Inlined Code Generation for Smalltalk

Daniel Franklin
Toronto Metropolitan University
Department of Computer Science
Toronto, ON, Canada
daniel.franklin@torontomu.ca

Dave Mason
Toronto Metropolitan University
Department of Computer Science
Toronto, ON, Canada
dmason@torontomu.ca

Abstract

In this experience report we present our early work at improving Smalltalk performance by inlining message sends during compilation. Smalltalk developers typically write small method bodies with one or two statements, this limits a compiler's ability to perform many optimizations, e.g. common sub-expression elimination. The overhead of message sends in a dynamically typed language introduces a high overhead cost when looking up the class of the message receiver. Inlining messages into a method body produces methods with fewer message sends and more statements allowing the compiler to optimize and generate efficient executable code and avoid lookup overhead. There are several challenges to inlining messages in Smalltalk that need to be resolved

We describe a generalized inlining approach that can be applied regardless of the execution target - bytecode, threaded-execution, or machine code - allowing interactive debugging with perfect verisimilitude to the JITed code. In this experience report, we describe the inlining approach taken for the Zag Smalltalk compiler that solves for these issues and improves performance.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Dynamic compilers**; *Just-in-time compilers.*

Keywords: compile,inlining,Smalltalk,method dispatch

1 Introduction and Motivation

Zag Smalltalk [5] is a new compiler and runtime, for the Smalltalk programming language. Zag only maintains in-memory versions of methods as Abstract Syntax Trees (ASTs), and compiles them to executable form on demand. The ASTs used to compile methods is a simplified versions of the Pharo AST, consisting of sends, references, assignments, literals, arrays, `self`, `super`, and returns.

The Zag Smalltalk compiler will inline message sends where possible as a compile time optimization and will aggressively include as many message sends as possible recursively. We are aware of only one other attempt at pervasive message inlining for a dynamic programming language which was done for SELF[9]. Currently, Smalltalk implementations like Pharo only inline a known list of special selectors. Zag Smalltalk will apply inlining to more situations in our efforts to improve overall performance.

Inspecting the Pharo v12 image we find that 35% of selectors have a single implementor and another 8% have 2 or 3. Further, 28% of sends are to `self` or `super` and 3% are to literals. There is some overlap between these measures, but between 43% and 74% of sends are to known methods, which makes inlining them straightforward (if the true class of `self` is known - as it usually is in Zag). Methods are also very small (64% of methods have only 1 statement - 18% have no sends and return a literal, parameter, or variable, and another 19.5% have only 1 send) so code explosion should be limited when inlining. When the compiler sees a message send we can find the implementors of the message's selector within the current image. If there are few implementors of the selector then it is a candidate to be inlined at compile time. Based on these findings, inlining should not be a conceptually complex task and will result in more opportunities for a compiler to optimize and to avoid the overhead of the message send.

However, since Smalltalk has no control structures (conditionals or loops) and uses message sends, block closures and non-local returns to derive them, inlining becomes simultaneously more difficult and more critical.

2 Background

Inlining is not a new compiler optimization but it is more challenging for dynamic languages like Smalltalk where the receiver class of a message send may be completely unknown. Statically typed Object-Oriented (OO) languages like C++ and Java normally have more concrete type information, but even here (in the face of multiple inheritance and interfaces) there may be very little information about the possible targets. In such languages a compiler will generally not know which selectors' implementations to inline, only at runtime when the statement executes can a receivers type be known. Dynamic programming languages provide incredible flexibility to the programmer. However, this flexibility is at the cost of message sends that require significantly more overhead than static languages when looking up the method to be invoked and setting up the stack for the call. Again here, static languages have similar problems when implementing multiple inheritance and interfaces.

The dynamic programming language SELF[9] uses runtime types and a type inference mechanism to find possible types for a receiver of a message send. The compiler uses the derived types to compile a method multiple times, one for each of the different types derived for the receiver. At runtime, knowing the type of the receiver, SELF can then call the appropriate type-constrained compiled method.

