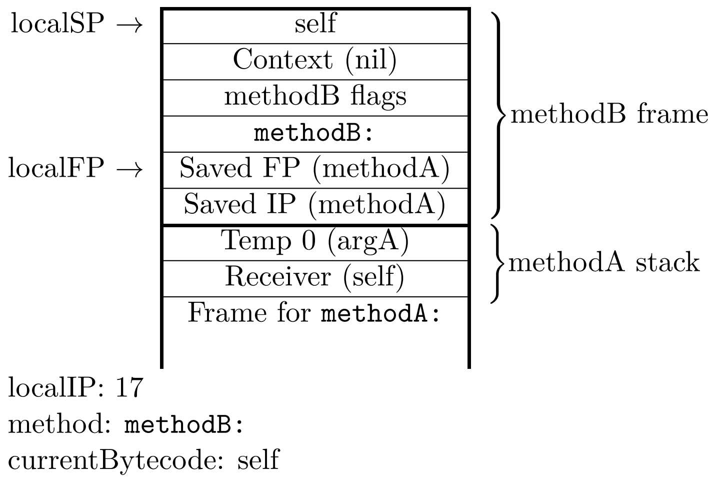


Figure 3.5: Activating `methodB:` after send

When `methodB:` is activated, several values are pushed onto the stack to serve as the stack frame for `methodB:` and to facilitate the eventual return from `methodB:` to `methodA:`. First, the instruction pointer, localIP, is pushed onto the stack- this saves the instruction pointer's location in `methodA:` for eventual return. Next, the previous frame pointer, localFP, is pushed onto the stack, and then the current stack pointer, localSP, is copied to localFP. localFP now serves as the frame pointer for `methodB:`. The frame pointer serves as a known point in the stack, where, by using offsets, execution can reference the receiver, arguments, temporaries, etc. Next, a reference to the current method, `methodB:`, is pushed onto the stack. While a reference to the method also exists in a local variable to the interpret loop, a reference on the stack is needed in case the method needs to be accessed from outside the interpret loop. Next, a set of flags is pushed - this contains values that need to be referenced such as the number of arguments. Next, a slot is pushed for a reference to a heap allocated context object - recall that the Stack Interpreter will allocate a context object as needed, so the slot initially is a reference to the nil object. Finally, a reference to self is pushed. If `methodB:` had temporary variables, they would be pushed now, but in this case they are not needed. From this point, the execution of methodB is simple to follow, as a reference to self is pushed onto the stack, followed by the argument and a call to `methodC:`, and the activation of `methodC:` will be similar to the activation of `methodB:`, pushing the same values needed for the stack frame, and updating the execution variables. Figure 3.6 shows the stack after activation of `methodC:`.

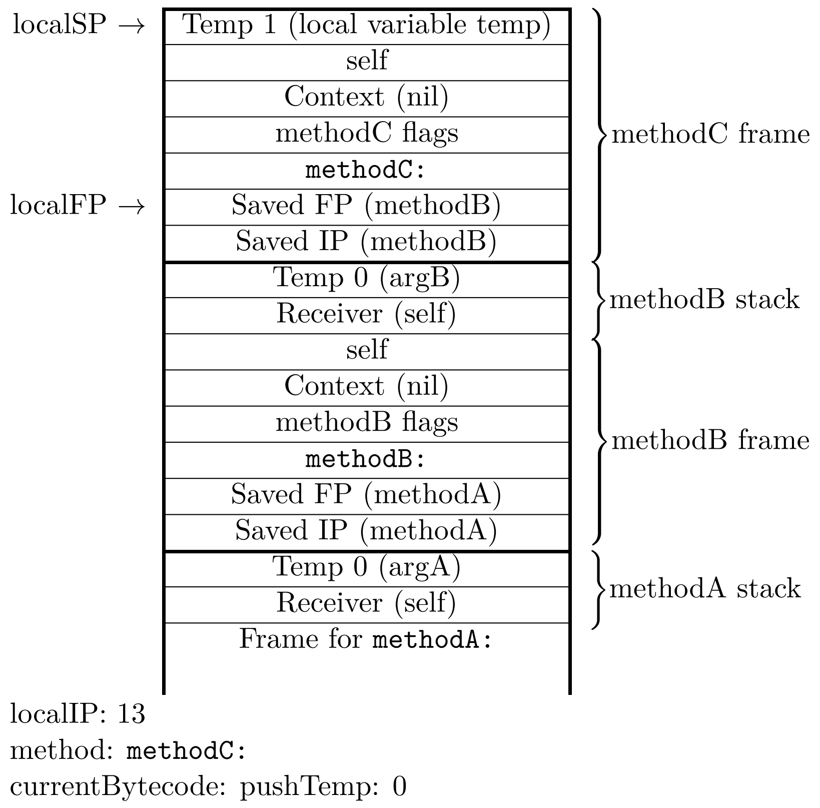


Figure 3.6: After activation of `methodC:`

Figure 3.7 shows the stack after execution of `methodC:` right before the return instruction (methodB's stack frame has been removed for clarity). Execution of `methodC:` has proceeded up to the point where the temporary variable has been pushed onto the stack, awaiting return.

3.2.3 Returning on the Stack Level

Continuing on from the previous section, the returnTop bytecode in `methodC:` will now be executed. The return sequence for returnTop is simple - the stack frame for `methodC:` should be removed, the execution variables should be set back to the sender, `methodB:`, the arguments for `methodC:` should be consumed, and the return value should be pushed the top of `methodC:`'s stack. This is accomplished in the following sequence - first, the top of the stack is copied into a temporary variable. Next, the frame pointer, localFP, is used to access the saved instruction pointer which is set back to localIP. Next, the

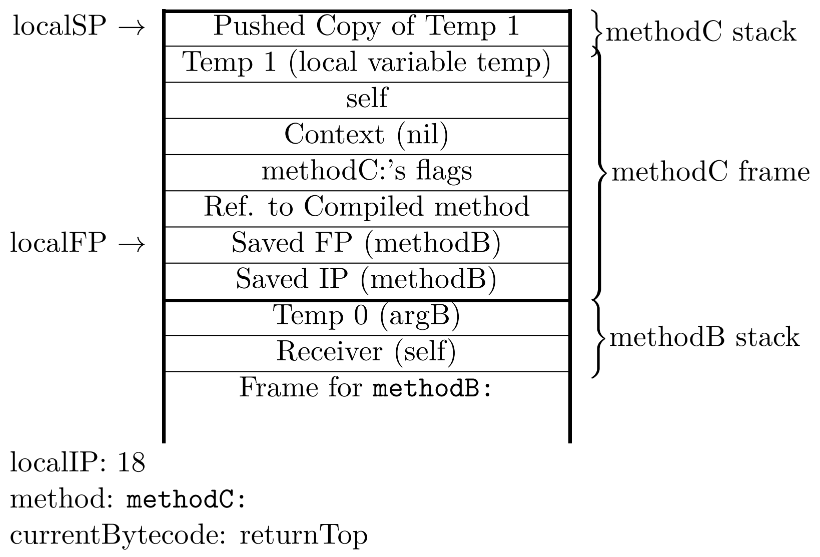


Figure 3.7: Prior to return in `methodC:`

stack pointer, localSP, is set to the value of localFP with an offset applied based on the number of arguments - this moves the stack pointer back to where the initial receiver was in order to consume the arguments. The method variable is set back to the saved method pointer, the next bytecode is fetched, and finally the saved return value is pushed onto the top of the stack. Figure 3.8 shows the state of the stack after returning to methodB from methodC.

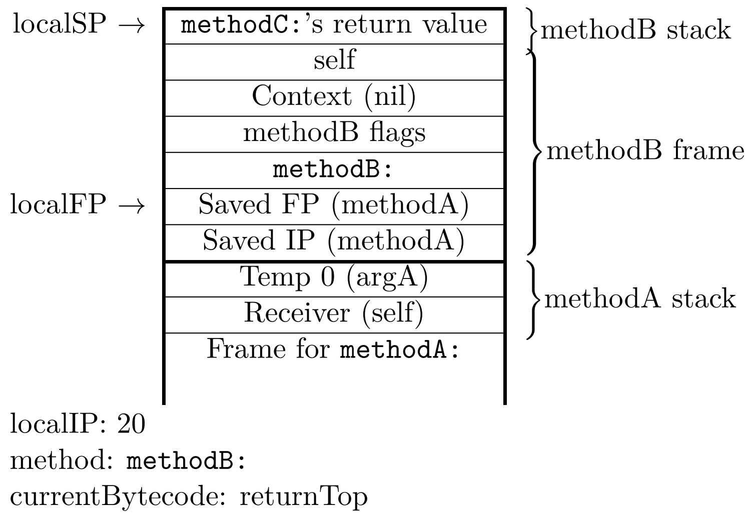


Figure 3.8: Stack of `methodB:` after returning from `methodC:`

Note that, since the send to `methodC:` inside `methodB:` was a tail call, `methodB:`

now immediately returns the top of the stack back to `methodA:`. Figure 3.9 shows the stack after performing the same returnTop sequence on `methodB:`. The return value of `methodC:` is now on the top of the stack of `methodA:`.

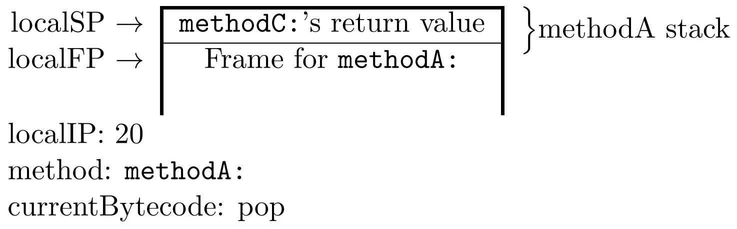


Figure 3.9: Stack of `methodA:` after returning from `methodB:`

3.2.4 Applying Tail Call Elimination

Having explored the concept of tail call elimination in the first two chapters, the opportunity to apply tail call elimination to the previous sequence of sends and returns should be clear. From previous work, emulating a jump without pushing a return instruction seems like a possible solution [37]. However, with Smalltalk bytecode, there is no direct jump to an address available. Instead, a modification of the send sequence such that the previous stack frame is removed or reused when making a tail call could be used, similar to the approach discussed by Hanson when calling a reduction [16]. Assuming that a tail call is somehow identified to the Stack Interpreter, the call and return sequence for the tail call in `methodB:` could be redone with this approach. First, recall Figure 3.6, which showed activation of `methodC:` after being sent from `methodB:`. Figure 3.10 shows the activation of `methodC:` after using a tail call elimination strategy.

Note that `methodC:` has essentially replaced `methodB:` in the stack. How did this happen? Putting aside the problem of how the interpreter should identify a tail call for now, the tail call implementation of a message send can be fairly simple. Recall that, when executing a call, the receiver and any arguments are on the top of the stack, and the frame pointer (localFP) and an offset is used to refer to them. Also, when returning, the arguments and receiver are consumed and replaced with the top of the stack. The stack frame needs to be replaced with a stack frame that contains references to the called method. In the example above, `methodB:`'s stack frame needs to be replaced with `methodC:`'s, and the receiver and arguments that precede `methodC:`'s stack frame need to be `methodC:`'s receiver and `methodC:`'s arguments instead of `methodB:`. The amount of operations required for this depend on how closely the called method matches the callee. What follows is the chosen implementation for several different cases of tail call

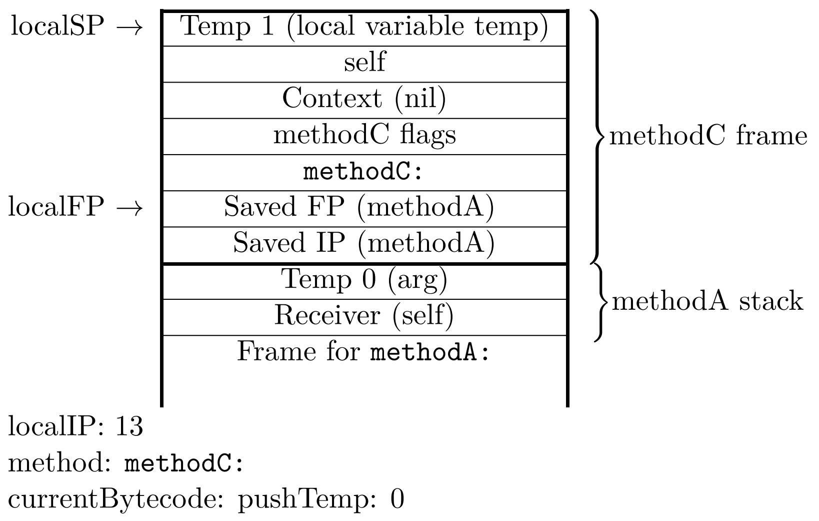


Figure 3.10: Tail call activation of `methodC:`

in the Stack Interpreter.

Tail Recursive Calls

For tail recursive calls, or self tail calls, the caller and callee methods are the same. This is the simplest case of tail call elimination. Since the method is the same, we know that the argument count is the same, and the saved method and method flags are the same. Tail recursive calls can be implemented with the least number of changes, as the number of arguments are the same and the method being called is the same. Therefore, the receiver and arguments that were used to call the method with the tail call can be replaced with the receiver and arguments of the tail call, and the saved method and method flags in the stack frame can stay the same. The frame pointer (localFP) can stay the same in this case as well. The saved instruction pointer stays the same, the instruction pointer (localIP) is set to the first instruction of the method, and the stack pointer (localSP) needs to be set to the frame pointer. Slots for the context, temporaries and the self reference need to be repushed.

Tail Calls with Same Arity

Calls with the same arity but different methods are slightly more complicated. For calls with the same number of arguments, in addition to replacing the arguments and receiver, the method for the frame must also be changed. The saved method and method flags must be updated with the new method, and the method execution variable must be updated. Like self tail calls, the frame pointer (localFP) can stay the same. The rest of the process is the same as self tail calls.

Tail Calls with Different Arity

Calls with a different arity are the most complicated cases. For calls with a different arity, the position of the stack frame is offset differently by the number of arguments. Therefore, nothing about the stack can be modified in place - essentially, the stack frame needs to be rebuilt with the new method, but in place of the old method instead of afterwards. Values such as the saved frame pointer, saved instruction pointer, tail call arguments and receiver need to be copied to temporary variables by the Stack Interpreter runtime to avoid being lost. The stack pointer (localSP) is set prior to the receiver of the method with the tail call in it, the receiver and arguments are pushed and then the stack frame for the tail call is rebuilt, pushing the saved frame pointer and saved instruction pointer from local variables in the process. In all of the above cases, the end result is such that the method with the tail call is no longer executing, and execution will not return to it. The returnTop bytecode seen in examples like `methodB:` will not be executed - instead, when `methodC:` returns, its return value will be returned to `methodA:`.

3.2.5 Identifying a Tail Call

Having explored the implementation of tail call elimination in the Stack Interpreter, the next question is how does the interpreter identify a tail call in order to execute a tail call instead of a regular message send? One possible implementation would simply check before execution of each send bytecode - it could prefetch the next bytecode and check to see if it is a returnTop bytecode - if so, execution of the bytecode could switch to a tail call version of that bytecode. This was ultimately the approach that was used in the implementation, however, this approach does have the overhead of implementing a new check on every send instruction. Some other approaches were explored to take advantage of existing checks in order to minimize any overhead, or to shift the overhead of doing a check for a tail call from runtime to compile time. If a tail call was identified at compile time, any extra overhead would be only on method compilation. Identifying a tail call at compile time is trivial, but coming up with a strategy for how to flag this for the run time is more difficult. Ideally, there would be enough free space in the bytecode set to implement new tail call versions of all send bytecodes seen in Table 3.2. However, the current bytecode set has very limited free space.[2] Instead, two different

[2]The Sista bytecode set has more available space for experimentation with new bytecodes, but was not the default bytecode set of either Squeak or Pharo at the time of development

approaches were explored for ways to switch execution of calls to tail call implementations using existing checks in the interpreter without adding any overhead to non-tail calls. These approaches would use checks that have already been discussed in the context of execution flow. Ultimately both of these approaches were discarded early in development tests, as little benefit was seen from the added complexity of their implementation, and are included for completeness.

Primitive Implementation

The first method tested is to add a primitive to the Stack Interpreter that signals a switch of execution to an interpret loop with tail calls instead of regular calls. As explored earlier in this chapter, methods with a primitive flag bit in their method headers will cause execution to switch out of the interpret loop. The primitive index in the method is extracted and the corresponding primitive function is executed - if the primitive fails, execution falls back to the interpret loop, and if the primitive succeeds, execution continues after the point that the primitive method was invoked. In this approach, when the compiler encounters a tail call, it sets a specific primitive on the method.

Alternative Bytecode Set

The next method tested is to add a new bytecode set, where all sends are implemented as tail calls [3]. As mentioned, method headers already have a flag to indicate whether an alternate bytecode set is being used. And the flag is already being checked on method activation to see what the bytecode offset should be[4]. If this flag is repurposed as a check for a tail call, it adds no extra overhead. In this implementation, the compiler determines that a method ends in a tail call, the alternate bytecode set flag will be set on the method header. Then, when the method is activated and the flag is checked, the bytecode set is flipped to the bytecode set where all calls are implemented as tail calls.

