Inlined Code Generation for Smalltalk

By Daniel Franklin and Dave Mason
Toronto Metropolitan University

Content

- Zag Smalltalk
- Why is Inlining Important
- Zag Smalltalk Architecture
- Inlining Scenarios
- A Practical Example

Zag Smalltalk

https://github.com/dvmason/Zag-Smalltalk

Objective of this project is to unburden the programmer from having to code for performance. Rather it should be the compiler and runtime’s job to take a program and generates efficient code which runs in an efficient VM.

- Multi threaded
- Advanced Garbage Collector
- JIT compilation using LLVM
- Inlining of methods

Inlining In Dynamic Languages

- currently, Smalltalk implementations like Pharo only inline a known list of special methods like `ifTrue:ifFalse:`, `to:do:` and `whileTrue:`
- many methods on collections are not included
- SELF was the first dynamic language with inlining support
- Nahuel Palumbo was experimenting at last ESUG

Why is inlining important?

- Static vs dynamic programming languages
- Typically Smalltalk Methods are unique
- Smalltalk methods have few expressions
- Provide optimization opportunities

number of implementors	method count
1	45739
2	7988
3	2938
4	1515
5	960

Table 1: Comparing the number of selectors with 1 or more implementations.

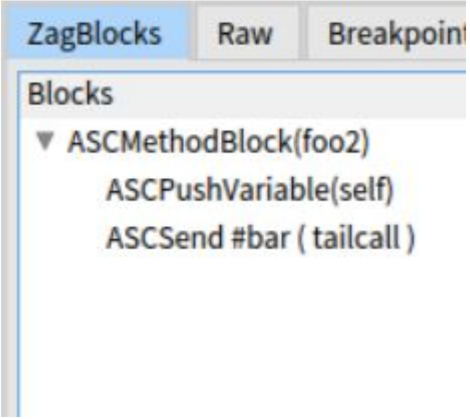
expression in method	numbers of methods
0	1429
1	83837
2	15752
3	10560
4	6238

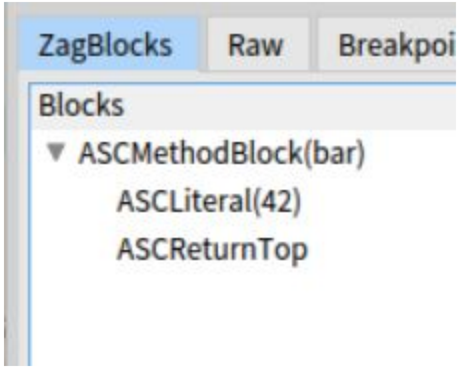
Table 2: Comparing the number of expressions in a method.

Simple Example

```
foo2
    ^ self bar
bar
    ^ 42
```

Before Inlining:



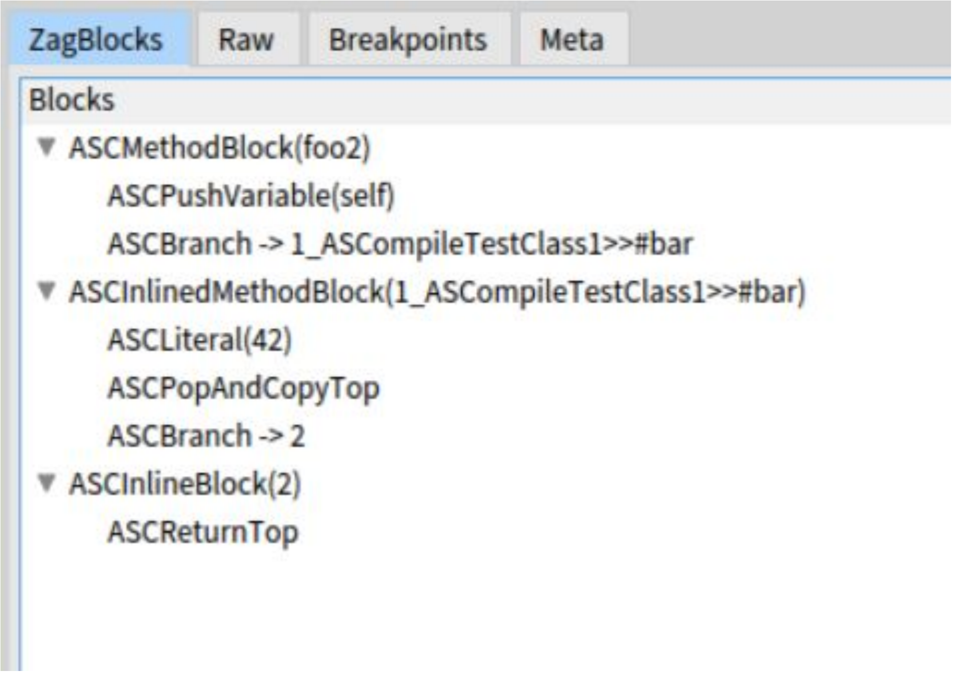


Simple Example

```
foo
    ^ self bar
bar
    ^ 42
```

After Inlining:

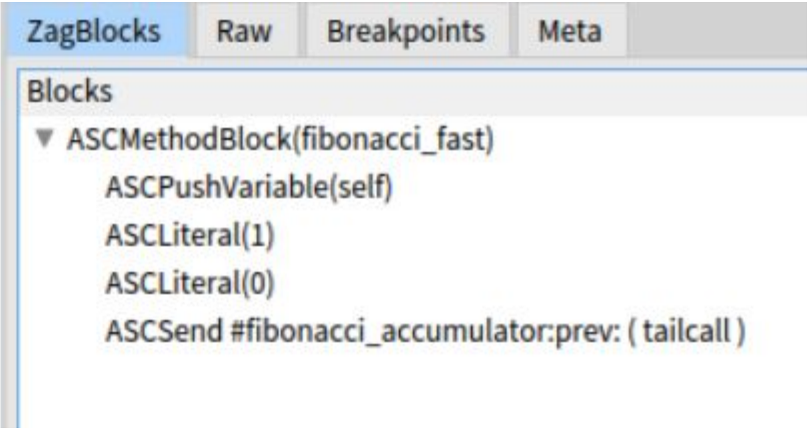
```
fooInlined
    ^ 42
```



Fibonacci Fast Example

```
fibonacci_fast
   ^ self fibonacci_accumulator: 1 prev: 0
```

Before Inlining:

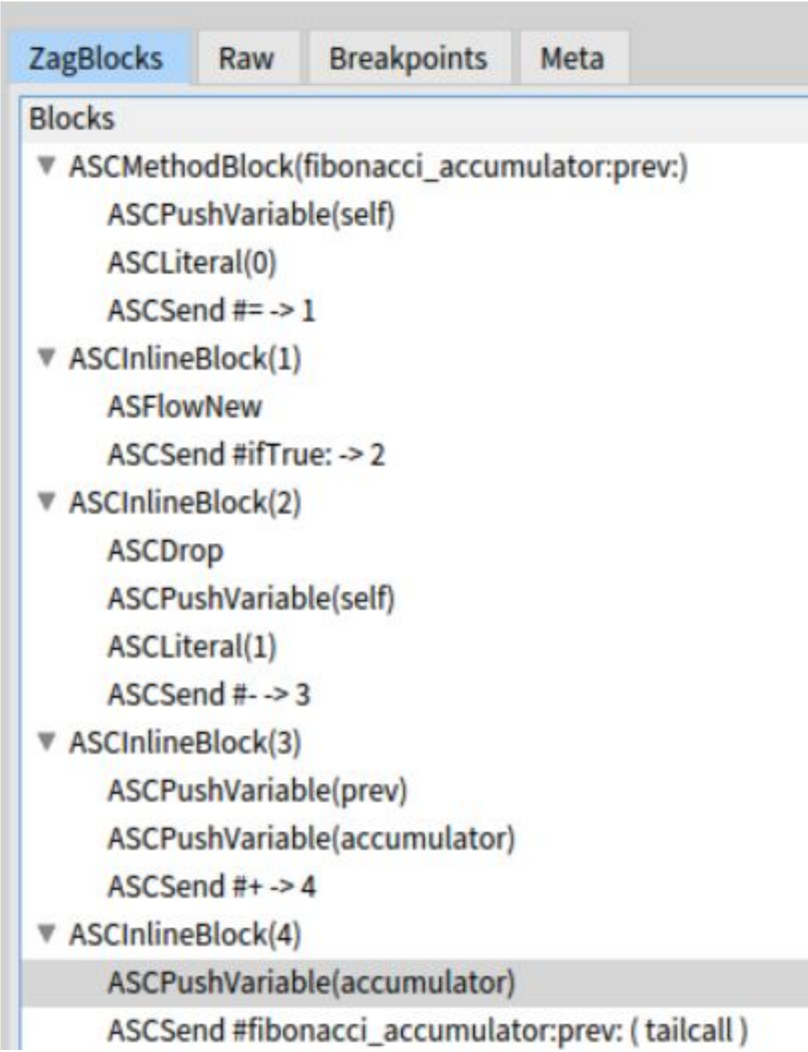


Fibonacci Fast Example

```
fibonacci_accumulator:
   accumulator prev: prev

   self = 0 ifTrue: [ ^ prev ].
   ^self-1
      fibonacci_accumulator:
         prev + accumulator
      prev: accumulator
```

Before Inlining:



Zag Architecture

- Method Dispatch
    - all methods are stored in AST form
    - a method is compiled for target class known at runtime rather than the class the method is defined in
    - this means that “self” and “super” are known

Inlining

5 kinds of inlinable code:

- self/super/literal sends
- primitives with known parameters (includes blocks)
- type-inferred sends
- sends with limited number of implementations (with fallback to DNU)
- tail-recursive self-sends

Inlining - self/super/literal sends

- because methods are compiled for a target class, we know exactly what methods are referred to by self/super
- because methods are compiled for the class corresponding to a literal, we know exactly …
- e.g. `SequenceableCollection>>#do:` (compiled for Array)

```
    do: aBlock
        1 to: self size do: [
          :index | aBlock value: (self at: index)
       ]
```

- if inlined, `aBlock` would also be literal

Inlining - primitives with known parameters (includes blocks)

- e.g. `Number>>#negated` (compiled for SmallInteger or …)

```
      ^ 0 - self
```

Inlining - type-inferred sends

- e.g. `anObject < another ifTrue: [...] ifFalse: [...]`
- comparators always return Boolean, so no possibility of DNU

Inlining - sends with limited number of implementations

- even without knowing the receiver type
- needs fallback to send with potential for DNU
- e.g. `anObject ifTrue: [...] ifFalse: [...]`
- needs fallback to send with potential for DNU

Inlining - tail-recursive self-sends

- copies the appropriate elements from the stack to the target locations (from the target basic-block)
- cleans up the stack
- branches to the target basic-block
- e.g. `BlockClosure>>#whileTrue:`

```
whileTrue: aBlock
    self value ifFalse: [ ^ nil ].
    aBlock value.
    ^ self whileTrue: aBlock
```

Compilation

- compiling a method from a class or ancestor for a target class
- produces a set of basic blocks each ending in a send or a return (only the last one)
- inlining pass looks at all basic blocks that end in a send, and try to inline them (inlining may produce more basic blocks to examine)
- do data flow analysis to convert to single static assignment (SSA) form (for e.g. LLVM target)
- potential peep-hole optimization on intermediate form
- generate code for target (LLVM, threaded, etc)

Compile-time data structures

- a compile-time stack mirrors the runtime stack
- when inlining a method, a new basic-block is created, and with a copy of the stack from the previous basic-block(s) but with the temporaries labeled with the parameter/self values, then the AST for the method is visited
- when inlining a block, a new basic-block is created, and with a copy of the stack from the previous block(s) but with the temporaries labeled with the parameter values the position of the block on the stack is an object that will look up names in the originating method, then the AST for the block is visited

Stack Representation

```
Collection>>#add:withOccurrences:
   add: newObject withOccurrences: anInteger

   #(1 2 3 4) add: 10 withOccurences: 5
```

Before call	After call	Inlining Call	After inlining
…..	….	….	….
#(...)	anObject	#(...) (self)	anObject
10		10 (newObject)	
5		5 (anInteger)	

Future Work

- Benchmarking How Fast Are We?
- Generating executable code.
- Type Inference for better compiling
- peephole optimizer