Most Smalltalk compilers inline a fixed set of messages, like `ifTrue:ifFalse:`, `to:do:` and `whileTrue:` and their variants. These special selectors are hot spots in the code with known implementors that can be inlined safely and gain significant performance improvements. While this is fairy effective, there are numerous opportunities for inlining that are missed by this heuristic. For example, there are 43 enumerating methods on collections, and often only a few are inlined.

We will use the `fibonacci` code in listing 1 for examples in this report.

```
fibonacci
    self <= 2
        ifTrue: [ ^ 1 ].
    ^ (self-1) fibonacci + (self-2) fibonacci
```

Listing 1. Smalltalk Source for fibonacci

2.1 Stack

The Zag runtime makes use of a separate per-process stack instead of the hardware stack to manage access to the variables of a program. The stack is where the method arguments, locals and temporaries are stored along with a reference to `self` which facilitates message sends and assignments. To prepare for the execution of a message send the receiver and arguments are found and pushed onto the top of the stack. A dynamic dispatch is then performed which consumes the newly added elements on the stack and replaces them with a return value on the stack.

2.2 Threaded Execution

One of the features of Zag Smalltalk is the parallel between threaded execution and JITed machine code execution[7]. This allows fine-grained switching between threaded-execution - which can be debugged - and JITed code - which mostly cannot. By doing inlining in the target-agnostic way described below, the code that is being debugged exactly parallels the JITed production code.

3 Related Work

3.1 Self

The implementation of inlining in Zag Smalltalk closely follows the behaviour found in SELF [9, 2], which we revisit here. SELF is a purely object based language without class definitions, although maps mimic class behaviour. In SELF an object has named slots that may contain parent references, data values, or statements in for form of a method or block. Programmers will then send an object a message who's name matches one of the named slots.

Inlining rules:

1. If the slot contains a method definition the compiler will inline it if the method is small, containing few statements, and non-recursive.
2. If the slot contains a block value, like a method definition it will be inlined if it is small.
3. If the slot is a read-only data value the compiler will replace the send with the value of the slot since it is known at compile time.
4. SELF allows any slot to be read only or read-write, and if a data value can be modified messages sends to it are replaced with code to load the value from the object's slot.
5. If a message send writes to a slot the compiler will inline the send with code to update the contents of the object's slot.

Another strategy used to improve performance by avoiding method look up is Polymorphic Inlining Caching(PIC). PICs are used at call sites to remember the last lookup for a particular receiver type. The next time the call is performed with the same receiver type the runtime can jump directly to the start of the method. One advantage of this method is the reuse of runtime behaviour when inlining a method. Since inlining is an advanced compile time optimization it is usually reserved for methods that are frequently called. Frequently called methods are hot spots in the code that would improve overall performance the most if they were optimized. When performing inlining under these conditions the PIC would inform the compiler what receiver types a variable has had at a call site. The inlining logic can use the PIC to predict the type that the receiver has been called with at runtime. SELF also implements type prediction based on historic usage. For example arithmetic message sends like +, -, *, and / are 90% of the time messages sent to numeric type Integer. With type prediction SELF will inline the method body of the mostly likely receiver type and perform a type check that will either jump to the inlined instructions or execute the original dynamic send.

3.2 Truffle/GraalVM

Truffle-Squeak[8] depends on Truffle's Language Agnostic Inlining[11, 4]

3.3 V8 JavaScript

JavaScript also does inlining[3]. JavaScript has some challenges beyond those of Smalltalk, but with explicit control structures, it is simultaneously easier to inline.

4 Inlining

Zag Smalltalk [5] uses a stack-based runtime with threaded instructions or native methods generated with LLVM[1].

4.1 Zag Code Generation

Zag Smalltalk stores all methods as Abstract Syntax Trees (ASTs). When a method is first required to be executed, it must be converted from an AST to either a threaded method or a native-code method. It is beyond the scope of this experience report to discuss the decisions about when a native-code version is produced.

