Inlined Code Generation for Smalltalk

Daniel Franklin[1,†], Dave Mason[2,*,†]

Toronto Metropolitan University, Toronto, Canada

Abstract

In this paper 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. 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. There are several challenges to inlining messages in Smalltalk that need to be resolved like detecting cycles in the call graph, resolving methods arguments to their equivalent in the calling method, and handling non-local returns in block arguments. In this paper, we describe the inlining approach taken for the Zag Smalltalk compiler that solves for these issues and improves performance.

Keywords

compile, inlining, Smalltalk, method dispatch

1. Introduction and Motivation

Zag Smalltalk [1] 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, 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[2]. Currently, Smalltalk implementations like Pharo only inline a known list of special methods. 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. 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 implementers of the message's selector within the current image. If there are few implementers of the selector then it is a candidate to be inlined at compile time. Based on these findings, inlining should not be a complex task and will result in more opportunities for a compiler to optimize and to avoid the overhead of the message send.

2. Background

Inlining is not a new compiler optimization but it is rare for dynamic languages like Smalltalk where the receiver class of a message send is not statically known. Dynamic programming languages provide incredible flexibility to the programmer. However, this flexibility is at the cost of message sends that

IWST 2024: International Workshop on Smalltalk Technologies, July 9–11, 2024, Lille, France

*Corresponding author.

†These authors contributed equally.

daniel.franklin@torontomu.ca (D. Franklin); dmason@torontomu.ca (D. Mason)

https://github.com/dfranklinsmail (D. Franklin); https://sarg.torontomu.ca/dmason (D. Mason)

0009-0000-6920-2522 (D. Franklin); 0000-0002-2688-7856 (D. Mason)

require significantly more overhead than static languages when looking up the method to be invoked and setting up the stack for the call.

The dynamic programming language SELF[2] implements a type inference mechanism to find all 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 type 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 implementers 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. 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 paper.

```
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 target and arguments are found on the stack and pushed onto the top of the stack. A dynamic dispatch is then performed which consumes the newly added elements on the stack and add a return value on the stack. Programmers often make use of the double dispatch [3] pattern, for example when visiting tree structures. If double dispatch is detected a simple stack optimization can be performed.

3. Method

Zag Smalltalk [1] uses a stack-based runtime with threaded instructions or native methods generated Just In Time with LLVM[4].

3.1. 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 JITed method. It is beyond the scope of this paper to discuss the decisions about when a JITed version is required.

3.1.1. 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.

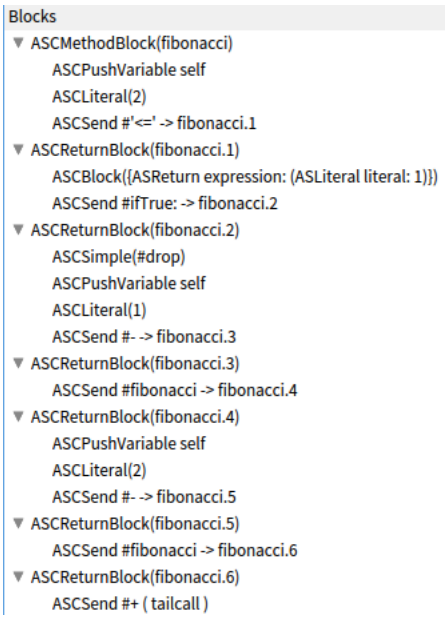

Figure 1: `fib` with no inlining

3.1.2. 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 with switches based on the class of the receiver.

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. This is the exact semantics we you expect from the stack operations, but as we will see below this requires particular attention when generating LLVM code. 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`.

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 and values don't need to be actually saved to memory, as we'll see below. 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 for a target such

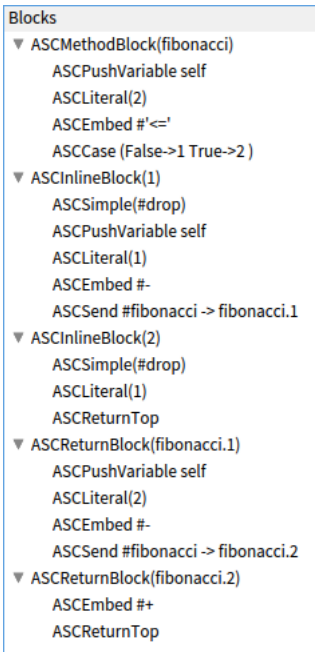

Figure 2: `fibonacci` with full inlining

as LLVM.

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. 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[5]. The advantage of dynamic dispatch languages like Smalltalk allow the compiled to know upfront what class a method is being compiled for, which means that all sends to `self` and `super` are identifiable and can be potentially inlined. Also, during compilation any literals found or inferred in the code can be safely typed and the message send inlined. See the trivial example in Listing 1.

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 implementers is small enough, the message send will be replaced by a class case instruction 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. Since the case was derived from the current image's known implementers 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.

```
abc
    ^ self def isNil
def
    ^ 42
abcAfterInlining
    ^ false
```

Listing 2: Inlining `self`

3.2. Rules for Inlining a Method's Operations

To include the calling methods statements into the receiver, refactoring with the following rules that handle references to locals, parameters, instance variables, class variables, `self` or `super` are required. References must be refactored with the following rules:

1. all names are accessed by going up the stack, so names of locals will be found before other names of enclosing inlining;
2. `self` and parameters are automatically set up as named versions of the values that were passed as arguments to the send;
3. the return point for a method needs to be accessible to blocks defined in that method so that "non-local" returns can simply jump to that exit point;
4. tail-recursive sends are recognized as loops so writing `whileTrue:` and related functions as tail recursive makes them zero cost.

4. Conclusions and Future Work

We are currently implementing inlining described in this paper, 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. We are currently inlining depth-first, which may not be optimal.

References

[1] D. Mason, Design principles for a high-performance smalltalk, in: International Workshop on Smalltalk Technologies, 2022. URL: https://api.semanticscholar.org/CorpusID:259124052.

[2] D. Ungar, R. Smith, C. Chambers, U. Holzle, Object, message, and performance: how they coexist in self, Computer 25 (1992) 53–64. doi:10.1109/2.161280.

[3] L. Bettini, S. Capecchi, B. Venneri, Translating double dispatch into single dispatch, Electronic Notes in Theoretical Computer Science 138 (2005) 59–78. URL: https://www.sciencedirect.com/science/article/pii/S1571066105051352. doi:https://doi.org/10.1016/j.entcs.2005.09.011, proceedings of the Second Workshop on Object Oriented Developments (WOOD 2004).

[4] J. Baig, D. Mason, Smalltalk jit compilation: Llvm experimentation, in: International Workshop on Smalltalk Technologies, 2024.

[5] D. Mason, 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, New York, NY, USA, 2023, p. 11–17. URL: https://doi.org/10.1145/3623507.3623551. doi:10.1145/3623507.3623551.