3.3 Cog JIT Compiler

The next section will discuss the implementation of tail call elimination in the OpenSmalltalk JIT compiler. The JIT compiler is known as Cog, or Cogit, and the OpenSmalltalk VM with the Cog JIT compiler is occasionally referred to as the Cog VM. The Cog VM is made up of two main classes - the CoInterpreter and the Cogit. The Cogit is

[3]Currently, the OpenSmalltalk-VM only supports two bytecode sets - this implementation prevents the use of both the default bytecode set and the Sista bytecodes

[4]As the flag is simply true or false, the offset is either 0 or 256

the JIT compiler itself, which is responsible for JIT compiling new methods, and making modifications to existing JIT-compiled methods. In Smalltalk, it is structured such that the actual platform-specific back end is another object set in an instance variable in the Cogit. Slang compiles these objects together into a platform-specific Cogit C file, which is then built as an executable VM. The CoInterpreter is an extension of the StackInterpreter with the added hooks installed to be able to JIT compile methods as appropriate, and transfer control from interpreted methods to JITted methods, and back again. The choice was made to add tail call elimination to the Cog VM because it is an important piece of the OpenSmalltalk VM moving forward. Also, as JIT compiled code executes on a call stack, it should be possible to use some of the same strategies used in implementing tail call elimination on the interpreter. First, however, the flow of execution in the Cog VM will be discussed. JIT compiled methods also use inline caching at send sites, which will be looked at in detail. Finally, a tail call elimination implementation will be presented, and how to manage it with inline caching and garbage collection will be briefly discussed.

3.3.1 Execution Flow in the Cog VM

Figure 3.11 shows the flow of execution in the Cog VM. Broadly speaking, execution in the Cog VM can be divided into the evaluation of bytecode, execution of the JIT-compiled assembly; and the runtime routines that support these. Since bytecode executing on the interpreter and the JIT-compiled assembly code share a call stack, one can think of them as executing on the Smalltalk stack (either by interpretation of bytecode or directly executing assembly) The virtual machine runtime executes on a separate call stack, referred to as the C stack. In the diagram, execution is divided into the same three areas: interpreter, runtime routines and machine code. Specifically, the runtime routines in the diagram represent trampolines, which switch execution from machine code to the interpreter, and the aptly named enilopmarts[5], which take execution from the interpreter to the machine code. As mentioned, interpreted bytecode and JIT compiled code share a call stack. Since the interpreter tracks execution of interpreted code using local variables (like the aforementioned stack pointer and instruction pointer), and assembly uses registers for these values, it's the job of the trampolines and enilopmarts to copy the values back and forth between the two execution environments so that they can share a common stack. As a general rule, when an enilopmart jumps from the interpreter to executing JIT compiled machine code, it saves the existing values of the registers (saving

[5]This is simply "trampoline" reversed.

the state of execution on the C stack), and copies the values of the interpreter's local variables to the registers. When a trampoline jumps back from JIT compiled machine code to the interpreter it does the reverse - saving the values of the registers into the interpreter's local variables (saving the state of execution on the Smalltalk stack), and copying the saved state of the C stack back into registers.

A note about the interpreter (the CoIntepreter class) - recall that, in the Stack Interpreter, much of the discussion of the execution flow was spent on the distinction between bytecodes executed within the interpret loop versus code executed outside the loop as primitives. All of this complexity still exists in the CoInterpreter, but has been removed from the diagram for simplicity. Also, the stack frame format is very similar between the CoInterpreter and the StackInterpreter, and differences between interpreted methods and JIT compiled methods are quite minor.

Following the flow chart in Figure 3.11, execution enters the interpreter at startup (the entry node) and, like the Stack Interpreter, execution continues in a loop interpreting bytecode. If the bytecode to execute is a return bytecode, a return sequence is called and, if the method being returned to is another interpreted method, execution remains in the interpreter. If the method being returned to is a JIT compiled method, execution enters a runtime enilopmart method to switch execution to machine code, in a process which will be elaborated on shortly. Continuing on, if the bytecode is neither a return nor a send, execution in the interpreter simply continues with the next bytecode. If the bytecode is a send, the combination of the selector and the class of the receiver are looked up in the interpreter's method cache[6]. The method cache is a lookup table of class and method selector combinations that have been called in the interpreter, and is not specific to a send site. If the method is not found in the cache, it is added to the cache and is activated for execution in the interpreter. If the method is found in the cache, a check is done as to whether it has been JIT compiled or not - if it has been JIT compiled, an enilopmart is called to transfer execution to machine code and the method is activated. If it hasn't been JIT compiled, a check is done to see if the method can be JIT compiled (methods over a certain size will never be JIT compiled due to space concerns). If the method can't be JIT compiled, it is activated for execution on the interpreter. If it can be JIT compiled, the method is compiled by the Cogit class to platform specific assembly, and then an enilopmart is called to transfer execution to machine code to activate the new method.

It's worth taking a look at what exactly enilopmarts and their corresponding tram-

[6]This was also the case for the Stack Interpreter, but was omitted as it was not relevant to the tail call implementation

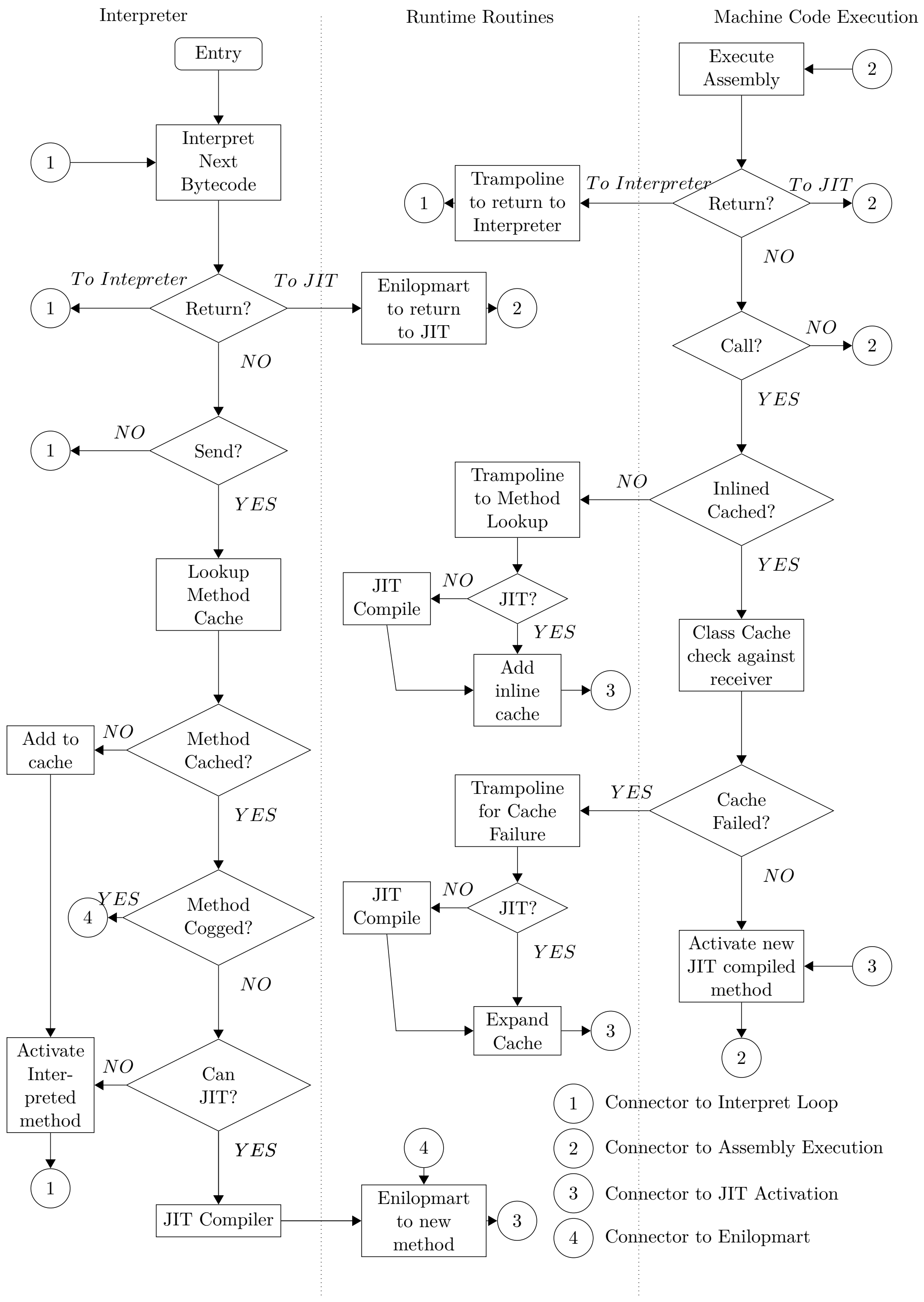


Figure 3.11: Method Execution in CogVM

polines in the other direction do to transfer execution between JIT compiled code and the interpreter. Recall that, as mentioned before, Smalltalk bytecode executed on the interpreter and JIT compiled code share a call stack. The actual virtual machine runtime uses a different call stack, which is referred to as the C stack. The CoInterpreter, like the Stack Interpreter, uses local variables for the stack pointer, frame pointer and instruction pointer to track the execution of Smalltalk bytecode on the Smalltalk stack, while tracking its own execution on the C stack in registers. However, when JIT compiled Smalltalk code is executing, those same registers are used to track execution on the Smalltalk stack. The runtime methods that constitute trampolines and enilopmarts are responsible for switching back and forth - a trampoline from JIT execution to the interpreter will copy the values of the registers used for the stack pointer, frame pointer and instruction pointer on the Smalltalk stack into the local variables used by the interpreter, and then restore the saved values from the C stack to reenter the runtime. An enilopmart will perform the opposite to transfer control from the interpreter to JIT code - it will save the values of the registers pointing to the C stack, and then copy the values of the local execution variables into the registers. Trampolines are often tailored to specific methods in the interpreter that need to be called from JIT code, for example, if a method fails in some way that needs to be handled by the interpreter - examples will be seen shortly.

Continuing on through Figure 3.11, assuming that execution has reached machine code execution through an enilopmart and a JIT compiled method has been activated, JIT compiled assembly code is now being executed in the node labeled 'Execute Assembly'. If the instruction is a return instruction, similar to how the interpreter handled returns, execution will either stay in machine code or return to the interpreter via a trampoline depending on what sort of method is being returned to. If the instruction is neither a return nor a call, execution of assembly continues on to the next instruction. If the instruction is a call instruction, there are a few possible scenarios - the details will be discussed shortly as it is very relevant to how tail call elimination will be implemented. However, from a high level perspective, a check for an inline cache in the JIT code is done at the call site. This is different than the method cache discussed earlier in the interpreter - this is a cache of a class and selector written directly into the call site in the assembly code. If there is no cache, a trampoline is called to look up the method, adding an inline cache if the method found is already JIT compiled, or is JIT compiled if not. If there is an inline cache, it is verified against the receiver, and either the cache matches and the new method is executed, or the cache is expanded and the new method is executed either in machine code or is JIT compiled and then executed.

3.3.2 JIT Compiling a Method

From the discussion above, a few design decisions can be made about the implementation of tail call elimination. The best way to illustrate these decisions will be with a concrete example of some method sends and a series of executions of that method in the Cog VM. The target platform for the JIT compiler will be IA-32 assembly code.

Method Definitions

What follows is the Smalltalk source and compiled bytecode for a set of methods that will be used in the concrete example. The first method, `sendModifyArg:toRcvr:`, is implemented on an object of class `SendingObject`.

```
SendingObject>>sendModifyArg:arg toRcvr:rcvr
    ↑ rcvr modifyArg:arg
```

This method takes an argument, arg, and an object instance, rcvr, and sends the `modifyArg:` message to rcvr with arg as the argument. It then returns the return value of `modifyArg:`. Here is the compiled bytecode of `sendModifyArg:toRcvr:`.

```
17 <11> pushTemp: 1
18 <10> pushTemp: 0
19 <E0> send: modifyArg:
20 <7C> returnTop
```

From the bytecode, it's obvious that the send of `modifyArg:` is a tail call, as the top of the stack is returned immediately afterwards. As a dynamically typed language, the reciever of `modifyArg:`, rcvr, does not have a specific type - any object that has implemented the method `modifyArg:` can successfully receive the message. The dynamic typing of Smalltalk will be seen with the next two methods. First, there is an implementation of `modifyArg:` on an object of class `ReceivingObjectA`. The method takes the argument arg, adds 2, copies the value into a temporary, and returns the temporary.

```
ReceivingObjectA>>modifyArg: arg
    | temp |
    temp := arg + 2.
    ↑ temp
```

Looking at the bytecode below, this implementation of `modifyArg:` does not have a tail call - after the send to +, further actions are taken before the top of the stack is returned.

```
13 <10> pushTemp: 0
14 <77> pushConstant: 2
15 <B0> send: +
16 <69> popIntoTemp: 1
17 <11> pushTemp: 1
18 <7C> returnTop
```

Continuing on, here is an implementation of `modifyArg:` on an object of class `ReceivingObjectB`. This method is identical to the previous implementation of `modifyArg:`, except that the argument's value is incremented by 1.

```
ReceivingObjectB>>modifyArg: arg
    | temp |
    temp := arg + 1.
    ↑ temp
```

The bytecode for this implementation of `modifyArg:` is as follows - again, it's nearly identical to the previous implementation.

```
13 <10> pushTemp: 0
14 <76> pushConstant: 1
15 <B0> send: +
16 <69> popIntoTemp: 1
17 <11> pushTemp: 1
18 <7C> returnTop
```

Finally, here is a snippet of code that calls `sendModifyArg:toRcvr:` numerous times. This code could be in a method, or executed in a workspace.

```
| sender receiverA receiverB |

sender := SendingObject new.
receiverA := ReceivingObjectA new.
receiverB := ReceivingObjectB new.

sender sendModifyArg:1 toRcvr: receiverA. "Send 1"
sender sendModifyArg:1 toRcvr: receiverA. "Send 2"
sender sendModifyArg:1 toRcvr: receiverA. "Send 3"
sender sendModifyArg:1 toRcvr: receiverB. "Send 4"
```

In this code snippet, instances of the classes `SendingObject`, `ReceivingObjectA`, and `ReceivingObjectB` are created first. Next, there are a series of message sends to `SendingObject` - the first four are with the rcvr argument `receiverA` and the next two are with the rcvr argument `receiverB`. Comments have been added to the sends for the sake of this discussion. The bytecode for this code snippet is irrelevant, and will not be shown. Why the repeated sends with the same argument? Remember that, in `sendModifyArg:toRcvr:`, there is a tail call to `modifyArg:` with rcvr as the receiver - the behaviour that will be shown in this concrete example is inline caching, as it is relevant to the implementation of tail call elimination.

Execution of Send 1

Referring again to Figure 3.11 and the previous description of execution flow, execution enters the interpreter which evaluates each bytecode - for the purpose of this discussion, execution will skip ahead to Send 1. Also, the assumption will be made that this is the first time that this snippet of code has been run and that these methods have been called since the image was started up[7]. Since the method `sendModifyArg:toRcvr:` has never been called, the method and class combination have not been added to the method lookup cache in the interpreter - they will be added to the cache and the method `sendModifyArg:toRcvr:` will be activated in the interpreter. Now, following the activation of `sendModifyArg:toRcvr:`, execution continues through that method until the send of `modifyArg:` to the receiver, which is an instance of `ReceivingObjectA`. The class and message combination are not in the method lookup cache, so they are added, and `ReceivingObjectA>>modifyArg:` are activated and executed in the interpreter. Figure 3.12 shows a simplified view of the stack at this point preceding Send 1.

[7]While a Smalltalk virtual machine preserves instances of objects and execution state in the image on virtual machine shutdown and startup, JIT compiled methods are not saved

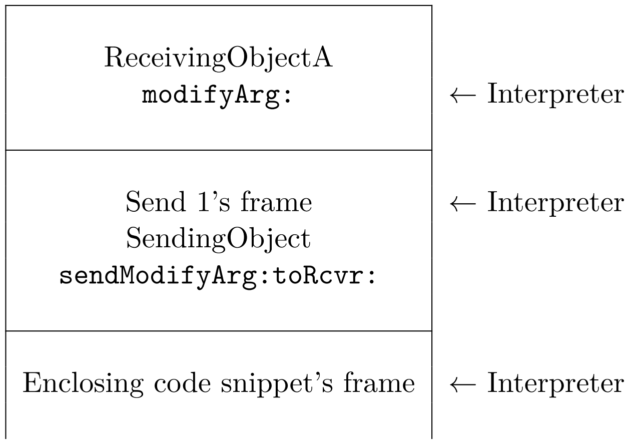


Figure 3.12: Stack before returning after Send 1

Continuing on with execution, `modifyArg:` executes a return of the top of the stack, which returns to `sendModifyArg:toRcvr:`. Since the call to `modifyArg:` was a tail call, the top of the stack is immediately returned to the enclosing code snippet, where the next bytecode to be executed is Send 2.