4.1.1 Abstract Syntax Tree. The AST has Method and Block objects, which both have bodies composed of a sequence of expressions, optionally terminated with a return. Expressions include: variable Assign, Array, Send/Cascade, Literal, variable Reference, `self`/`super`, and ThisContext. This is a simplified version of the Pharo AST, and can easily be generated from Smalltalk source, or by walking a Pharo AST.

Regardless of whether generating for a threaded or native target, the AST compiler works in two stages: conversion to basic blocks, followed by target code generation.

4.1.2 AST to Basic Blocks. The first stage converts the AST into a series of basic blocks, made up of a sequence of stack operations (sends, pushes, pops, returns, etc.). Each basic block ends with a send, return, branch, or class switch. Part of this stage is an optional target-independent inlining of methods and blocks. As part of inlining, many of the sends may be replaced.

The first basic block is a MethodBlock. A MethodBlock sets up a compile-time stack that represents the `self`, parameters, and locals of the method (or BlockClosure). The basic conversion outputs a sequence of pushes and pops, mimicking the runtime stack as well as adding the corresponding operations to the current basic block. When a send or class switch is required it forces the creation of a new basic block.

If this was a send then the new basic block will be a ReturnBlock. Because the send will execute code we know nothing about, any and all values must be represented on the stack or the heap. A ReturnBlock will be initialized with a stack as it would be on return from the message send, that is with the stack as it was just before the send, but with the receiver and parameters of the send replaced by a result value. Then we resume the basic conversion.

Figure 1 shows a version of the `fibonacci` method on Integer with no inlining. Note that there are many different ReturnBlocks and an expensive send that returns to each.

Figure 2 shows a version of the `fibonacci` method on Integer with extensive inlining. Note that there are now only two sends and hence ReturnBlocks and most sends have been recognized because of the primitives they correspond to when inlined, and the result of the comparison is known to be boolean, so we can do a class switch on `True` and `False`.

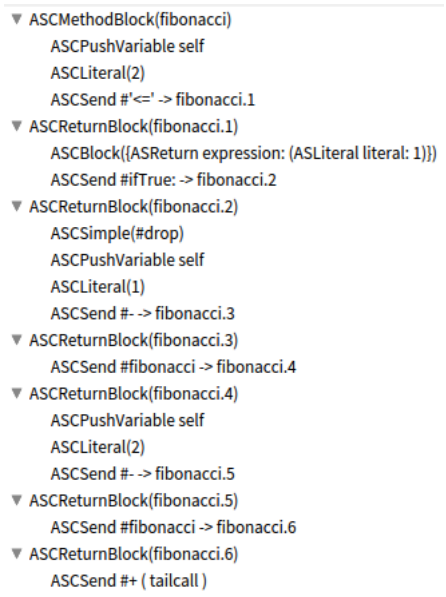

Figure 1. `fib` with no inlining

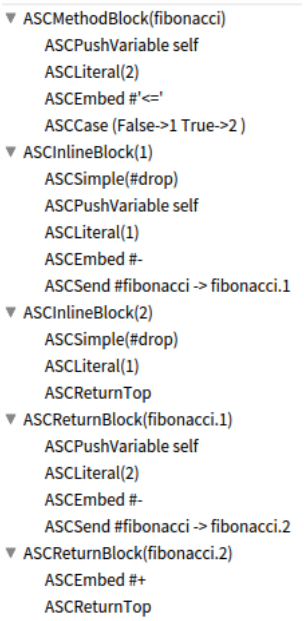

Figure 2. `fibonacci` with full inlining

If this was a class switch the new basic blocks will be an InlineBlock. Each of the new basic blocks will have a stack initialized from the stack at the point of the creation of the class switch. Because of the structural nature of the replacement of the send → class switch, all of the various paths will eventually come to a common InlineBlock which will have a stack the same as if the send was still in place, that is with the receiver and parameters replaced by a result. A path that is a

sequence of InlineBlocks is highly advantageous for a target like LLVM because they will all be part of a single function which gives the low-level optimizer lots to work with, and values don't need to be actually saved to memory. If, along any of the paths an actual send is required, because there is no safe inlining that can be done, a ReturnBlock may be required, in which case the common block will be converted to a ReturnBlock, at some performance cost.

