Feasibility of leveraging LLVM for JIT Compilation In Dynamic Language Ecosystems

Janat Baig[1]

Abstract

Just-In-Time (JIT) compilation plays a crucial role in improving the performance of dynamic languages, which often face unique challenges such as runtime type resolution and method dispatch. This thesis explores the viability of LLVM as a JIT compiler for dynamic languages, focusing on Smalltalk as a case study. We provide an overview of the relevant compiler pipeline, detailing how LLVM was integrated to generate optimized code for Smalltalk's object-oriented model. This is all to illustrate whether the effort to integrate LLVM into a dynamic language ecosystem is worth it.

Keywords

LLVM, JIT code generation, Smalltalk

1. Thesis Statement

This work evaluates the integration effort required to adopt LLVM into a dynamic language runtime and assesses whether the performance improvements are substantial enough to warrant that effort.

2. Introduction and Motivation

Zag Smalltalk [1] is a from-scratch implementation of a Smalltalk virtual machine whose low-level is implemented in Zig [2]. The project aims to implement a higher-performance Smalltalk as well as to inspire and support existing Open Smalltalk systems.

To further advance Zag's performance, the integration of a Just-In-Time (JIT) compiler using LLVM [3] is underway. We deem LLVM as a suitable choice for developing our JIT as it is known to offer a modular and reusable compiler toolset that supports a wide range of languages and platforms. Its intermediate representation (IR) enables powerful optimizations and efficient machine code generation.

3. Dynamic Languages & JIT Compilation

Dynamic languages such as Smalltalk are designed to be highly flexible and developer-friendly. One of their core characteristics is that many programming decisions — such as determining the type of a variable or resolving which method to call — are delayed until the program is actually running. This stands in contrast to statically compiled languages like C or Rust, where such decisions are made at compile time, allowing the compiler to generate efficient machine code in advance. Because dynamic languages defer this kind of analysis, they are able to support features like dynamic typing, reflection, and runtime method injection—but they often pay for this flexibility with slower performance at runtime.

Since much of the decision-making is postponed, dynamic language implementations, relative to their runtime, typically perform little work during compile time. Their compilers are primarily responsible for converting source code into an intermediate representation — often bytecode or an abstract syntax tree (AST) — and preparing the runtime system that will carry out the actual execution and optimization. Type checking, method resolution, and even function dispatch are handled using

janatbaig2@gmail.com (J. Baig)
https://github.com/JanBaig/ (J. Baig)

runtime logic and/or dynamic dispatch tables. This shifts much of the complexity away from the compiler and onto the runtime environment.

Zag-Smalltalk, as an implementation of a dynamic language, adopts the same philosophy. It minimizes work during the compile-time phase, focusing only on preparing the essential components needed for execution. For instance, at compile time, Zag transforms the core compiler logic — responsible for converting ASTs into threaded code (as explained in Section 4.5) — from its initial AST form into threaded code. This way, when execution begins, the runtime already has everything it needs to dynamically resolve methods, manage message sending, and apply optimizations — all without relying on traditional ahead-of-time compilation.

To address the performance limitations that come with this dynamic behavior, dynamic language systems often employ Just-In-Time (JIT) compilation — making it a natural choice for Zag to pursue as well. A JIT compiler is a runtime component that translates portions of a program — typically "hot" or frequently executed code — into machine code during execution, rather than before the program starts. What sets a JIT apart is its ability to leverage runtime information such as type feedback, branch prediction, call frequency etc to optimize code while it is being compiled. This enables the JIT to apply aggressive optimizations that static compilers cannot. Meanwhile, "cold" or rarely executed code can be compiled with minimal effort to reduce overhead. The Java Virtual Machine (JVM) is a well-known example of a runtime system that uses JIT compilation to strike a balance between flexibility and performance [4].

As explained in more detail below, during runtime, Zag stores all methods in AST form until they are invoked. When a message is sent to an object, the system consults the target class's dispatch table to find the appropriate method. If the method has already been compiled, the dispatch entry points directly to its compiled form. If not, the entry points to fallback code that triggers compilation. The method is first compiled into an unoptimized threaded form, inserted into the dispatch table, and executed. If it is invoked frequently enough, it is recompiled into an optimized threaded version. After additional execution, the JIT is invoked and the method is recompiled into LLVM IR. This IR is then optimized and compiled using LLVM's JIT compilation infrastructure. Our goal is to generate this LLVM IR directly in memory via the LLVM C API, enabling us to apply both built-in and custom optimization passes within the Zag runtime.

4. Zag execution model

Code execution models exist along a spectrum — from bytecode interpreters, which operate on compact, intermediate representations of code, to ahead-of-time (AOT) compilers that produce native machine instructions for a specific target architecture. Each point along this spectrum reflects a trade-off among code compactness, execution speed, and implementation complexity.

Zag adopts a dual execution model — threaded code and continuation passing style (CPS) execution. Methods are initially compiled into threaded code for compactness, debugging, and ease of execution, and can later be recompiled into native code using CPS for performance-critical execution paths. While each execution model in Zag serves a different purpose, they share a unified calling convention that enables seamless switching between the two models (as explained in subsection 4.1). This allows us to switch from CPS to threaded mode for debugging or vice versa for performance-critical paths.

The following sections introduce key infrastructure concepts — such as compiled methods, primitives, and the Zag calling convention — and provide an in-depth explanation of each code generation model, detailing how control is passed between methods, how continuations are represented, and the distinct advantages each model offers in terms of performance and debuggability.

4.1. Calling Convention

Zag uses a standardized calling convention, referred to as a threaded function (`threadedFn`), to unify both threaded and continuation-passing style (CPS) execution. This shared function signature ensures compatibility between the two execution modes and enables seamless transitions between them during runtime.