Execution of Send 2 - JIT Compilation

Every JIT compiled method is compiled with a similar set of entry instructions. A method is called at the address indicated above by the entry: label with a cached class tag in the **ecx** register and a reference to the receiver object in the **edx** register. See Table 3.3 for a table of how the IA-32 registers are used by the JIT compiled code.

Register	Purpose
eax	Temporary storage
ecx	Selector or Class Tag
edx	Receiver or Result
ebx	Number of Arguments
esi	First Argument
edi	Second Argument
esp	Stack Pointer
ebp	Frame Pointer
eip	Instruction Pointer

Table 3.3: IA-32 Registers and Usage

Continuing on, when the intepreter executes Send 2, the combination of the class `SendingObject` and the method `sendModifyArg:toRcvr:` are in the method lookup cache. Following the flowchart, a check is done to see if `sendModifyArg:toRcvr:` for SendingObject has already been JIT compiled. It has not at this point been JIT compiled, so the method is checked to ensure it can be JIT compiled and, since the method is not overly large, it passes and execution continues to the JIT compiler. What follows is a breakdown of the method `sendModifyArg:toRcvr:` compiled to IA-32 assembly code. The code will be divided up in sections to offer explanation of what each section does, as understanding it will be required later for the implementation of tail call elimination.

```
ABB60
  objhdr: 8000000A000035
  nArgs: 2 type: 2
  blksiz: 80
  method: 29278C0
  mthhdr: 4100007
  selctr: 29278A0=#sendModifyArg:toRcvr:
  blkentry: 0
  stackCheckOffset: 5F/ABBBF
  cmRefersToYoung: no cmIsFullBlock: no
```

This is the header for a JIT compiled method as extracted from the Cog VM simulator - ABB60 is the starting address of the method, followed by some information such as the address of the bytecode compiled version of the method, the selector, etc.

```
000abb7c: xorl %edx, %edx : 31 D2
000abb7e: call .+0xfff54b9d (0x00000720=ceMethodAbort2Args) : E8 9D 4B F5 FF
000abb83: nop : 90
000abb84: andl $0x00000001, %eax : 83 E0 01
000abb87: jmp .+0x00000011 (0x000abb9a=sendModifyArg:toRcvr:@3A) : EB 11
000abb89: nop : 90
000abb8a: nop : 90
000abb8b: nop : 90
entry:
000abb8c: movl %edx, %eax : 89 D0
000abb8e: andl $0x00000003, %eax : 83 E0 03
000abb91: jnz .+0xfffffff1 (0x000abb84=sendModifyArg:toRcvr:@24) : 75 F1
000abb93: movl %ds:(%edx), %eax : 8B 02
000abb95: andl $0x003fffff, %eax : 25 FF FF 3F 00
000abb9a: cmpl %ecx, %eax : 39 C8
000abb9c: jnz .+0xffffffe0 (0x000abb7e=sendModifyArg:toRcvr:@1E) : 75 E0
```

Figure 3.13 shows the format of an object header in the Spur memory manager used by the Cog VM, which is what the class tag is extracted from. Note that the object header format is actually almost the same between the 32 bit and 64 bit memory manager - in the 32 bit version, the header is made up of two 32 bit words. The relevant piece of information here is the 22 bit class index, which is used as the class tag stored in the **ecx** register.

s	s	s	s	s	s	s	s	x	x	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h
x	x	x	f	f	f	f	f	x	x	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
31			28				24			21																				1	0

Word (32 bit)	Bit(s)	Meaning
1	24-31	Number of Slots
1	22 - 23	Unused
1	0 - 21	Identity Hash
2	22 - 23, 29 - 31	Unused
2	24 - 28	Object Format
2	0 - 21	Class Index

Figure 3.13: Spur Object Header Format

Immediate Object (SmallInteger)

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1

Immediate Object (Character)

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	0

Pointer to Heap Allocated Object

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0

Figure 3.14: Spur Object Pointer Formats

Another necessary piece of information is the pointer format, shown for 32 bit Spur in Figure 3.14. Heap allocated objects are aligned such that every valid address is a multiple of 4 - this means that every valid pointer's last two least significant bits will be 0 and 0. The Spur memory manager uses this to encode what's known as immediate objects - any SmallInteger or Character instance is encoded directly in the address as opposed to being a pointer. SmallIntegers have 1 as the least significant bit, and Characters have 1 and 0 as the two least significant bits [8]. With this information, the execution of the assembly code after the entry: label can be followed. Recall that the **edx** register has a pointer to the receiver and the **ecx** register has a class tag in it. First, the pointer of

[8]In the 64 bit version of Spur, the 3 least significant bits are used to encode immediates, as all valid pointers are multiples of 8

the receiver is copied into the temporary register and is tested for whether it's actually a pointer by taking the last 2 bits and checking if they equal 0 - if they are not, execution jumps back (in this case to address abb84), and the two bits have an and operation applied on them with 1 - this produces 1 in **eax** if the receiver is a SmallInteger and 0 if the receiver is a Character (execution then jumps ahead to abb9a, which will be covered imminently). If the receiver is actually a pointer, the value at the pointer is copied to the temporary register **eax** - this is the header of the object. The class index, the last 22 bits of the object header, is then put into the temporary register. Finally, at address abb9a, the value of the temporary register (which is either the class index, 1 for a SmallInteger and 0 for a Character) is compared to the actual inline cache in **ecx** - execution jumps to a method lookup failure trampoline back to the interpreter if they don't match, and continues on if they do. The next section of the JIT compiled code is the actual building of the stack frame for `sendModifyArg:toRcvr:`.

```
noCheckEntry:
000abb9e: movl %ss:(%esp), %ebx : 8B 1C 24
000abba1: movl %edx, %ss:(%esp) : 89 14 24
000abba4: pushl %esi : 56
000abba5: pushl %edi : 57
000abba6: pushl %ebx : 53
000abba7: pushl %ebp : 55
000abba8: movl %esp, %ebp : 89 E5
000abbaa: pushl $0x000abb60=sendModifyArg:toRcvr:@0 : 68 60 BB 0A 00
IsAbsPCReference:
000abbaf: movl $0x0052d108=nil, %ebx : BB 08 D1 52 00
000abbb4: pushl %ebx : 53
000abbb5: pushl %edx : 52
000abbb6: movl %ds:0x7ffffdc8=&stackLimitFromMachineCode, %eax : A1 C8 FD FF 7F
000abbbb: cmpl %eax, %esp : 39 C4
000abbbd: jb .+0xffffffbd (0x000abb7c=sendModifyArg:toRcvr:@1C) : 72 BD
HasBytecodePC bc 16/17:
```

Note the label `noCheckEntry`: - recall that the entry: label was jumped to if the send was coming from an inline cache that needed to be checked. Execution jumps to `noCheckEntry`: if a cache does not need to be checked, for example, if execution is coming from the interpreter after JIT compiling the method for the first time, or coming from a send without an inline cache. In these cases, the class is known before calling the method. Continuing execution, it's worth noting, at the top of the stack, there would be a return instruction for the calling method, preceded by any arguments that could not fit into available registers. The first few instructions rearrange the stack so that the receiver and all of the arguments, including ones in registers, are on the call stack, with the return instruction on the top - this is the state as of instruction abba6. Next, similar to activation of methods as seen in the Stack Interpreter, the frame pointer is pushed onto the stack, and the stack pointer is copied to the frame pointer. Next, a reference to the method is pushed onto the stack, and a reference to the nil object is pushed as a placeholder for a heap-allocated context object. Finally, the receiver is pushed again. A check is made to see if the stack pointer has passed a stack overflow limit, and, if it has, execution jumps back up to the previous chunk of assembly code shown - wiping the receiver register before calling the method failure trampoline to go back to the interpreter (a blank receiver is used to flag the stack overflow state). With a stack frame built, finally, the JIT compiled version of the method body of `sendModifyArg:toRcvr:` can be shown.

```
000abbbf: movl 12(%ebp), %esi : 8B 75 0C
000abbc2: movl 8(%ebp), %edx : 8B 55 08
000abbc5: movl $0x01f51ee8=#modifyArg:, %ecx : B9 E8 1E F5 01
000abbca: call .+0xfff548a9 (0x00000478=ceSend1Args) : E8 A9 48 F5 FF
IsSendCall modifyArg: bc 19/20:
000abbcf: movl %ebp, %esp : 89 EC
000abbd1: popl %ebp : 5D
000abbd2: ret $0x000c : C2 0C 00
000abbd5: int3 : CC
000abbd6: int3 : CC
000abbd7: int3 : CC
000abbd8: int3 : CC
000abbd9: int3 : CC
000abbda: int3 : CC
000abbdb: int3 : CC
startpc: 16
  16rABBAF IsAbsPCReference (16rABBDF)
  16rABBBF HasBytecodePC    (16rABBDE, bc: 16)
  16rABBCF IsSendCall       (16rABBDD, bc: 19 #modifyArg:)
```

Recalling the bytecode implementation of sendModifyArg:toRcvr, the method was translated to 4 bytecode instructions - pushing temp 1 (the rcvr argument), pushing temp 0 (the arg argument), sending `modifyArg:` to temp 1 with temp 0 as an argument, and returning the top of the stack. The JIT compiler uses stack to register mapping to avoid stack based operations when registers can be used instead, so neither push is compiled to assembly. When the send `modifyArg:` bytecode is compiled, the receiver and argument are placed into the appropriate registers (recall that they were pushed onto the stack from registers during the stack frame build). The next instruction moves a pointer to the selector `modifyArg:` into the class tag register, and is followed by a call to a method lookup trampoline. As previously discussed, this will eventually be replaced with an inline cache as execution proceeds but, at JIT compilation, a receiver is not known so no cache is made. Finally, the returnTop bytecode is compiled into a simple sequence - first the stack pointer is set back to the frame pointer, and then the top of the stack (the saved frame pointer) is popped off the stack into the frame pointer. A return instruction is next, which consumes a saved instruction on the top of the stack to return execution back to the caller of the method. The remainder of the assembly shown is padding (the int3 instructions), as well as a lookup map that can be used by various processes to find certain instructions like message sends in the JIT-compiled code, as well as map from the JIT compiled code to the bytecode and vice versa.

Execution of Send 2 (cont.), and 3

Now that the JIT compiled version of `sendModifyArg:toRcvr:` has been explained, execution of Send 2 can continue on the flowchart. An enilopmart is called to take execution into machine code and the activation of the JIT compiled method. Remember that, since the method was newly JIT compiled, execution of the method begins at the `noCheckEntry`: label in the JIT code. The method is activated, and execution continues of assembly code until the call is hit - as the call does not have an inline cache, a method lookup trampoline is called to look up the new method. Following the flowchart and what has been revealed of the implementation, the method `modifyArg:` is looked up using the selector `modifyArg:` and the class `ReceivingObjectA`, and, as it is not yet JIT compiled, execution jumps back to the interpreter. However, `modifyArg:` for `ReceivingObjectA` is in the interpreter's method lookup cache and capable of being JIT compiled, so now this method is JIT compiled. Now, since `modifyArg:` for `ReceivingObjectA` has been JIT compiled, the method lookup trampoline will actually create an inline cache this time by using the JIT compiler to rewrite the call instruction.

```
000abbc5: movl $0x000011e2, %ecx : B9 E2 11 00 00
000abbca: call .+0x0000003d (0x000abc0c=modifyArg:@2C) : E8 3D 00 00 00
```

The moving of the selector `modifyArg:` into **ecx** has been replaced with moving the class index of `ReceivingObjectA` into **ecx**, and the call to the method lookup trampoline is replaced with the address of the entry: label in `modifyArg:`. Recalling back to

Chapter 2 and the high level discussion of the Cog VM - this inline cache is known as a monomorphic inline cache - it is monomorphic because it only applies to `modifyArg:` implemented in one class - `ReceivingObjectA`. An inline cache that applies to more than one class is referred to as a polymorphic inline cache. Monomorphic caches are specific to a send site. Polymorphic caches are specific to a selector, and can be used by multiple send sites. Since the cache is newly created, execution doesn't actually jump to the address used for the cache - it jumps to the `noCheckEntry`: label in `modifyArg:`. Execution of `modifyArg:` continues and returns through `sendModifyArg:toRcvr:`, which returns back to the enclosing code. Figure 3.15 shows the state of the call stack before performing a return from `modifyArg:` in Send 2[9].

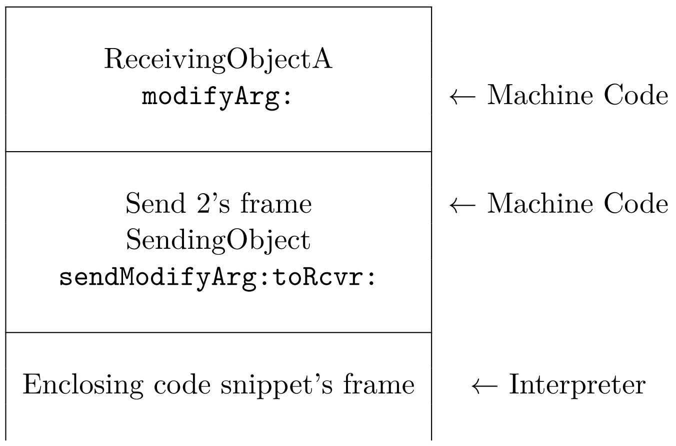


Figure 3.15: Stack before returning after Send 2

Continuing on to Send 3, execution returns back in the interpreter and then Send 3 is encountered. `sendModifyArg:toRcvr:` is in the method lookup cache, and, this time, is already JIT compiled, so execution can jump straight into machine code. Following along execution of sendModfifyArg:toRcvr:, the send to `modifyArg:` is encountered and execution of `sendModifyArg:toRcvr:` uses the inline cache to jump to the entry: label in ReceivingObjectA's JIT compiled code, where it successfully compares the receiver's class index to the cached class tag.

Send 4

At this point in time, a monomorphic inline cache has been created and used successfully in Send 3, and now Send 4 is about to be executed. However, Send 4 differs from the pre-

[9]Some details have been omitted - the method lookup trampoline will actually have a stack frame between sendModifyArg:toRcvr and `modifyArg:` but it's not relevant to this discussion

vious sends in that the second argument is receiverB, an instance of `ReceivingObjectB`. At this point, the flow of execution has been well established, so execution will jump ahead to the send of `modifyArg:` in `sendModifyArg:toRcvr:`. Since the send site has a monomorphic inline cache, execution jumps to `ReceivingObjectA`'s implementation of `modifyArg:`. However, this time, the class lookup fails as `ReceivingObjectB` has a different class index and execution jumps to a method failure trampoline. In the method failure trampoline, a lookup is done for the method that actually matches the combination of the selector `modifyArg:` and the class `ReceivingObjectB` - the method is JIT compiled if it has not been JIT compiled already, and execution continues to expand the existing cache. Since the cache is currently monomorphic, it is replaced with a polymorphic inline cache, which is a separate method that acts as a lookup table for all encountered classes that match a given selector. Polymorphic inline caches, or PICs, are not send site specific, so it's possible that a PIC will already exist for a given selector - in that case, the class will be added to an existing PIC for that selector. Once a PIC has either been created or assigned, the send site is linked to the PIC instead of directly to a method address. Here is the modified send site in `sendModifyArg:toRcvr:`.

```
000abbc5: movl $0x000011e2, %ecx : B9 E2 11 00 00
000abbca: call .+0x0000015d (0x000abd2c=modifyArg:@2C) : E8 5D 01 00 00
```

The class tag hasn't changed, but the address of the call has - it now goes to a PIC lookup method, some of which will be shown next.

```
ABD00
  nArgs: 1 type: 4
  blksiz: A8
  selctr: 1F51EE8=#modifyArg:
  cPICNumCases: 2 cpicHasMNUCase: no
000abd1c: xorl %ecx, %ecx : 31 C9
000abd1e: call .+0xfff54b65 (0x00000888=cePICAbort1Args) : E8 65 4B F5 FF
000abd23: nop : 90
000abd24: andl $0x00000001, %eax : 83 E0 01
000abd27: jmp .+0x00000011 (0x000abd3a=modifyArg:@3A) : EB 11
000abd29: nop : 90
000abd2a: nop : 90
000abd2b: nop : 90
entry:
000abd2c: movl %edx, %eax : 89 D0
000abd2e: andl $0x00000003, %eax : 83 E0 03
000abd31: jnz .+0xfffffff1 (0x000abd24=modifyArg:@24) : 75 F1
```

```
000abd33: movl %ds:(%edx), %eax : 8B 02
000abd35: andl $0x003fffff, %eax : 25 FF FF 3F 00
000abd3a: cmpl %ecx, %eax : 39 C8
000abd3c: jnz .+0x0000004a (0x000abd88=modifyArg:@88) : 75 4A
000abd3e: movl $0x0, %ebx : BB 00 00 00 00
000abd43: jmp .+0xfffffed6 (0x000abc1e=modifyArg:@3E) : E9 D6 FE FF FF
ClosedPICCase0:
...
ClosedPICCase4:
000abd88: movl $0x0, %ebx : BB 00 00 00 00
000abd8d: cmpl $0x000011e3, %eax : 3D E3 11 00 00
000abd92: jz .+0xffffff16 (0x000abcae=modifyArg:@3E) : 0F 84 16 FF FF FF
ClosedPICCase5:
000abd98: movl $0x000abd00=modifyArg:@0, %ecx : B9 00 BD 0A 00
000abd9d: jmp .+0xfff54c96 (0x00000a38=cePICMiss1Args) : E9 96 4C F5 FF
startpc: nil
```

Execution enters at the entry: label, and, like method lookup with the monomorphic cache, the class tag in **ecx** is compared to the receiver. However, if this check fails, execution jumps through each additional case, comparing the receiver to a cached class tag, jumping to a method if successful and proceeding to the next case if not. Eventually, if all cases fail, execution jumps to another cache failure trampoline which expands the PIC to add another class. There's a fixed limit on how many classes can be added to a PIC for a selector - when that happens, an open PIC is created instead of the closed PIC. Earlier in chapter two, these sends were referred to as megamorphic sends. Open PICs don't have a class limit, but megamorphic send sites are rare enough that they will not be explored further.