After all the basic blocks have been generated, the data-flow graph is updated for the basic blocks, if it is required by the target code generator. This integrates the *require/provides* values of the various blocks. We iterate over all the blocks, taking the values that are required for a given basic block and telling all of the blocks that transfer to it that it needs those values. This may make those basic blocks now require the values, so we continue the iteration until we reach a fix-point, where nothing is changing. This analysis can reduce the memory traffic for native targets such as LLVM.

Method dispatch uses both the class and the selector to find a compiled method in a dispatch hash table[10]. If a compiled method is not found for the given selector in the classes dispatch table, the method is compiled from the AST and installed into the dispatch table for the receiver class[6]. Because Zag compiles code for a concrete target class, all sends to `self` and `super` are identifiable and can be potentially inlined.[1] Similarly, during compilation any literals found or inferred in the code can be safely typed and the message send potentially inlined. See the trivial example in listing 2.

Our method of inlining will keep track of the types of variables from assignments and method dispatch through the stack elements. When a send is encountered, if the class of the target is known, we will inline the corresponding method, in the cases of literals, `self` and `super`. Alternatively, if the number of implementors is small enough, the message send will be replaced by a class case instruction which is based on the dynamic classes of the receiver. At runtime the case instruction will attempt to match the runtime type of the class of the receiver, when a match is found a jump to the correct inlined method will be taken. The method being compiled will have sections of instructions that have been extracted from the method bodies of the sends, one for each type found. Since the case was derived from the current image's known implementors, at runtime a match should be found. If the runtime target type does not match any of the types for the class switch statement we will default to performing the message send which likely will result in a Does Not Understand (DNU) being thrown.

4.2 Inlining Strategies

Inlining is the primary target-independent optimization for the Zag Smalltalk compiler. This is similar to SELF; it is available because methods are compiled for each target (class in Smalltalk, map in SELF).

[1]Note "potentially" because the inlining budget may be reached so that no additional inlining will be performed.

```
foo2
    ^ self bar
bar
    ^ 42

After inlining would result in
foo2
    ^ 42
```

Listing 2. Inlining `self` method send

When the Zag compiler is asked to compile a particular selector for a target class, it looks at the class and its superclasses for the corresponding method, stored as an abstract syntax tree (AST). It is translated to an intermediate form of a series of basic blocks - each ending in either a send or a return. This is perfectly executable and, in fact, is the initial implementation for a method. If it is executed frequently enough, it will get re-compiled with some level of inlining.

When inlining, the compiler looks at each block that ends with a send, and tries to find a way to replace the send with a semantically equivalent operation - i.e. to inline the send. This is repeated as long as there is a send that could be replaced.

In some cases all sends can be replaced, in which case there isn't even a reason to create a Context, saving a considerable number of instructions.

There are seven patterns that we have identified that can be inlined, based on what we can determine about the receiver.

4.2.1 Send to self, super or known type.

The first pattern occurs when a send is sent to `self`, `super` or a known type, in which case - like static programming languages - we are able to uniquely identify the method and inline the AST into the current compiled method's basic blocks. We create a new block which will contain the inlined instructions from compiling the AST for the designated method. The only restriction for this case is when the method has a recursive send (a call to a method we're currently inlining (any enclosing one)). While compiling the AST of the inlined method we simply rename the top elements of the stack to self and the names of the parameters, and instead of returning, the code would branch to a new block that contains the instructions performed after the send was called.

For example when compiling a method see listing 2. The result of inlining bar, where bar returns a literal 42, and foo2 returns a self call to bar are the blocks depicted in figure 3. In this example the resulting AST has three blocks.