3.3.3 Tail Call Elimination in the JIT

Having thoroughly explored JIT compilation of methods and the first two levels of inline caching, some design decisions can be made about when and how to introduce tail call elimination. While the approach used in the Stack Interpreter could be maintained for interpreted methods in the Cog VM, following the flow of execution shows that executing interpreted methods may not be overly common - since methods are JIT compiled the second time they are sent, commonly used methods will only ever be run through the interpreter at fairly rare occurrences - after image startup and recompilation of the method come to mind. Therefore, adding tail call elimination for interpreted methods adds complexity without necessarily much gain, and will not be attempted. Similarly, adding tail call elimination for unlinked sends also may be unnecessary, as send sites tend to be linked at least with a monomorphic cache on first execution of the send. Tail call elimination will not be attempted for unlinked sends either. Monomorphic sends are the most common types of sends, with about 90% of send sites being monomorphic versus 9% polymorphic and 1% megamorphic [23]. Because of this, tail call elimination was prioritized for monomorphic send sites, and was not attempted for polymorphic and megamorphic send sites due to time constraints.

Identifying Tail Calls

In the Stack Interpreter, tail calls were identified at compilation time, and various techniques were attempted to reduce the overhead of implementing tail calls in the interpreter due to a lack of bytecode space. Since tail call elimination in the CoInterpreter will be skipped, tail call elimination in the Cog VM does not have this issue. Tail calls can be identified on JIT compilation by checking for a returnTop or returnBlock bytecode immediately following a send bytecode. The actual generated assembly code can use jump instructions as opposed to call instructions - as tail calls never need to return to their call stack, there is no need for a call instruction to push a return instruction pointer onto the stack [37]. The challenge encountered here are cases where, despite executing a jump instruction, there actually is a need for a return instruction to be at the top of the stack. There are several cases in the Cog VM where a return instruction on the top of the stack is expected - based on the above restrictions, they can be limited to the method lookup trampoline and the method lookup failure trampoline. The method lookup trampoline is called when a send site is unlinked - as stated above, tail call elimination will not be introduced for that case. A method failure trampoline is called when a class tag in a cache fails to match the receiver. As covered earlier - the check of a class tag happens in the linked method itself and relies on the presence of the sending method's return instruction to be at the top of the stack. For example, when `modifyArg:` was called with `ReceivingObjectB` in Send 4 in the above example, the class tag check failed in the implementation of `ReceivingObjectA`'s `modifyArg:`, and the method failure trampoline used the return instruction in `sendModifyArg:toRcvr:` to reference the selector being looked for. It is clear that there needs to be a way to use a jump instruction when performing a tail call, but also ensure that the return instruction is pushed onto the stack when needed. A solution, and the one used in this implementation, will be presented next.

Tail Call Elimination Implementation

The following listing is the JIT compiled version of `sendModifyArg:toRcvr:` with tail call elimination implemented. Interspersed with the unlinked assembly code will be the linked assembly code, as much of the changes are only used when the method is linked. Much of the compiled method is the same as the previously presented version - differences will be discussed as they come up. The following section of JIT compiled code is essentially identical (except for the addresses) - this covers the entry code and frame build, as well as the preparation for the send.

```
AFFB8
   objhdr: 8000000A000035
   nArgs: 2 type: 2
   blksiz: B8
   method: 29278C0
   mthhdr: 4100007
   selctr: 29278A0=#sendModifyArg:toRcvr:
   blkentry: 0
   stackCheckOffset: 5F/B0017
   cmRefersToYoung: no cmIsFullBlock: no
000affd4: xorl %edx, %edx : 31 D2
000affd6: call .+0xfff50745 (0x00000720=ceMethodAbort2Args) : E8 45 07 F5 FF
000affdb: nop : 90
000affdc: andl $0x00000001, %eax : 83 E0 01
000affdf: jmp .+0x00000011 (0x000afff2=sendModifyArg:toRcvr:@3A) : EB 11
000affe1: nop : 90
000affe2: nop : 90
000affe3: nop : 90
entry:
000affe4: movl %edx, %eax : 89 D0
000affe6: andl $0x00000003, %eax : 83 E0 03
000affe9: jnz .+0xfffffff1 (0x000affdc=sendModifyArg:toRcvr:@24) : 75 F1
000affeb: movl %ds:(%edx), %eax : 8B 02
000affed: andl $0x003fffff, %eax : 25 FF FF 3F 00
000afff2: cmpl %ecx, %eax : 39 C8
000afff4: jnz .+0xffffffe0 (0x000affd6=sendModifyArg:toRcvr:@1E) : 75 E0
noCheckEntrys:
000afff6: movl %ss:(%esp), %ebx : 8B 1C 24
000afff9: movl %edx, %ss:(%esp) : 89 14 24
000afffc: pushl %esi : 56
000afffd: pushl %edi : 57
```

```
000affe: pushl %ebx : 53
000afff: pushl %ebp : 55
000b0000: movl %esp, %ebp : 89 E5
000b0002: pushl $0x000affb8=sendModifyArg:toRcvr:@0 : 68 B8 FF 0A 00
IsAbsPCReference:
000b0007: movl $0x0052d108=nil, %ebx : BB 08 D1 52 00
000b000c: pushl %ebx : 53
000b000d: pushl %edx : 52
000b000e: movl %ds:0x7ffffdc8=&stackLimitFromMachineCode, %eax : A1 C8 FD FF 7F
000b0013: cmpl %eax, %esp : 39 C4
000b0015: jb .+0xffffffbd (0x000affd4=sendModifyArg:toRcvr:@1C) : 72 BD
HasBytecodePC bc 16/17:
000b0017: movl 12(%ebp), %esi : 8B 75 0C
000b001a: movl 8(%ebp), %edx : 8B 55 08
```

What follows is the first difference, between the tail call elimination version of the method, and the non tail call elimination version - there is an extra jump. When unlinked, the jump goes nowhere, and execution proceeds to the original implementation of the send, where the selector is moved to the class tag and the lookup trampoline is called. This ensures that a return address is still on the top of the stack when the trampoline is called. The return instructions are also kept, so that, when the method is eventually returned to, the return of the top of the stack still works.

```
Unlinked:
000b001d: jmp .+0x00000000 (0x000b001f=sendModifyArg:toRcvr:@67) : EB 00
000b001f: movl $0x01f51ee8=#modifyArg:, %ecx : B9 E8 1E F5 01
000b0024: call .+0xfff5044f (0x00000478=ceSend1Args) : E8 4F 04 F5 FF

Linked:
000b001d: jmp .+0x00000017 (0x000b0036=sendModifyArg:toRcvr:@7E) : EB 17
000b001f: movl $0x000011e2, %ecx : B9 E2 11 00 00
000b0024: call .+0x00000073 (0x000b009c=modifyArg:@2C) : E8 73 00 00 00

IsSendCall modifyArg: bc 19/20:
000b0029: movl %ebp, %esp : 89 EC
000b002b: popl %ebp : 5D
000b002c: ret $0x000c : C2 0C 00
```

When the send is actually linked and is a monomorphic send, the **jmp** instruction is overwritten to jump ahead, in this case to address 000b0036. Essentially, it enters the class tag lookup that it would typically enter at the start of the receiving method. However, as it has been moved to the sending method instead, the class check can abort to the original implementation of the call if it fails, ensuring that the return instruction will be on top of the stack when calling the cache failure trampoline. The following section begins the implementation of the class tag lookup, as adapted to a tail call - entering at address 000b0036.

```
000b002f: jmp .+0xffffffee (0x000b001f=sendModifyArg:toRcvr:@67) : EB EE
000b0031: andl $0x00000001, %eax : 83 E0 01
000b0034: jmp .+0x0000001d (0x000b0053=sendModifyArg:toRcvr:@9B) : EB 1D
000b0036: movl -8(%ebp), %eax : 8B 45 F8
000b0039: cmpl $0x0052d108=nil, %eax : 3D 08 D1 52 00
000b003e: jnz .+0xffffffdf (0x000b001f=sendModifyArg:toRcvr:@67) : 75 DF
```

The operations at 000b0036 to 000b003e first check for something completely unrelated - in chapter 2, stack page management was briefly mentioned in the Stack Interpreter. The call stack is divided up into stack pages that are linked together via heap allocated context objects, and each page of stack has an upper limit on the number of frames allowed - if a stack frame goes over the limit, a stack overflow condition happens and the frame is allocated on a new page instead. Recall that in the Stack Interpreter and in the Cog Interpreter, slots are included in stack frames for a pointer to a context object and are typically nil. A frame that is at the base of a stack frame needs to have a populated context object, as the context object will have the reference back to the previous page. In this implementation, the complexity of trying to rewrite a stack frame for a different method with an existing context object is avoided by this check - if the context object is not nil, execution jumps to the original implementation. If the context object is nil, execution continues through the class tag lookup.

```
Unlinked:
000b0040: movl $0x01f51ee8=#modifyArg:, %ecx : B9 E8 1E F5 01

Linked:
000b0040: movl $0x000011e2, %ecx : B9 E2 11 00 00

000b0045: movl %edx, %eax : 89 D0
000b0047: andl $0x00000003, %eax : 83 E0 03
000b004a: jnz .+0xffffffe5 (0x000b0031=sendModifyArg:toRcvr:@79) : 75 E5
000b004c: movl %ds:(%edx), %eax : 8B 02
000b004e: andl $0x003fffff, %eax : 25 FF FF 3F 00
```

```
000b0053: cmpl %ecx, %eax : 39 C8
000b0055: jnz .+0xffffffd8 (0x000b002f=sendModifyArg:toRcvr:@77) : 75 D8
```

The class tag lookup code is essentially identical to its implementation in the entry code. In its unlinked state, the selector is copied to **ecx** - this code should never be reached as execution will never jump here when unlinked. When linked, the class tag is copied to **ecx**, like in the original implementation of the call. Next, like in the entry code for the receiving method, the class index of the receiver (or the least significant bit, if it's an immediate object) is compared to the class tag in **ecx** - if they don't match, execution jumps back to the original send implementation so that the call can put the return instruction on the top of the stack. Ultimately, if the monomorphic inline cache is replaced with a polymorphic cache, the **jmp** at 000b001d is simply reverted back to a jump of 0 to fall back to the original implementation. Note that this choice of implementation does cause the class check to be executed redundantly, but does ensure that the stack is in the correct state. If the class check does match execution proceeds to the tail call.

```
000b0057: movl %ebp, %esp : 89 EC
000b0059: popl %ebp : 5D
000b005a: popl %eax : 58
000b005b: addl $0x0000000c, %esp : 83 C4 0C
000b005e: pushl %eax : 50

Unlinked:
000b005f: jmp .+0xfff50414 (0x00000478=ceSend1Args) : E9 14 04 F5 FF

Linked:
000b005f: jmp .+0x0000004a (0x000b00ae=modifyArg:@3E) : E9 4A 00 00 00

IsSendCall:
000b0064: int3 : CC
000b0065: int3 : CC
000b0066: int3 : CC
000b0067: int3 : CC
000b0068: int3 : CC
000b0069: int3 : CC
startpc: 16
  16rB0007 IsAbsPCReference (16rB006F)
  16rB0017 HasBytecodePC    (16rB006E, bc: 16)
  16rB0029 IsSendCall       (16rB006D, bc: 19 #modifyArg:)
```

```
16rB0064 IsSendCall      (16rB006B, bc: 0)
```

Finally, having ensured that the class tag matches the receiver, the tail call itself is implemented. Similar to the Stack Interpreter implementation, the tail call is implemented such that the current stack frame is popped, and the arguments and receiver that were on the stack are consumed and replaced with the arguments and receiver of the tail call. First, the frame pointer is copied into the stack pointer, and then the saved frame pointer on the top of the stack is popped into the frame pointer. Now, the saved instruction pointer of the caller of the method with the tail call is on top of the stack, followed by the arguments and the receiver used to call the method with the tail call. Recall that, as part of the JIT compiled `noCheckEntry` labelled code that builds a stack frame, the arguments and receiver are pushed onto the stack from registers if there are enough registers to hold the arguments - this means that when doing the tail call, if there are enough registers to hold the arguments, the tail call only needs to consume the existing arguments. In this case, `modifyArg:` has only one argument, so its argument will be re-pushed onto the stack from a register when it is activated along with the receiver. This is implemented by popping the saved instruction pointer into a temp register, shifting the stack pointer up by the number of arguments + the receiver (consuming the arguments and the receiver), and then pushing the return instruction back on top of the stack. Finally, instead of a call, the tail call is executed as a **jmp** - the unlinked version will never be called, so the linked version is the only one worth considering. Because the class lookup has already occurred, the **jmp** jumps to the `noCheckEntry` label in the destination method, skipping a redundant class check. The cost to this approach is the extra space usage - each tail call has significantly more instructions with this implementation, and some ideas on how to work around this will be discussed later on.

3.3.4 Garbage Collection and Method Compaction

Complications to this implementation of tail call elimination arise in the form of method compaction and garbage collection. JIT compiled machine code exists in a limited amount of memory - when this space fills, the Cog VM performs compaction of JIT compiled methods by freeing older methods - the remaining methods are then shifted into the space made available. Since calls and jumps in JIT code use offsets to the actual address they jump to, any linked send needs to be relinked to a new offset depending on where the method has moved to. Also, any linked send to a method which has been freed needs to be unlinked. In garbage collection, linked sends may also have to be unlinked if the target method or PIC has been freed. Since the tail call elimination implementation of a linked send is quite different than that of a non tail call method, different steps need to be followed to unlink a tail call. Both the original call address and the jump address need to be set back to the trampoline, and both the original cache tag and the second cache tag in the class lookup machine code need to be set back to the selector as well. Also, the jump to the class lookup needs to be set back to 0. The VM also needs to test each send site to see if it's a tail call before deciding on what steps to take - deciding whether a call is a tail call or not is simply a matter of checking for some known bytes around the call, such whether a call and class tag is preceded by a jump. This complexity adds overhead to the implementation of tail call elimination.

3.3.5 Debugging

Recall that, in the above implementations, tail call elimination is not performed when there is a context object present. This was a work around to avoid tail call elimination for base frames in a stack page. A side effect of this implementation is that, when debugging, tail call elimination is not performed as context objects are created for each frame. Recall from the previous chapter that maintaining proper stack traces was a reason for not implementing tail call elimination in Python - this side effect points to a mechanism for maintaining proper stack traces when debugging.

3.4 Designing Tests for Tail Call Elimination

The effectiveness of tail call elimination will be tested by executing test programs and timing the execution of these programs using Smalltalk's facilities. The different implementations will be tested and compared to each other, and to versions of the Stack Interpreter and Cog VM compiled without tail call elimination turned on. The choice of tests is designed to show both the ideal performance and real world performance. Ideal performance is tested by demonstrating tail call elimination in the special case of tail call recursion. Real world performance is tested by running a computationally significant process - the recompilation of every method in the system was chosen as a sufficiently complex method to execute. Testing will be done for execution time and memory usage.

Tests are run on an Apple Macbook Pro (13 inch late 2013) with a dual core Intel Core i5 processor running at 2.6GHz. The test machine has 16gb of 1600MHz DDR2 memory. The operating system version is macOS High Sierra (v10.13.6). Tail call elimination was implemented onto OpenSmalltalk-VM obtained on January 24th, 2019[30]. Unless otherwise stated, no parameters were changed from the default VM parameters.

3.5 Test Runner

Creating a framework for testing and recording results is the first challenge. As Smalltalk code is run in a live environment with garbage collection events, and is running in an operating system with its own events that can affect the availability of resources, it's important to build a test suite that can run a test multiple times for later analysis.

3.5.1 Execution Time

The following listing is a Smalltalk method which will be used as a test runner to track execution time of a test over the course of multiple executions of that test. We will refer to each execution of a test as a run.

```
ThesisTestSuite class>>runBlock: ablock runs: r
  | execution |
  execution := ThesisTestExecution new.
  execution block: ablock; executions: r.

  r timesRepeat:[
    |time|
    time := Time millisecondsToRun:ablock.
    execution addTime:time.
    ].
  ↑execution
```

This method is implemented as a class method for a class called `ThesisTestSuite`. It takes a block of code and the number of runs as arguments. It creates an instance of a class, `ThesisTestExecution`, which is a container for information about the current test - it stores the block and the number of runs in instance variables. The method then runs the block by the number of runs, and the time in milliseconds of each run of the block is saved in a collection in the instance of `ThesisTestExecution`. Finally, when completed, the instance is returned.

3.5.2 Memory

A Smalltalk VM offers methods to get information about the current state of the image and memory usage, which provides a simple way of getting a snapshot of memory usage during a test. However, memory usage is best monitored during execution of the test, not after, to get a sense of how the test affects memory consumption. As a result, memory usage will be tracked by saving the current memory usage in a file at key intervals during method execution and then comparing memory usage afterwards - the existing test suite will be reused for memory tests.

3.6 Tail Call Recursion

The first set of tests will show examples of tail recursive behaviour, using common examples of tail recursive calculations as well as an implementation of a Smalltalk control structure in a tail recursive form.

3.6.1 Factorial

The first test will be the tail recursive implementation of factorial shown back in chapter 1. Here is the implementation again, this time as a class method on a class called `ThesisTestSuite`.

```
ThesisTestSuite class>>factorial: n
   ↑self factorial:n accumulator: 1

ThesisTestSuite class>>factorial: n accumulator: i
   (n = 0) ifTrue:[↑i]
   ifFalse:[↑self factorial: (n-1) accumulator: (i * n)].
```

Here is compiled bytecode for `factorial:accumulator:`, demonstrating clearly the tail call to factorial:accumulator as the last instruction before the return to the top of the stack.

```
21 <10> pushTemp: 0
22 <75> pushConstant: 0
23 <B6> send: =
24 <99> jumpFalse: 27
25 <11> pushTemp: 1
26 <7C> returnTop
27 <70> self
28 <10> pushTemp: 0
29 <76> pushConstant: 1
30 <B1> send: -
31 <11> pushTemp: 1
32 <10> pushTemp: 0
33 <B8> send: *
34 <F0> send: factorial:accumulator:
35 <7C> returnTop
```

Execution Time

The method to measure execution time in Smalltalk has a precision of one millisecond. Therefore, rather than measure the execution time of one calculation of the factorial of n, we will measure the total execution time of several iterations of calculating the factorial of n. When discussing these tests, we will refer to the number of repeated calculations that make up a test as an iteration. We can say that a test is made up of one or more iterations of an action (a calculation, or a method call), and each run of a test collects a measurement.

Two values of factorial will be calculated - the factorial of 500 and the factorial of 5000 - this is to verify if there is any significant performance in performance between smaller and larger call stacks. Here is the factorial test suite for measuring performance speed - note that the block run in `ThesisTestSuite` contains an inner `timesRepeat:` block. The inner `timesRepeat:` block is where the factorial of n is calculated by the number of iterations, and the measurement of the execution time is taken at the completion of all iterations. The method `report:name:testtype:n:iterations:runs:` is a utility method for writing the results of each run to a file - the implementation will not be shown here.

```
ThesisTestSuite>>runFactorial: n iterations: i runs: r
   | execution testname testtype |
   testname := 'factorial'.
   testtype := 'execution'.
   execution := ThesisTestSuite runBlock:[
      i timesRepeat:[ThesisTestSuite factorial:n]
      ] runs: r.
   self report: execution name: testname
      testtype: testtype n: n iterations: i runs: r.

"Factorial 500"
ThesisTestSuite runFactorial:500 iterations:1000 runs: 100.

"Factorial 5000"
ThesisTestSuite runFactorial:5000 iterations:100 runs: 25.
```

Memory Usage

To track memory usage during the execution of a tail recursive algorithm, a slight modification of the existing `factorial:accumulator:` method is made. Now, when the base case of factorial is reached, the current memory usage (based on the same calculation done by the Smalltalk image when displaying memory usage) is stored into a class variable `Memory`.

```
ThesisTestSuite class>>factorialMem: n accumulator: i
   (n = 0) ifTrue:[Memory := self getMemory. ↑i]
   ifFalse:[↑self factorialMem: (n-1) accumulator: (i * n)].

factorialMem: n
   ↑self factorialMem:n accumulator: 1
```

What follows is the listing of the test suite with modifications to save memory measurements into a file. The methods **`startFileName:type:n:iterations:runs:`** and **`recordValue:value:`** are utility methods for creating and recording memory values to a file and will not be included here. Note that the tests are run with only one iteration, as we are measuring total memory usage at a given point in a test, not a cumulative measurement like execution time. Instead, we can simply execute more runs.

```
ThesisTestSuite class>>runFactorialMem: n each: m times: iterations
   | execution file |
   file := self startFileName:'factorial' type: 'memory' n: n
      iterations: i runs: r.

   execution := ThesisTestSuite runBlock:[
      i timesRepeat:[
         ThesisTestSuite factorialMem:n.
         self recordValue: file value: Memory asString
      ]
   ] runs: r.

   file close.

"Factorial 500"
ThesisTestSuite runFactorialMem:500 iterations:1 runs: 1000.

"Factorial 5000"
ThesisTestSuite runFactorialMem:5000 iterations:1 runs: 250.
```

3.6.2 Fibonacci Sequence

Another well known recursive algorithm is the calculation of a specific number in the Fibonacci sequence. Like the implementation of factorial shown above, the Fibonacci sequence can be calculated using recursion with and without tail recursion. For this test, a tail recursive implementation of the Fibonacci sequence will be used, and is presented in the following listing.

```
ThesisTestSuite class>>fibonacci: n
  ↑self fibonacci: n a: 0 b: 1.

ThesisTestSuite class>>fibonacci: n a: a b: b
  (n = 0) ifTrue:[↑a].
  (n = 1) ifTrue:[↑b].
  ↑self fibonacci: n - 1 a: b b: a+b
```

The compiled bytecode for this method will be omitted, but the tail recursive call of `fibonacci:a:b` is apparent in the listing. For values of n greater than 1, a tail call is made back to `fibonacci:a:b` with n-1, tracking the result in the arguments a and b.

Execution Time

Like the factorial test above, the Fibonacci sequence for a given number will be calculated a number of times corresponding to the number of iterations. A measurement of the execution time will be taken after the number of iterations are complete, and the entire test will be repeated with multiple runs. Also similar to the factorial test, both a lower and higher number will be tested with Fibonacci to detect any difference in smaller vs larger numbers of created stack frames. The following listing is the test runner and the two tests for the Fibonacci sequence.

```
ThesisTestSuite class>>runFactorial: n iterations: i runs: r
  | execution testname testtype |
  testname := 'factorial'.
  testtype := 'execution'.
  execution := ThesisTestSuite runBlock:[
    i timesRepeat:[ThesisTestSuite factorial:n]
    ] runs: r.

  self report: execution name: testname testtype: testtype n: n
    iterations: i runs: r.
```

```
"Fibonacci 1000"
ThesisTestSuite runFibonacci:1000 each:1000 times: 100.

"Fibonacci 10000"
ThesisTestSuite runFibonacci:10000 each:100 times: 25.
```

Memory Usage

Memory usage for the Fibonacci sequence will be measured in the same way as factorial - a memory measurement will be taken at the end of each recursive case. The following listing shows the modifications made to the tail recursive Fibonacci sequence implementation, with memory measurements at each base case. Also shown is the modifications made to the test runner and the tests to be run - the modifications made are similar to those made to the factorial test runner.

```
ThesisTestSuite class>>fibonacciMem: n a: a b: b
   (n = 0) ifTrue:[Memory := self getMemory.↑a].
   (n = 1) ifTrue:[Memory := self getMemory.↑b].
   ↑self fibonacciMem: n - 1 a: b b: a+b.

ThesisTestSuite class>>fibonacciMem: n
   ↑self fibonacciMem: n a: 0 b: 1.

ThesisTestSuite class>>runFibonacciMem: n iterations: i runs: r
   | execution file |

   file := self startFileName:'fibonacci' type: 'memory' n: n
      iterations: i runs: r.
   execution := ThesisTestSuite runBlock:[
      i timesRepeat:[
         ThesisTestSuite fibonacciMem:n.
         self recordValue: file value: Memory asString
      ]
   ] runs: r.
   file close.

"Fibonacci 2000"
ThesisTestSuite runFibonacciMem:1000 iterations:1 runs: 1000.
```

```
"Fibonacci 10000"
ThesisTestSuite runFibonacciMem:10000 iterations:1 runs: 250.
```

3.6.3 WhileTrue Recursive Loop

As mentioned in chapter 1, Smalltalk does not have typical control structures for iteration and conditions such as for loops, while loops and if-then-else blocks. Instead, control structures are implemented by sending messages to blocks of code, which function as closures and as first class objects. Many of these methods are highly optimized by the compiler and are actually compiled to more traditional imperative control structures. However, implementing tail call elimination may present the opportunity to compile some control structures to bytecode that maintains the semantics of Smalltalk, while still executing efficiently. The next test presents an alternate implementation of a common Smalltalk control structure - `whileTrue:`. This alternate implementation given the selector `whileTrueTail:` in order to differentiate it to the compiler from the original implementation, so that the compiler does not try any additional optimizations. This control structure is implemented as an instance method for `BlockClosure`, or blocks of code, and is given a block of code as an argument - if the block instance evaluates to true, the block passed in as an argument will be evaluated, and then the method will be called again with the same block argument.

```
BlockClosure>>whileTrueTail: aBlock
  ↑ self value ifTrue: [
     aBlock value.
     ↑ self whileTrueTail: aBlock
  ]
```

Execution Time

Next is the test runner for execution time for this method. The call to `whileTrueTail:` is used as part of a simple loop structure, as seen in the following listing in the argument to `runBlock:`. A local variable `j`, initialized to 0, is incremented while the condition `j < n` is true. Once this condition fails, the loop is repeated by a specified number of iterations. Once the iterations complete, a measurement of execution speed is taken, and the entire test is repeated for the specified number of runs. The test will be run for both a smaller value of n and a larger value of n, which are specified in the listing as well.

```
ThesisTestSuite>>runWhileTrueTail: n iterations: i runs: r
   | execution testname testtype |
   testname := 'whiletrue'.
   testtype := 'execution'.
   execution := ThesisTestSuite runBlock:[ i timesRepeat:[
      | j |
      j := 0.
      [j<n] whileTrueTail:[j := j + 1]
      ].
   ] runs: r.

   self report: execution name: testname testtype: testtype n:n
      iterations: i runs: r.

"WhileTrue 1000"
ThesisTestSuite runWhileTrueTail:1000 iterations:10000 runs: 100.

"WhileTrue 10000"
ThesisTestSuite runWhileTrueTail:10000 iterations:1000 runs: 25.
```

Memory Usage

For the memory usage test, a modification is made to the implementation of the test `whileTrueTail:` - a memory measurement will be taken when the condition fails and the argument block is no longer repeated. The memory measurement is stored in a class variable of the `ThesisTestSuite` class, and is recorded in a file prior to the next run. Included in this listing is the modified test runner and the actual tests to be run.

```
BlockClosure>>whileTrueTailMem: aBlock
   ↑ self value ifTrue: [
      aBlock value.
      ↑ self whileTrueTailMem: aBlock
      ] ifFalse:[
         ThesisTestSuite setMemory
      ].

ThesisTestSuite>>runWhileTrueTailMem:n iterations: i runs: r
   | execution file |

   file := self startFileName:'whiletrue' type: 'memory' n: n
```

```
    iterations: i runs: r.

  execution := ThesisTestSuite runBlock:[
    i timesRepeat: [
      |j|
      j := 0.
      [j<n] whileTrueTailMem:[j := j + 1].
      self recordValue: file value: Memory asString.
    ]
  ] runs: r.

  file close.

"WhileTrue 1000"
ThesisTestSuite runWhileTrueTailMem:1000 iterations:1 runs: 1000.

"WhileTrue 10000"
ThesisTestSuite runWhileTrueTailMem:10000 iterations:1 runs: 250.
```

3.7 Real World Scenarios

Testing the effect of tail call elimination on real world performance is done by selecting computationally significant methods in a Smalltalk image and measuring execution time.

3.7.1 Compiler

For this test, the method `recompileAll` has been chosen - this method cycles through every class and trait in the Smalltalk image and calls the method `compileAll` which, as the name suggests, compiles every method that is part of that class or trait. Progress is displayed with a progress bar on the display. Listed below is the implementation of `recompileAll`.

```
Compiler>>recompileAll
  Smalltalk allClassesAndTraits
    do: [:classOrTrait | classOrTrait compileAll]
    displayingProgress:[:classOrTrait| 'Recompiling␣', classOrTrait]
```

Execution Time

Running `recompileAll` takes some time to execute, so the measurement taken will be the milliseconds to execute one recompileAll, averaged over 10 runs. Here is the test runner method.

```
ThesisTestSuite>>runCompileAllRuns: r
  | execution testname testtype|
  testname := 'compileall'.
  testtype := 'testtype'.
  execution := ThesisTestSuite runBlock:[Compiler recompileAll] runs: r.
  self report: execution name: testname testtype: testtype
    iterations: 1 runs: r.

"Compile x10"
ThesisTestSuite runCompileAllRuns: 10.
```

Each test will be run immediately on loading the Smalltalk image. Execution is logged to a file contain the results of each run, and information about the VM used for the test. Compile x10 will be run for both Stack Interpreter implementations and JIT implementations.

Memory Usage

In order to monitor the memory usage of `recompileAll:`, a method needs to be devised to check memory usage at key intervals during the execution. We can make a modification to sample memory during the method by making our own implementation of the method - in this implementation, a memory sample will be taken after the recompilation of each class. The following is the listing and the test runner for this test.

```
ThesisTestSuite>>recompileAll: file

  Smalltalk allClassesAndTraits
    do: [:classOrTrait | classOrTrait compileAll.
      file nextPutAll: (self getMemory) asString , Character cr asString.
    ]
    displayingProgress:[:classOrTrait| 'Recompiling␣', classOrTrait].

ThesisTestSuite>>runCompileAllMemRuns: r
  | execution file |
  file := self startFileName:'compileAll' type: 'memory' iterations: 1
    runs: r.
  execution := ThesisTestSuite runBlock:[
    self recompileAll: file
  ] runs: r.
```

"Compile x10"
```
ThesisTestSuite runCompileAllMemRuns: 10.
```

3.7.2 Browse Test

The next test is designed to simulate interaction with the user interface. It takes a class as a parameter and sends the browse message to each subclass of that class. The browse message opens up a browser window, which is the primary development interface in Smalltalk for adding and modifying classes and methods. After each browser window is opened, a message is sent to do an update of the user interface - this allows each browse windows to be drawn on the screen without waiting for the entire do loop to complete. Once all of the browse windows are open, all open instances of browse windows are sent a delete message to close all of the windows. Again, after each window is closed, the user interface is redrawn. The following listing shows the code for this test, with class as a parameter. For the purpose of this test, the class Number has been chosen. The Number class has nine subclasses in the Squeak image that is being used for testing.

```
ThesisTestSuite>>windowTestWithClass: class
  class allSubclassesDo:[ :a |
    a browse.
    self currentWorld doOneCycle
    ].
  Browser allInstancesDo:[:a |
    (a dependents select:[:
      each | each isMemberOf: PluggableSystemWindow ]
    ) do:[: each|
      each delete.
      self currentWorld doOneCycle
      ].
    ].
```

Execution Time

The measurement taken will be the milliseconds to execute one cycle of opening and closing all browser windows averaged over 10 runs. Here is the test runner method.

```
ThesisTestSuite>>runWindowsTest: class runs: r
  | execution testname testtype |
  testname := 'windows'.
  testtype := 'execution'.
  execution := ThesisTestSuite runBlock:[
    self windowTestWithClass:class.
    ] runs: r.

  self report: execution name: testname testtype: testtype
    n:class iterations: 1 runs: r.

"Windows x10"
ThesisTestSuite runWindowsTest: Number runs:10.
```

Memory Usage

To measure memory usage, a modification will be made to the windows test - memory measurements will be taken after opening all browse windows and after closing the windows. Here is the modified version of the test, and the modified test runner to be used for this test.

```
ThesisTestSuite>>windowTestMemWithClass: class file: file
  class allSubclassesDo:[ :a | a browse. self currentWorld doOneCycle].
  self recordValue: file value: (self getMemory) asString.
  Browser allInstancesDo:[:a |
  (a dependents select:[: each | each isMemberOf: PluggableSystemWindow ])
  do:[: each| each delete. self currentWorld doOneCycle].
  self recordValue: file value: (self getMemory) asString.
  ].

ThesisTestSuite>>runWindowsTestMem: class runs: r
  | execution file |

  file := self startFileName:'windows' type: 'memory'
    n: class iterations: 1 runs: r.
```

```
   execution := ThesisTestSuite runBlock:[
      self windowTestMemWithClass:class file: file.
      ] runs: r.

   file close.

"Windows x10"
ThesisTestSuite runWindowsTestMem: Number runs:10.
```

Chapter 4

Results

This chapter will discuss the results of the experiments laid out in the previous chapter. The first section will include statistics on how many tail calls are in a Smalltalk image statically in compiled code, and how many tail calls are actually called dynamically. The second section will discuss the results of the tail recursive tests, and the third section will discuss the results of the real world test. Each test result section will be broken down into execution time and memory, and will be followed with a discussion of the results.