1. The first block is a `MethodBlock` which prepares the call by pushing the receiver, in this case `self` onto the stack. After `self` is pushed onto stack as a temporary a jump occurs to the inline block

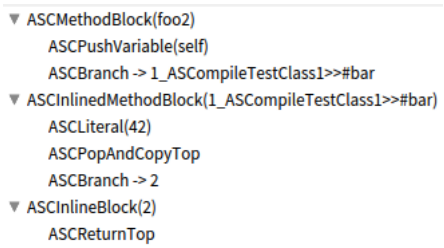

Figure 3. `foo2` calls `bar` with inlining

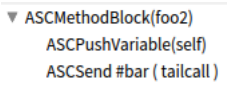

Figure 4. `foo2` calls `bar` without inlining

```
recursiveMethod
        self recursiveMethod
```

Listing 3. Trivial recursive self send

2. The inline block of this simple bar method simply pushes the return value, in this case a literal 42 onto the stack. The instruction to PopAndCopyTop will copy the top value, which is the result, and remove any pushed parameters as well as
   self from the stack that were added to prepare for the original send. Then the result is pushed back on the stack, in effect cleaning up the stack.
3. the last block jumps back to the calling method which in this case simply returns the top value on the stack to the caller.

Without inlining the foo2 would be a simple send, see figure 4, and in this case we have identified it as a tail call send which may result in an optimization that replaces the foo call with just the bar call, when possible.

4.2.2 Recursive send to self, super or known type. A recursive send to self can not be inlined. For example see listing 3. However, we know exactly the code that should be executed, so we can save the return addresses as if it were a normal send, and then branch directly to the target method as seen in figure 6. The output block for this method call pushes self onto the stack in preparation for the send. The recursive send is recognized and modifies the stack by removing the previously generated send and then branching back to the top of the block as seen in 5.

4.2.3 Recursive tail-call send to self, super or known type. If the previous recursive case the send is a tail-call as shown in listing 4, then there is no need to save the return addresses, but rather a simple branch to the target method. This creates a loop as seen in figure 6.

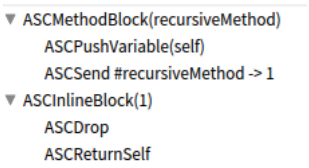

Figure 5. simple recursive self send

```
recursiveMethod
        ^ self recursiveMethod
```

Listing 4. Trivial recursive self send

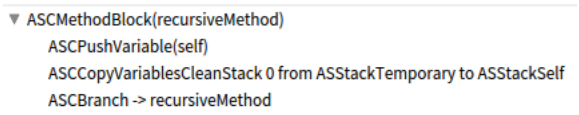

Figure 6. simple recursive self send

```
primCall
        1 + 1
```

Listing 5. Trivial primative send

4.2.4 Send to self, super or known type where the method is primitive. A method with a primitive send as seen in listing 5 is replaced with with a primitive operation and passes control to the original return block. The blocks as seen in figure 7 shows the send for plus is in fact replaced with a direct call to the embedded instruction for plus, which replaces the send. The non inlined version of the method can be seen in 8 shows a send being performed. This may require that the receiver and parameters provably have certain properties (like they are both `SmallIntegers`). If we have weaker knowledge we may be able to use a primitive, but it may dispatch a send if the appropriate properties don't hold at runtime (like the result of an add of two `SmallIntegers` produces a `LargeInteger`). In this case we will have to force a `Context` to have been created in this method.

4.2.5 Send value, value:, etc. to a literal BlockClosure. This is a special case of a primitive method send. This is very similar to the first case, with two notable exceptions:

1. any explicit return is a non-local return, so it will need to branch to the return point that the method in which the block is defined would use;
2. when doing name resolution, when we get to the position of the receiver on the stack, we skip to the scope where this block is defined.