Refer to Table 4.1 and Table 4.2 for the full chart of tests to be run and Table 4.3 for the list of virtual machine implementations to be tested. As explained, iterations refer to the number of times an individual calculation is run before collecting a measurement, and runs refer to the number of times a test is run, with each run corresponding to a measurement. Mean results are presented in a visual format. In addition, the mean, standard deviation, and median for each test are presented in a table. Percentage improvement for tail call eliminating implementations is also included, with the column headers %Imp for the mean improvement, and %SD for the percentage of uncertainty in the improvement. The formulas to calculate the percentage improvement and the uncertainty in the improvement are below.

$$imp_{ab} = \frac{mean_b - mean_a}{mean_b} * 100 \tag{4.1}$$

$$unc_{ab} = \frac{\sqrt{\sigma_a^2 + \sigma_b^2}}{mean_b} * 100 \tag{4.2}$$

where:

σ_a = standard deviation of tail call eliminating version

σ_b = standard deviation of non tail call eliminating version

$mean_b$ = mean of non tail call eliminating version

$mean_a$ = mean of tail call eliminating version

$imp_a b$ = percentage of difference of a relative to b

$unc_a b$ = percentage of uncertainty in difference from a relative to b

Test Name	Iterations	Runs
Factorial 500	1000	100
Factorial 5000	100	25
Fibonacci 1000	1000	100
Fibonacci 10000	100	25
whileTrue 1000	10000	100
whileTrue 10000	100	25
CompileAll	1	10
Windows	1	10

Table 4.1: Execution Tests

Test Name	Iterations	Runs
Factorial 500	1	1000
Factorial 5000	1	250
Fibonacci 1000	1	1000
Fibonacci 10000	1	250
whileTrue 1000	1	1000
whileTrue 10000	1	250
CompileAll	1	10
Windows	1	10

Table 4.2: Memory Tests

4.1 Tail Call Statistics

The number of potential tail calls that could be eliminated can be looked at from two perspectives - static and dynamic. In this case, static refers to compiled Smalltalk code and dynamic refers to the number of tail calls actually executed at runtime.

4.1.1 Static Tail Calls

Statically, the number of tail calls can be determined by examining each instance of `CompiledMethod` in a new Smalltalk image and counting each send bytecode in tail

Code	Implementation	Tail Call Elimination
si	Stack Interpreter	N
sitce	Stack Interpreter	Y
cog	Cog JIT Compiler	N
cogtce	Cog JIT Compiler	Y

Table 4.3: Implementations to Test

position, that is to say, immediately followed by a `returnTop` bytecode. For a sense of how tail call elimination may benefit a specific operation, counting tail calls could be limited to a specific package, or set of packages. As testing will be done using the

Bytecode	Packages	Tail Calls	Total	Percentage
131 to 134	All	3723	46240	8.05
176 to 191	All	1606	60142	2.67
192 to 207	All	1544	58207	2.65
208 to 255	All	18289	243382	7.51
All	All	25162	407971	6.17
131 to 134	Compiler	249	1785	13.95
176 to 191	Compiler	79	1649	4.79
192 to 207	Compiler	49	905	5.41
208 to 255	Compiler	486	4408	11.03
All	Compiler	863	8747	9.87

Table 4.4: Static Tail Call Counts

`recompileAll:` method in the `Compiler` class, static tail call numbers for methods in the compiler-related packages will be given, along with totals in Table 4.4. In addition, the total number of calls in the image are given to provide a sense of context. Refer back to Table 3.2 for the difference between the bytecode sets. The static tail call statistics show that 6.17% of compiled calls in methods are tail calls. There is a definite difference between different types of methods, as keyword messages have the highest prevalence of tail calls, while the math operator and special selector bytecode sets have a lower prevalence.

4.1.2 Dynamic Tail Calls

The static count of tail calls in methods may not be reflective of how many tail calls are actually executed, as not all methods may be executed at the same rate. Dynamically, the number of tail calls can be counted as they are executed. As a baseline, the amount of executed tail calls can be counted immediately after image startup, and can then be counted again after an action. In this case, the total tail calls will be counted after running `recompileAll:`, to give a sense of how performance may be affected.

Bytecode	Action	Tail Calls	Total	Percentage
131 to 134	Startup	3371	13198	25.54
176 to 191	Startup	1525	10991	13.87
192 to 207	Startup	3938	13036	30.21
208 to 255	Startup	38835	181829	21.36
All	Startup	47669	219054	21.76
131 to 134	Recompile	12131067	34943753	34.72
176 to 191	Recompile	1024145	25987376	3.94
192 to 207	Recompile	5292921	111534740	4.75
208 to 255	Recompile	73593216	378784134	19.43
All	Recompile	92041349	551250016	16.70

Table 4.5: Dynamic Tail Call Counts

Table 4.5 shows the total tail calls on image startup and after running `recompileAll:`. Again, the total dynamic sends are included to give context. The dynamic statistics present a different picture of tail call prevalence, with 21.76% of all calls on image startup being tail calls. Interestingly, the bytecode set with the highest prevalence of executed tail calls is the special selector set, despite the fact that, statically, relatively few calls using this bytecode set were tail calls. After running `recompileAll:`, the totals better reflect the static prevalence of tail calls, with keyword message bytecodes having significantly more tail calls. Speculatively, the prevalence of tail calls in the extended send set could be due to the number of keyword messages in the `Compiler` class with more than two arguments.

4.2 Tail Recursion Tests

In this section, the result for the tail recursive tests are presented. For each test, the results of the execution test will be shown, followed by the result of the memory usage test. Each test will be followed by a brief discussion of the results.

4.2.1 Factorial 500 Tests

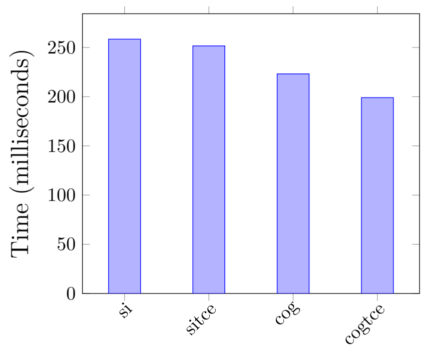
The Factorial 500 tests use the calculation of the factorial of 500 as the core calculation of the test. The tail recursive implementation of factorial is included here again. The test runner for execution time and for memory usage for factorial is in Section 3.6.1.

```
ThesisTestSuite class>>factorialMem: n accumulator: i
   (n = 0) ifTrue:[Memory := self getMemory. ↑i]
   ifFalse:[↑self factorialMem: (n-1) accumulator: (i * n)].

factorialMem: n
   ↑self factorialMem:n accumulator: 1
```

Execution Time

The execution time test measures the milliseconds to run 1000 iterations of factorial 500. In total, 100 runs are performed. See Figure 4.1 for the results.



Version	Mean	%Imp	%SD	Std Dev	Median
si	258.43			1.41	258.00
sitce	251.61	2.6	0.7	1.02	251.00
cog	223.18			15.46	217.00
cogtce	199.01	10.8	6.9	1.04	199.00

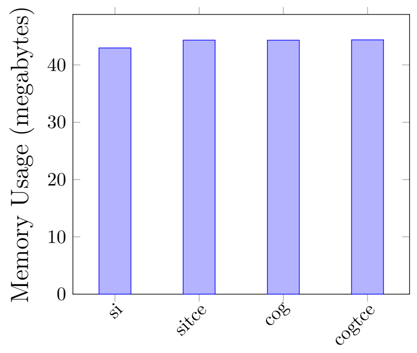
Figure 4.1: Factorial 500 - Execution Time

Execution time tests show a slight difference between implementations with tail call elimination, and implementations without tail call elimination. Mean execution time for both tail call eliminating versions indicate a slight reduction, with a slightly higher reduction being indicated in the Cog implementation.

Memory Usage

The next results to show are the memory usage tests for Factorial 500, in which a measurement of memory usage is taken at the base case of each calculation of factorial. Each test consists of one iteration of factorial 500, and the test is run 1000 times. See Figure 4.2 for the results.

Memory usage across all implementations for Factorial 500 is very similar across the different runs, registering slight increases for both the tail call eliminating implementations. However, both increases fall within one standard deviation.



Version	Mean	%Imp	%SD	Std Dev	Median
si	42.96			1.70	43.03
sitce	44.32	-3.2	5.6	1.70	44.31
cog	44.31			1.70	44.42
cogtce	44.37	-0.2	5.4	1.70	44.37

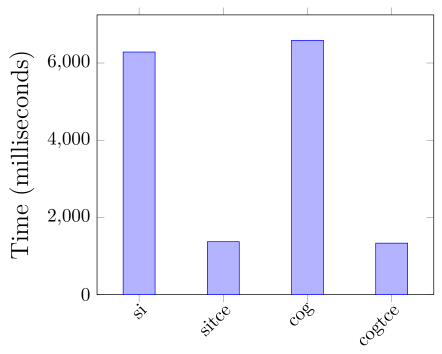
Figure 4.2: Factorial 500 - Memory Usage

4.2.2 Factorial 5000 Tests

The Factorial 5000 tests use the calculation of the factorial of 5000 as the core calculation of the test. The tail recursive implementation of factorial can be referenced above in Figure 4.2.1. The test runner for execution time and for memory usage for factorial is in Section 3.6.1.

Execution Time

The execution time test measures the milliseconds to run 100 iterations of factorial 5000. In total, 25 runs are performed. See Figure 4.3 for the results.



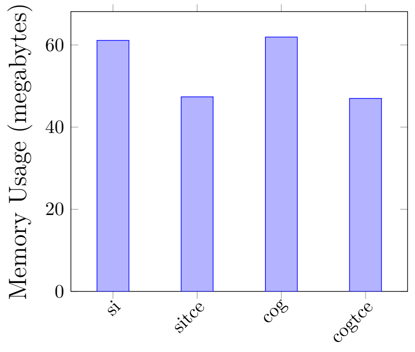
Version	Mean	%Imp	%SD	Std Dev	Median
si	6284.60			12.52	6283.00
sitce	1372.40	78.2	0.4	21.88	1356.00
cog	6587.72			16.45	6591.00
cogtce	1333.28	79.8	0.5	27.36	1314.00

Figure 4.3: Factorial 5000 - Execution Time

Execution time shows a significant difference between implementations with tail call elimination, and implementations without tail call elimination. For the stack interpreter, mean execution time is reduced by 78.2 ±0.4% of the original implementation, and, for the JIT compiler, execution time is reduced to 79.8±0.5% of the original implementation.

Memory Usage

The next results to show are the memory usage tests for Factorial 5000. Again, a measurement of memory usage is taken at the base case of each run of factorial. Each test consists of one iteration of factorial 5000 and the test is run 250 times. See Figure 4.4 for the results.



Version	Mean	%Imp	%SD	Std Dev	Median
si	61.11			3.28	60.33
sitce	47.37	22.5	7.7	3.35	47.22
cog	61.92			6.17	59.50
cogtce	46.99	24.1	11.3	3.33	46.89

Figure 4.4: Factorial 5000 - Memory Usage

Memory usage for Factorial 5000 also shows a significant difference between implementations with tail call elimination and implementations without tail call elimination.

4.2.3 Fibonacci 1000 Tests

The Fibonacci 1000 tests use the calculation of the Fibonacci sequence for 1000. Included again is the tail recursive implementation of the calculation of a Fibonacci sequence number. The actual test runner for this can be seen in 3.6.2.

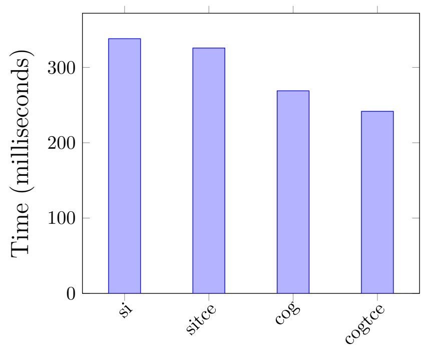
```
ThesisTestSuite class>>fibonacci: n
  ↑self fibonacci: n a: 0 b: 1.

ThesisTestSuite class>>fibonacci: n a: a b: b
  (n = 0) ifTrue:[↑a].
  (n = 1) ifTrue:[↑b].
  ↑self fibonacci: n - 1 a: b b: a+b
```

Execution Time

For execution time, each test consists of 1000 iterations of calculating the Fibonacci sequence for 1000. The milliseconds to run each test was collected for 100 runs. See

Figure 4.5 for the results.



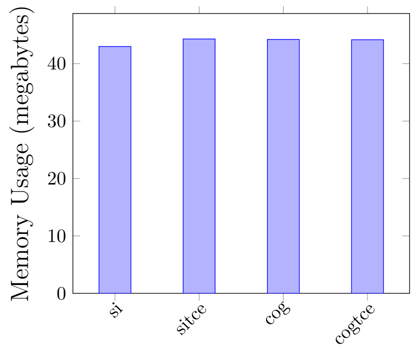
Version	Mean	%Imp	%SD	Std Dev	Median
si	338.12			1.54	338.00
sitce	325.73	3.7	1.2	3.68	324.00
cog	268.94			1.99	268.00
cogtce	241.72	10.1	0.9	1.33	242.00

Figure 4.5: Fibonacci 1000 - Execution Time

The results of the Fibonacci 1000 tests follow a similar pattern as the results for Factorial 500. Possible improvements are shown in mean execution time between the tail call and non-tail call implementations of each VM, with a higher possible improvement indicated for the Cog VM.

Memory Usage

Next are the memory usage results for the Fibonacci 1000 test. For this test, memory usage was measured in the base case when calculating the Fibonacci sequence for 1000. Measurements were taken for 1000 runs - see Figure 4.6 for the results.



Version	Mean	%Imp	%SD	Std Dev	Median
si	42.99			1.70	42.98
sitce	44.30	-3.0	5.6	1.70	44.31
cog	44.22			1.70	44.24
cogtce	44.17	0.1	5.4	1.70	44.18

Figure 4.6: Fibonacci 1000 - Memory Usage

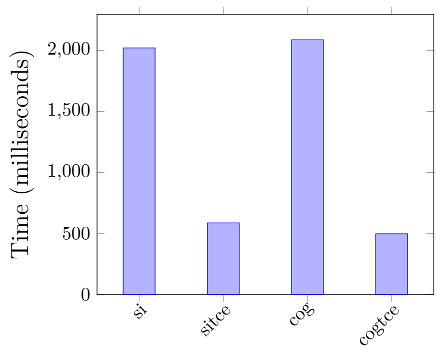
Memory usage follows a very similar pattern to Factorial 500, with a slight increase in usage for the tail call eliminating implementation of the Stack Interpreter.

4.2.4 Fibonacci 10000 Tests

The Fibonacci 10000 tests use the calculation of the Fibonacci sequence for 10000 - the tail recursive implementation of this can be viewed above in Section 4.2.3. The actual test runner for this can be seen in Section 3.6.2.

Execution Time

For execution time, each test consists of 100 iterations of calculating the Fibonacci sequence for 10000. The milliseconds to run each test was collected for 25 runs. See Figure 4.7 for the results.



Version	Mean	%Imp	%SD	Std Dev	Median
si	2019.44			24.74	2006.00
sitce	586.00	71.0	1.3	8.77	583.00
cog	2086.08			38.08	2067.00
cogtce	496.60	76.2	1.9	10.21	494.00

Figure 4.7: Fibonacci 10000 - Execution Time

Execution time shows a significant difference, with large reductions in execution time indicated for both implementations of tail call elimination. These results echo the results of Factorial 5000, where larger numbers result in significant reduction in execution time.

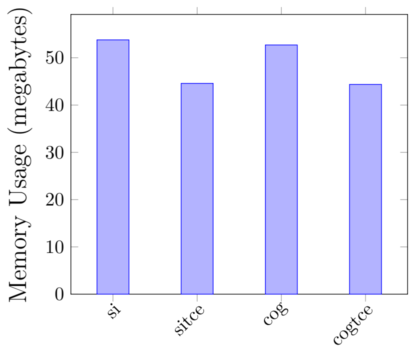
Memory Usage

Next are the memory usage results for the Fibonacci 10000 test. For this test, memory usage was again measured in the base case when calculating the Fibonacci sequence for 10000. Measurements were taken for 250 runs - see Figure 4.8 for the results.

Like the execution time results, the memory usage results follow a similar pattern as the results of Factorial 5000, with a significant decrease in usage for both tail call eliminating implementations.

4.2.5 WhileTrue 1000 Tests

The WhileTrue 1000 test uses a tail recursive implementation of a typical Smalltalk loop structure, `whileTrue:`, to repeatedly execute a block 1000 times. Included again is the



Version	Mean	%Imp	%SD	Std Dev	Median
si	53.79			4.94	54.73
sitce	44.59	17.1	9.7	1.70	44.58
cog	52.71			5.05	53.78
cogtce	44.37	15.8	10.1	1.71	44.42

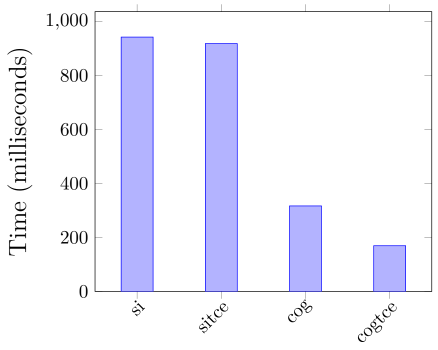
Figure 4.8: Fibonacci 10000 - Memory Usage

tail recursive implementation of this method. The test runner for the execution tests can be reviewed in Section 3.6.3.

```
BlockClosure>>whileTrueTail: aBlock
  ↑ self value ifTrue: [
     aBlock value.
     ↑ self whileTrueTail: aBlock
  ]
```

Execution Time

Execution time for this test is measured by running the tail recursive whiletrue loop with a value of 1000, and measuring execution time for 10000 iterations. This test is repeated for 100 runs and results are presented in Figure 4.9.



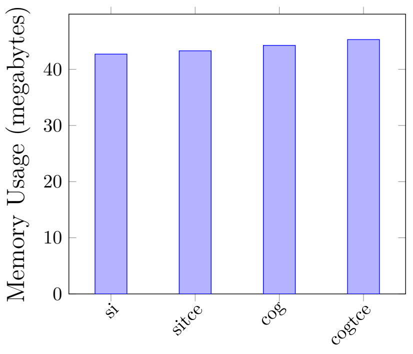
Version	Mean	%Imp	%SD	Std Dev	Median
si	943.06			1.75	943.00
sitce	918.93	2.6	0.4	2.99	918.00
cog	316.94			2.78	317.00
cogtce	169.37	46.6	0.9	0.60	169.00

Figure 4.9: WhileTrue 1000 - Execution Time

The execution results for this test are interesting - in general, this test runs significantly better in the Cog VM than in the Stack Interpreter, and in particular the tail call eliminating version of the Cog VM performs well. The Stack Interpreter implementation indicates a slight possible reduction in mean execution time, whereas the Cog VM implementation shows a large reduction.

Memory Usage

Next are the memory usage results for the Whiletrue 1000 test. For this test, memory usage was measured after the completion of one iteration of whiletrue with a value of 1000. Measurements were taken for 1000 runs - see Figure 4.6 for the results.



Version	Mean	%Imp	%SD	Std Dev	Median
si	42.71			1.58	42.64
sitce	43.30	-1.4	3.9	0.58	43.30
cog	44.26			1.70	44.34
cogtce	45.30	-2.3	4.0	0.58	45.30

Figure 4.10: WhileTrue 1000 - Memory Usage

Memory usage results shows slight increases for the tail call eliminating versions of the implementations.

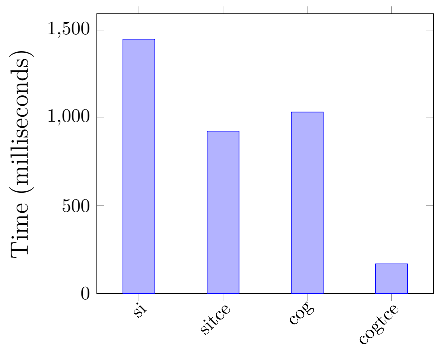
4.2.6 WhileTrue 10000 Tests

The WhileTrue 10000 test again uses the tail recursive implementation of `whiteTrue:` to repeatedly execute a block 10000 times. The code and execution time test runner can be reviewed above in Section 4.2.5 The memory test runner can be reviewed in Section 3.6.3.

Execution Time

In the final tail recursive test, execution time for this test is measured by running the whiletrue loop with a value of 10000, and measuring execution time for 1000 iterations. This test is repeated for 25 runs and results are presented in Figure 4.11.

The execution results for this test is also interesting - both implementations show a large reduction in execution time when using tail call elimination, particularly for the Cog implementation.

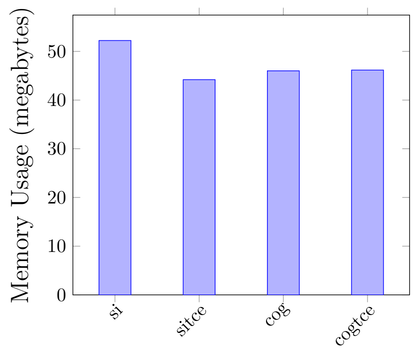


Version	Mean	%Imp	%SD	Std Dev	Median
si	1448.28			45.25	1454.00
sitce	924.60	36.2	3.1	2.40	925.00
cog	1033.20			25.98	1050.00
cogtce	168.36	83.7	2.5	0.49	168.00

Figure 4.11: WhileTrue 10000 - Execution Time

Memory Usage

In the final memory usage test for tail recursion, memory usage was measured after the completion of one iteration of whiletrue with a value of 10000. Measurements were taken for 250 runs - see Figure 4.8 for the results.



Version	Mean	%Imp	%SD	Std Dev	Median
si	52.23			1.84	52.29
sitce	44.19	15.4	3.5	0.15	44.19
cog	46.01			1.79	46.16
cogtce	46.16	-0.3	3.9	0.15	46.16

Figure 4.12: WhileTrue 10000 - Memory Usage

Memory usage results show a decrease in the memory usage for the Stack Interpreter, and a slight increase in memory usage for the Cog VM.

4.3 Real World Performance Test

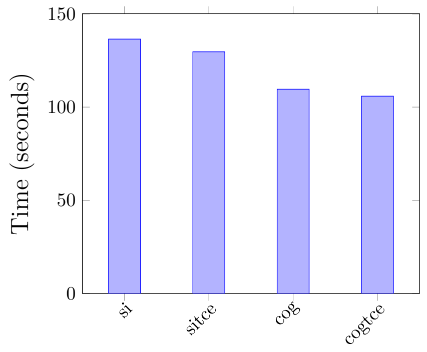
4.3.1 Compile All Test

The compile all test calls the `recompileAll` method on the `Compiler` object and recompiles every method in the Smalltalk image. As a reminder, the code for this test is included below - the test runners and memory test runner can be reviewed in Section 3.7.1.

```
Compiler>>recompileAll
   Smalltalk allClassesAndTraits
      do: [:classOrTrait | classOrTrait compileAll]
      displayingProgress:[:classOrTrait| 'Recompiling␣', classOrTrait]
```

Execution Time

The next set of tests are the compiler tests using the `recompileAll:` method. As mentioned before, the method will be run 10 times and the time in milliseconds is collected for each run. The results are displayed in Figure 4.13.



Version	Mean	%Imp	%SD	Std Dev	Median
si	136.49			2.30	135.52
sitce	129.68	5.0	1.7	0.33	129.60
cog	109.59			0.90	109.29
cogtce	105.86	3.4	0.9	0.45	105.84

Figure 4.13: Compile-All - Execution Time

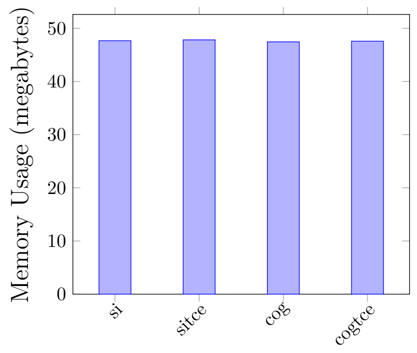
Try: Try: 3.4

Both implementations show an improvement in overall execution time when using tail call elimination. The Stack Interpreter implementation's execution time shows a 5.0±1.7% improvement. The Cog VM implementation shows less of an improvement - a 3.4±0.9% improvement to mean execution time. While this improvement is small, the results are tightly clustered enough that this improvement is still several standard deviations away from the non tail call eliminating version of Cog.

Memory Usage

The next results to show are the memory usage tests for the Compiler tests. Again, a measurement of memory usage is taken after compiling each class. See Figure 4.14 for the results.

In general, the results seen are not particularly different from one another, or even particularly different between the Stack Interpreter and the Cog Interpreter, with minor increases in memory usage being registered that fall well below one standard deviation.



Version	Mean	%Imp	%SD	Std Dev	Median
si	47.67			4.41	47.67
sitce	47.83	-0.3	13.1	4.45	47.85
cog	47.46			4.39	47.41
cogtce	47.58	-0.3	13.0	4.36	47.57

Figure 4.14: Compile-All - Memory Usage

4.3.2 Windows Test

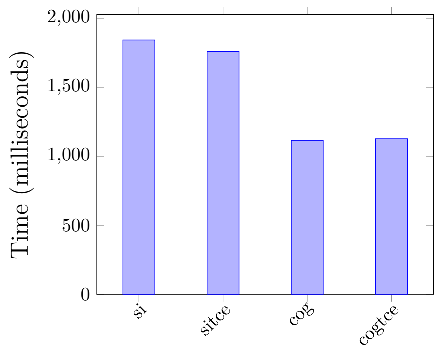
The windows test calls the `browse` method on each subclass of Number, and then closes the browse windows. As a reminder, the code for this test is included below - the test runners and memory test runner can be reviewed in Section 3.7.2.

```
ThesisTestSuite>>windowTestWithClass: class
   class allSubclassesDo:[ :a |
      a browse.
      self currentWorld doOneCycle
      ].
   Browser allInstancesDo:[:a |
      (a dependents select:[:
         each | each isMemberOf: PluggableSystemWindow ]
      ) do:[: each|
         each delete.
         self currentWorld doOneCycle
         ].
      ].
```

Execution Time

The next set of tests are the execution tests for the windows test. As mentioned before, the method will be run 10 times and the time in milliseconds is collected for each run. The results are displayed in Figure 4.15.

Note that, while the results for the Stack Interpreter indicate a possible small improvement, the Cog implementation sees a possible performance decrease. Tuning the memory available for JIT compiled code produces different results - the results presented



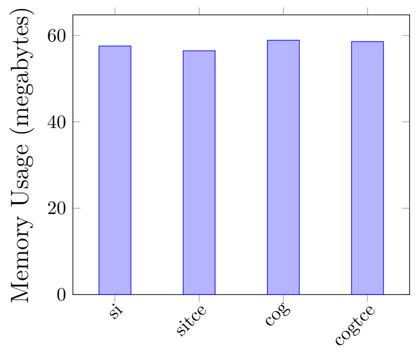
Version	Mean	%Imp	%SD	Std Dev	Median
si	1842.60			51.99	1850.00
sitce	1759.90	4.5	4.2	56.60	1754.00
cog	1115.40			62.26	1124.00
cogtce	1127.20	-1.1	7.5	55.21	1124.00

Figure 4.15: Windows - Execution Time (8 mb JIT space for Cog)

here for both the tail call eliminating version of Cog and the non tail call eliminating version of Cog both have 8 megabytes made available for JIT compiled code. By default (and in all other tests), only one megabyte is made available. With one megabyte, performance suffers even more.

Memory Usage

The next results to show are the memory usage tests for the Windows tests. A measurement of memory usage is taken after opening all windows, and again after closing all windows. See Figure 4.16 for the results.



Version	Mean	%Imp	%SD	Std Dev	Median
si	57.56			1.83	57.69
sitce	56.46	1.9	6.1	2.98	56.17
cog	58.89			2.70	59.42
cogtce	58.58	0.5	6.2	2.49	58.54

Figure 4.16: Windows - Memory Usage

The results here are not particularly interesting, with tests indicating possible decreases in memory usage for both tail call eliminating versions.

4.4 Analysis

4.4.1 Tail Call Recursion Analysis

As a general observation, the results obtained during these tests were tightly clustered. As a result, analysis will only consider the difference in the mean measurements between relevant implementations. Looking at the results of the tail recursive tests, there is a consistent pattern across both the Factorial and Fibonacci tests. For Factorial 500 and Fibonacci 1000, the execution time results indicate a possible slight improvement for the Stack Interpreter with tail call elimination with mean reductions of 2.6±0.7% and 3.7±1.2% respectively. For Factorial 5000 and Fibonacci 10000, the execution time results indicate a larger improvement for both the Stack Interpreter with tail call elimination with mean reductions of 78.2±0.4% and 71.0±1.3%. A similar pattern is exhibited for the Cog VM implementation. For Factorial 500 and Fibonacci 1000, the execution time results show possible mean reductions of 10.8±6.9% and 10.1±0.9%. For Factorial 5000 and Fibonacci 10000, there is a larger improvement for Cog with mean reductions of 79.8±0.5% and 76.2±1.9%.

Memory usage results for Factorial and Fibonacci tests also show a consistent pattern. For Factorial 500 and Fibonacci 1000, memory usage test results indicate a possible slight increase for the Stack Interpreter with tail call elimination with decreases of -3.2±5.6% and -3.0±5.6% respectively. In Cog, these tests don't show a significant difference either way. Memory usage shows a mean decrease for Factorial 5000 and Fibonacci 10000 22.5±7.7% and 17.1±9.7% in the Stack Interpreter, and 24.1±11.3% and 15.8±10.1% in Cog.

Results for the Whiletrue 1000 and 10000 tests are roughly consistent with the results of the other tail recursive tests, with much more significant differences between the Stack Interpreter and the Cog VM on display. For Whiletrue 1000, the Stack Interpreter showed a small possible decrease with 2.6±0.4%, and the Cog VM showed a significant decrease of 46.6±0.9%. For Whiletrue 10000, the Stack Interpreter showed a possible decrease of 36.2±3.1%, and the Cog VM showed a very significant decrease of 83.7±2.5%.

Memory usage showed a pattern of slight increases in memory usage for Whiletrue 1000, which echoes the results of Factorial 500 and Fibonacci 1000. The Stack Interpreter showed a small decrease with -1.4±3.9%, and the Cog VM showed a decrease of -2.3±4.0%. For Whiletrue 10000, the Stack Interpreter saw a memory usage decrease of 15.4±3.5%. Oddly, for the Cog VM, despite the significant decrease in execution time, memory usage was not significantly different, registering a slight increase.

The consistent results of the Factorial and Fibonacci tests suggest that tail call elimination improves execution time, and that this improvement becomes more pronounced as the number of recursive calls increase, with significant decreases in execution time being seen for both Factorial 5000 and Fibonacci 10000. Memory usage is also significantly improved when calculating large numbers, though there appears to be a small cost in terms of memory usage when calculating smaller numbers. The results for the whileTrue test generally exhibit a similar pattern, with much more pronounced difference between the Stack Interpreter and the Cog VM. In particular, there is little variation between the tail call and non tail call implementation of the Stack Interpreter for Whiletrue 1000, whereas the Cog version of the same test already shows a significant improvement. Even the Whiletrue 10000 test exhibits disparate results, though both implementations see significant improvement. Speculatively, it could be that calculating large numbers in the Fibonacci and Factorial tests perform similarly between the Cog VM and the Stack Interpreter, while simple comparisons of smaller numbers may perform signficantly better in JIT compiled code. Memory results also indicate a discrepancy between the two implementations, with only the stack interpreter showing memory usage improvements in the Whiletrue 10000 test. Memory usage for the Whiletrue 1000 test seem to indicate again that there may be a small cost in memory usage to these implementations of tail call elimination.

The results suggest that the value of tail call elimination in tail recursion may be less about reducing the number of instructions required to execute a call and return, and more about reducing the number of stack frames and stack pages allocated, which could lessen the amount of background work that the VM and the garbage collector need to do. The slight memory usage increase seen in the tail call eliminating versions are difficult to explain for the Stack Interpreter - this could possibly be due to some overhead when checking each send for tail call potential. For the Cog VM this could be explained by the fact that JIT compiled methods have a slightly larger memory footprint.