4.2.6 Send where there are few implementations of a method. If there are only a few classes that have access to an implementation of a particular message (i.e they have it

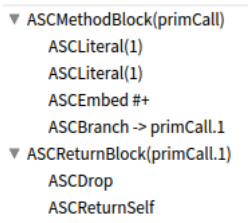

Figure 7. A primitive addition inlined example send

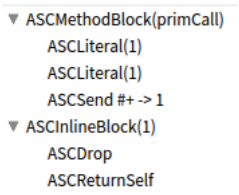

Figure 8. A primitive addition non-inlined example send

themselves or a superclass has an implementation), we emit a class-case instruction and inline the method for each of them, with each of them branching to the original return block on "return". To have the correct/conservative semantics, there is also a fall-back of doing the original send, unless we can prove that the list is exhaustive.

4.2.7 Send where the target is the result of a comparison primitive.

As a special case of the previous case, we know that comparison primitives always return `true` or `false` (or an error if the values are incomparable), so we have an exhaustive list. But more, we now know something about the relationship of these values. So, for example, we might know that a value is in the range of 1 to the size of an array, which means that we can safely use that value to index into the array.

5 Conclusions and Future Work

We are currently implementing inlining described in this experience report, for the dynamic programming language Smalltalk. Once completed we will run benchmarks to confirm inlining produces higher-performing code.

Future Work

We are currently doing nothing to handle code explosion from inlining. We don't anticipate this being a problem, as (non-tail) recursive calls stop inlining and inlining seems to create smaller code as often as it creates larger code. There are a lot of other optimizations we expect to perform including local type inference, recognizing inlined methods whose results are discarded, recognizing more primitive special cases.

References

[1] Janat Baig and Dave Mason. 2024. Smalltalk JIT compilation: LLVM experimentation. In *International Workshop on Smalltalk Technologies*.

[2] C. Chambers and D. Ungar. 1989. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (PLDI '89). Association for Computing Machinery, Portland, Oregon, USA, 146–160. ISBN: 089791306X. DOI: 10.1145/73141.74831.

[3] Google. [n. d.] Javascript inlining heuristics. Retrieved Oct. 7, 2024 from https://chromium.googlesource.com/v8/v8/+/refs/heads/lkgr/src/compiler/js-inlining-heuristic.cc.

[4] GraalVM. [n. d.] Truffle approach to function inlining. Retrieved Oct. 7, 2024 from https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/Inlining.

[5] Dave Mason. 2022. Design principles for a high-performance smalltalk. In *International Workshop on Smalltalk Technologies*. https://api.semanticscholar.org/CorpusID:259124052.

[6] Dave Mason. 2023. Revisiting dynamic dispatch for modern architectures. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2023). Association for Computing Machinery, Cascais, Portugal, 11–17. ISBN: 9798400704017. DOI: 10.1145/3623507.3623551.

[7] Dave Mason. 2023. Threaded execution as a dual to native code. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Programming '23). Association for Computing Machinery, Tokyo, Japan, 7–11. ISBN: 9798400707551. DOI: 10.1145/3594671.3594673.

[8] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. Graalsqueak: toward a smalltalk-based tooling platform for polyglot programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (MPLR 2019). Association for Computing Machinery, Athens, Greece, 14–26. ISBN: 9781450369770. DOI: 10.1145/3357390.3361024.

[9] D. Ungar, R.B. Smith, C. Chambers, and U. Holzle. 1992. Object, message, and performance: how they coexist in self. *Computer*, 25, 10, 53–64. DOI: 10.1109/2.161280.

[10] Jan Vitek and R. Nigel Horspool. 1996. Compact dispatch tables for dynamically typed object oriented languages. In *Compiler Construction*. Tibor Gyimóthy, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, 309–325. ISBN: 978-3-540-49939-8.

[11] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (SPLASH '12). Association for Computing Machinery, Tucson, Arizona, USA, 13–14. ISBN: 9781450315630. DOI: 10.1145/2384716.2384723.