4.4.2 Real World Analysis

The results of the compiler test show a modest benefit to execution speed using tail call elimination in a real world scenario. The Stack Interpreter implementation of tail call elimination showed better results with a decrease of 5.0±1.7%, despite the overhead of checking each send for the next bytecode to determine whether it is a tail call or not. The Cog VM implementation of tail call elimination showed a more modest improvement of 3.4±0.9%, despite not having the same overhead. The execution time may be affected by the limits of this implementation. As only monomorphic sends are implemented using tail call elimination, the number of tail calls actually successfully executed are lower in the Cog implementation compared to the Stack Interpreter implementation. In Section 4.1.2 the number of dynamic tail calls was given. Testing indicates that, of those tail calls, about 1% fail the monomorphic send cache and are reverted to normal calls to polymorphic caches. However, statistics were not collected for how many calls are then made using those polymorphic caches, however we speculate that about 15% of tail calls use those caches. The results of testing memory usage confirms that, despite the use of tail call elimination, memory usage in this scenario stayed the same. Reflecting on the results of the tail recursive tests, this further confirms the speculation that improvements in memory usage are tied to deep call stacks that require the allocation of many frames - perhaps the methods used by the compiler simply don't create the stack depth necessary for the benefit of tail call elimination to be seen. For the windows tests, results indicate that, while UI interaction may be improved with tail call elimination, the implementation presented for the Cog VM in this work may not be ideal, as significant tuning to JIT code space memory was required just to produce results that indicate a small decrease in performance.

4.4.3 Final Analysis

In general, the performance results across tests vary significantly depending on the type of tests and the number of iterations. In addition, some tests showed a significant difference in performance between the Stack Interpreter implementation and the Cog implementation. Performance appears to be affected by three different factors: stack size, instruction count and code space for JIT compiled code. Regarding stack size, the results indicate that, across implementations, performance in terms of execution speed and memory usage appear to be dramatically affected by the number of iterations in tail recursive cases. The drastic improvements seen with tail call elimination in these cases are likely due to the reduction in size of the call stacks needed to execute these tests. It's clear as to why memory usage would be reduced in this case, as tail recursion elimination prevents stack growth. Execution time is likely reduced as the overhead for the virtual machine to manage stack overflow, creating new frames, and obtaining memory for these frames is eliminated. With the dramatic savings in memory usage seen with these results, there is also less overhead from the operating system, as there isn't the same pressure for memory pages. Regarding instruction count, there could potentially be a difference in the number of instructions executed between the Stack Interpreter implementation and the Cog implementation. While the number of instructions executed in Cog is known, as these instructions are generated by the JIT compiler, the number of instructions executed by the Stack Interpreter is an unknown. Speculatively, this could be a source of some of the differences in performance between the Stack Interpreter and the Cog implementations. Finally, regarding code space for JIT compiled code, this could also be a source of some performance difference between the two implementations. At least one test indicated that JIT compiled code space management could be a source of significant overhead for the Cog implementation, causing a significant difference between implementations. This overhead could also be a source of some of the minor differences between implementations as well.

Chapter 5

Conclusions

My thesis is that tail call elimination is a useful optimization for an object-oriented language such as Smalltalk. This has been shown through the design of two implementations of tail call elimination in the Smalltalk VM, targeting two variants of the VM. The results for both implementations have shown that, generally speaking, there are significant improvements in execution time for computationally expensive real world processes - these improvements show that tail call elimination is indeed a useful optimization. In addition, tail call elimination has been shown to offer huge improvements in both execution time and memory usage for tail recursive functions with deep call stacks. While this may be of limited use in the current code base, having tail call elimination offers Smalltalk programmers the ability to program effectively using this style. The results presented here may also have general application to other environments and programming languages such as the JVM and Python. The challenges faced in this work are likely to be applicable, as JVM and Python both compile to platform-independent bytecode, support object oriented programming with polymorphism, and have implementations which use JIT compilation.

5.1 Contributions

- Designing and implementing an approach for tail call elimination in the Stack Interpreter;
- Providing experimental evidence that there are significant benefits to this implementation;
- Designing and implementing an approach for monomorphic tail call elimination in

the Cog VM;

- Providing experimental evidence that there are significant benefits even with this limited implementation.

5.2 Future Work

The results presented are promising enough that future work in implementing tail call elimination in the Opensmalltalk Virtual Machine is worth pursuing. This paper introduced two possible methods for statically identifying tail calls in the Stack Interpreter - one by flagging the method with a special primitive, and the other by switching implementation to a different bytecode set. However, the introduction of the Sista bytecode set presents an opportunity for the introduction of at least one tail call bytecode, which would allow for the recognition of tail calls in the interpreter at compile time. This would eliminate the need to find creative ways to flag a method as having a tail call at compile time. Recall from Chapter 2 that a Sista tail call bytecode can take extensions - it could encode the number of arguments and the literal selector (or a special selector index). This should improve the presented implementation of tail call elimination in the Stack Interpreter. In the Cog VM, work could be done to support polymorphic and megamorphic send caches, allowing tail call elimination to be done on more than just monomorphic sends. In addition, work could be done to reduce the redundant class lookup used in the JIT code presented in this implementation. A tail call runtime method could be implemented which does the necessary class lookups while still preserving the instruction pointer in case of a call. Introducing a new parameter to the virtual machine to control whether tail call elimination is being performed would be useful, as it could be turned off to maintain proper stack traces. While maintaining full stack traces during a debug session was a side effect of this implementation, formalizing this with a parameter would be ideal, as the implementation may change.

Bibliography

[1] Andrew W. Appel. *Compiling with Continuations.* New York, NY, USA: Cambridge University Press, 1992. ISBN: 0-521-41695-7.

[2] Henry G. Baker. "CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A." In: *SIGPLAN Not.* 30.9 (1995-09), pp. 17–20. ISSN: 0362-1340. DOI: 10.1145/214448.214454. URL: http://doi.acm.org/10.1145/214448.214454.

[3] Nick Benton, Andrew Kennedy, and George Russell. "Compiling Standard ML to Java Bytecodes". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming.* ICFP '98. Baltimore, Maryland, USA: ACM, 1998, pp. 129–140. ISBN: 1-58113-024-4. DOI: 10.1145/289423.289435. URL: http://doi.acm.org/10.1145/289423.289435.

[4] Clément Béra. "A Low Overhead Per Object Write Barrier for the Cog VM". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies.* IWST'16. Prague, Czech Republic: ACM, 2016, 22:1–22:10. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991063. URL: http://doi.acm.org/10.1145/2991041.2991063.

[5] Clément Béra. *FullBlockClosure design.* 2016-06. URL: https://clementbera.wordpress.com/2016/06/27/fullblockclosure-design/.

[6] Clément Béra. *Spur's new object format.* 2014-01. URL: https://clementbera.wordpress.com/2014/01/16/spurs-new-object-format/.

[7] Clément Béra and Eliot Miranda. "A bytecode set for adaptive optimizations". In: *International Workshop on Smalltalk Technologies.* Cambridge, United Kingdom, 2014. URL: https://hal.inria.fr/hal-01088801.

[8] Per Bothner. "Kawa: Compiling Dynamic Languages to the Java VM". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference.* ATEC '98. New Orleans, Louisiana: USENIX Association, 1998, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=1268256.1268297.

[9] John Clements and Matthias Felleisen. "A Tail-recursive Machine with Stack Inspection". In: *ACM Trans. Program. Lang. Syst.* 26.6 (2004-11), pp. 1029–1052. ISSN: 0164-0925. DOI: 10.1145/1034774.1034778. URL: http://doi.acm.org/10.1145/1034774.1034778.

[10] William D. Clinger. "Proper Tail Recursion and Space Efficiency". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation.* PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pp. 174–185. ISBN: 0-89791-987-4. DOI: 10.1145/277650.277719. URL: http://doi.acm.org/10.1145/277650.277719.

[11] ClojureTV. *Brian Goetz - Stewardship: the Sobering Parts.* 2014-11. URL: https://www.youtube.com/watch?v=2y5Pv4yN0b0.

[12] L. Peter Deutsch and Allan M. Schiffman. "Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL '84. Salt Lake City, Utah, USA: ACM, 1984, pp. 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542. URL: http://doi.acm.org/10.1145/800017.800542.

[13] *ECMAScript 2015 Language Specification.* URL: http://www.ecma-international.org/ecma-262/6.0/ (visited on 2019-04-13).

[14] *ECMAScript 6 compatibility table.* URL: http://kangax.github.io/compat-table/es6/ (visited on 2019-04-13).

[15] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[16] Chris Hanson. "Efficient Stack Allocation for Tail-recursive Languages". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming.* LFP '90. Nice, France: ACM, 1990, pp. 106–118. ISBN: 0-89791-368-X. DOI: 10.1145/91556.91603. URL: http://doi.acm.org/10.1145/91556.91603.

[17] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* OOPSLA '97. Atlanta, Georgia, USA: ACM, 1997, pp. 318–326. ISBN: 0-89791-908-4. DOI: 10.1145/263698.263754. URL: http://doi.acm.org/10.1145/263698.263754.

[18] *JIT Compiler Structure.* URL: https://github.com/dotnet/coreclr/blob/master/Documentation/botr/ryujit-overview.md (visited on 2019-04-13).

[19] Richard Kelsey. *Tail-Recursive Stack Disciplines for an Interpreter*. Tech. rep. 1993.

[20] *Kotlin*. URL: `https://kotlinlang.org/docs/reference/` (visited on 2019-04-13).

[21] *LLVM Language Reference Manual*. URL: `https://llvm.org/docs/LangRef.html` (visited on 2019-04-13).

[22] Eliot Miranda. *A Spur gear for Cog*. 2013-09. URL: `http://www.mirandabanda.org/cogblog/2013/09/05/a-spur-gear-for-cog/`.

[23] Eliot Miranda. *Build Me a JIT as fast as you can...* 2011-03. URL: `http://www.mirandabanda.org/cogblog/2011/03/01/build-me-a-jit-as-fast-as-you-can/`.

[24] Eliot Miranda. *Closures Part I*. 2008-06. URL: `http://www.mirandabanda.org/cogblog/2008/06/07/closures-part-i/`.

[25] Eliot Miranda. *Closures Part II – the Bytecodes*. 2008-07. URL: `http://www.mirandabanda.org/cogblog/2008/07/22/closures-part-ii-the-bytecodes/`.

[26] Eliot Miranda. *Simulate Out Of The Bochs*. 2008-03. URL: `http://www.mirandabanda.org/cogblog/2008/12/12/simulate-out-of-the-bochs/`.

[27] Eliot Miranda. "The Cog Smalltalk Virtual Machine writing a JIT in a high-level dynamic language". In: *VMIL '11*. 2011.

[28] Eliot Miranda. *Under Cover Contexts and the Big Frame Up*. 2009-01. URL: `http://www.mirandabanda.org/cogblog/2009/01/14/under-cover-contexts-and-the-big-frame-up/`.

[29] Eliot Miranda and Clément Béra. "A Partial Read Barrier for Efficient Support of Live Object-oriented Programming". In: *Proceedings of the 2015 International Symposium on Memory Management*. ISMM '15. Portland, OR, USA: ACM, 2015, pp. 93–104. ISBN: 978-1-4503-3589-8. DOI: `10.1145/2754169.2754186`. URL: `http://doi.acm.org/10.1145/2754169.2754186`.

[30] *Opensmalltalk-VM*. URL: `https://github.com/OpenSmalltalk/opensmalltalk-vm` (visited on 2019-04-13).

[31] Simon Peyton Jones. "Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine." In: *J. Funct. Program.* 2 (1992-04), pp. 127–202. DOI: `10.1017/S0956796800000319`.

[32] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. "C–: A Portable Assembly Language That Supports Garbage Collection". In: *Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*. PPDP '99. London, UK, UK: Springer-Verlag, 1999, pp. 1–28. ISBN: 3-540-66540-4. URL: `http://dl.acm.org/citation.cfm?id=645815.668891`.

[33] *[Proposal] Support tail recursion #1235*. URL: `https://github.com/dotnet/roslyn/issues/1235` (visited on 2019-04-13).

[34] Guido van Rossum. *Tail Recursion Elimination*. 2009-04. URL: `http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html`.

[35] M. Schinz and M. Odersky. "Tail call elimination on the Java Virtual Machine". English. In: *Electronic Notes in Theoretical Computer Science* 59.1 (2001). Cited By :10, pp. 158–171. URL: `www.scopus.com`.

[36] *Standard ECMA-335 Common Language Infrastructure (CLI)*. URL: `https://www.ecma-international.org/publications/standards/Ecma-335.htm` (visited on 2019-04-13).

[37] Guy Lewis Steele Jr. "Debunking the "Expensive Procedure Call"; Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO". In: *Proceedings of the 1977 Annual Conference*. ACM '77. Seattle, Washington: ACM, 1977, pp. 153–162. ISBN: 978-1-4503-3921-6. DOI: `10.1145/800179.810196`. URL: `http://doi.acm.org/10.1145/800179.810196`.

[38] Gerald J. Sussman and Guy L. Steele Jr. *An Interpreter for Extended Lambda Calculus*. Tech. rep. Cambridge, MA, USA, 1975.

[39] *Tail calls in F#*. URL: `https://blogs.msdn.microsoft.com/fsharpteam/2011/07/08/tail-calls-in-f/` (visited on 2019-04-13).

[40] David Tarditi, Peter Lee, and Anurag Acharya. "No Assembly Required: Compiling Standard ML to C". In: *ACM Lett. Program. Lang. Syst.* 1.2 (1992-06), pp. 161–177. ISSN: 1057-4514. DOI: `10.1145/151333.151343`. URL: `http://doi.acm.org/10.1145/151333.151343`.

[41] T. Tauber, X. Bi, Z. Shi, W. Zhang, H. Li, Z. Zhang, and B. C. D. S. Oliveira. *Memory-Efficient tail calls in the JVM with imperative functional objects*. English. Vol. 9458. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Cited By :1. 2015, pp. 11–28. URL: `www.scopus.com`.

[42] *The Clojure Programming Language*. URL: `https://clojure.org/index` (visited on 2019-04-13).

[43] *The Revised6 Report on the Algorithmic Language Scheme*. URL: `http://www.r6rs.org/` (visited on 2019-04-13).

[44] *The Scala Programming Language*. URL: `https://www.scala-lang.org/` (visited on 2019-04-13).

Glossary

Activation Record A data structure storing information about the activation of a call or method. Often contains pointers to previous activation records.

Binary Message A message with an operator (such as $+$ or $*$) as a selector and one argument.

Block In Smalltalk, one or more lines of code surrounded by square brackets - blocks are not evaluated until a specific message is sent. Blocks are first class objects and are closures.

Call The process by which a function or procedure invokes another function or procedure, with the capability of returning. Analogous to send in Smalltalk.

Cog The just-in-time compiler of the OpenSmalltalk Virtual Machine.

Cogit The name of the JIT compiler class in OpenSmalltalk.

CoInterpreter The name of the OpenSmalltalk VM interpreter class which does contain a JIT compiler.

Continuation A representation of the current execution state of a program. Often represented as a lambda or as an object.

Continuation-Passing Style A programming style in which functions never return a result, instead passing a continuation to the next function.

Closure A function that has captured the free variables that it references.

Enilopmart A runtime routine for switching execution from JIT compiled assembly to the interpreter.

Image A collection of Smalltalk objects which make up an executable Smalltalk system, including source code, compiled code, and tools for interacting with the system. Runs on a Smalltalk VM.

Inline Cache A cache used by the JIT compiler to store addresses of methods for specific class/selector combinations. These are inline because either the cache itself, or the address of the cache, is written directly into a send site.

Jump The process by which execution of a program jumps from one location to another, without the capability of returning.

Just-In-Time (JIT) Compiler A compiler which compiles bytecode to platform specific assembly language as needed at run time.

Keyword Message A message with a selector made up of one or more keywords. Each keyword ends in a colon and takes an argument.

Literal The components of a method which are hard-coded values, such as strings, numbers, and names of selectors. A compiled method contains an array of references to these values.

Message A combination of a selector and zero or more arguments. Passed by objects to communicate with each other.

Megamorphic Inline Cache A method lookup routine for a specific selector - can support more classes than a polymorphic inline cache.

Monomorphic Inline Cache A send site specific cache of a class and a selector used in JIT compiled code.

Polymorphic Inline Cache A selector specific runtime routine that contains a limited set of class tags and addresses for methods matching the class tag and selector. Can be used by multiple send sites.

Primitive A method which is implemented as part of the virtual machine runtime instead of as a Smalltalk method.

Receiver In message passing, the object receiving a message.

Reduction A procedure call where the calling procedure will not continue execution after the callee is complete. Analogous to a tail call.

Selector The component of a message that specifies which method the sender is requesting the receiver to execute. Usually represented as a text description of the method or a mathematical operator.

Self A reserved word in Smalltalk which serves as a reference to the current object. Typically used for an object to send messages to itself.

Send The process by which objects pass messages in Smalltalk. Analogous to call in other languages.

Sender In message passing, the object sending a message.

Stack Interpreter The name of the OpenSmalltalk VM interpreter class which does not contain a JIT compiler.

Subproblem A procedure call where the calling procedure will continue execution after the callee is complete.

Tail call A call, or send, which is followed immediately by a return.

Tail recursion A pattern in which recursive calls are also tail calls.

Trampoline A loop which calls functions and iterates as long as a function is returned.

Trampoline (Cog VM) A runtime method which switches execution from the interpreter to JIT compiled code.

Unary Message A message with a selector and no arguments.

Workspace An interactive environment for executing Smalltalk code in Squeak.