Listing 1 shows a function that adheres to the `threadedFn` calling convention (in Zig).

1. `pc` — points to a threaded program counter.
2. `sp` — points to the top of the stack.
3. `process` — points to the private data of this process (which includes the stack).
4. `context` — points to the context of the calling method (on the initial execution of a method), or the current context as needed to store state before a message send.
5. `extra` — a multi purpose value (outside scope of this thesis)

```
pub fn someThreadedFn(pc: PC, sp: SP, process: *Process, context: *Context, sig:
    Signature) SP {
    return @call(tailCall, ...);
}
```

Listing 1: Zag Calling Convention

```
&someThreadedFn // <- a threaded word
```

Listing 2: A threaded word

4.2. Compiled Methods

In Zag, every method is represented at runtime by a `CompiledMethod` object. This structure serves as the compiled form of a Smalltalk method and holds all the information necessary for the runtime to begin the method's execution.

Listing 3 showcases the struct definition for a `compiledMethod`.

1. The `code` field contains the method's implementation in threaded code — essentially an array of threaded word addresses.
2. The `executeFn` field holds the address of a threaded function that the runtime can begin executing. Initially, it stores the address of the first threaded word from the `code` array.
3. The `jitted` field holds a pointer to a version of the function that has been compiled to machine code via LLVM's ORC JIT.
4. The `header` contains a pointer to the heap header. A heap header contains information on the class type, locals and instance variables that an instance of a class contains.

```
pub const CompiledMethod = struct {
    header: HeapHeader,
    stackStructure: StackStructure,
    signature: Signature,
    executeFn: ThreadedFn.Fn,
    jitted: ?ThreadedFn.Fn,
    code: [codeSize]Code, // the threaded version of the method
    // other struct-specific methods defined (in Zig)
}
```

Listing 3: A Zag CompiledMethod

4.3. Primitives

Primitives are low-level operations that interface directly with the underlying system and are implemented outside of a normal Smalltalk system. They are not written by users, nor are they generated by the standard Smalltalk compiler. Instead, primitives provide essential functionality — such as memory access, basic arithmetic, and I/O — by calling into external standalone machine code that is included in smalltalk environments [5].

Because primitives are responsible for foundational tasks that are referenced constantly during program execution, they effectively serve as the equivalent of atomic operations. Their existence is crucial for smalltalk-like systems: without them, the system would rely entirely on high-level interpretation for even the simplest operations, resulting in severe performance penalties. By compiling these operations in advance and invoking the optimized machine code directly at runtime, Smalltalk systems preserve the flexibility of a dynamic environment without sacrificing the speed needed for efficient execution of core behaviors. Note that primitives, unlike sends, do not create new contexts for themselves. They execute in the current context as they're supposed to be fast, minimal and atomic [6].

In OpenSmalltalk-based systems like Pharo and Squeak, many primitives are written in Slang, a restricted subset of Smalltalk designed to be automatically translated into C. This allows primitive code to be maintained through methods written in familiar Smalltalk syntax: instead of manually managing C functions, Smalltalk developers write methods inside the image that include a pragma declaration like `<primitive: N>`. A pragma acts as a special annotation. When a method includes a `<primitive: N>` pragma, it tells the VM: "When this method is called, first attempt to run primitive number N." If the primitive succeeds, the Smalltalk method body is skipped; if it fails (for example, due to unexpected types), the Smalltalk code is executed as a fallback. Listing 4 showcases this for the implementateion of `#+` for the `LargeInteger` class.

```
+ anInteger
        <primitive: 21>
        ^super + anInteger "fall back code"
```

Listing 4: Primitive Pragma for #+

Importantly, developers can browse and edit the Smalltalk methods that invoke primitives — including modifying the fallback logic. However, the actual implementation of the primitive itself (the low-level external standalone machine code associated with primitive number N) is not modified inside the Smalltalk environment; it is generated externally through the Slang-to-C pipeline and compiled into the virtual machine.

In Zag, primitives are implemented directly in Zig, without relying on Slang or a C backend. Primitive definitions are grouped into modules, each handling a distinct category of operations. For example, primitives related to the `process`, `context`, `LLVM` etc are all categorized into different groups of modules. As reflst:zag-primitive shows, the implementation of a primitive within Zag is in Zig, and it follows the threaded function calling convention as explained in section 4.1.

```
pub const @"+" = struct {
    pub const number = 1;
    pub fn primitive(pc: PC, sp: SP, process: *Process, context: *Context, extra: Extra) Result {
        // SmallInteger>>#+
        if (inlines.p1(sp.next, sp.top)) |result| {
            const newSp = sp.dropPut(result);
            return @call(tailCall, process.check(context.npc.f), .{ context.tpc, newSp, process,
                context, undefined });
        } else |_| {}
```

```
        return @call(tailCall, Extra.primitiveFailed, .{ pc, sp, process, context, extra });
    }
}
```

Listing 5: Zag Runtime Implementation of the Primitive +

4.4. Execution Model Switching

During runtime, a method initially begins execution in threaded code, which consists of an array of threaded word addresses. If the runtime detects that a method is being invoked frequently — due to recursion, loops, or other performance-critical behavior — it triggers JIT compilation of that method into native code (following a CPS-style layout). The `executeFn` field in the `compiledMethod` is then updated to point to the JIT-compiled native function, and execution switches from threaded to CPS accordingly.

However, if an interruption occurs during native execution (for example, for debugging or stepping), the system supports switching back to the threaded code. This is done by setting the `nativePC` field in the current `Context` to the value of the corresponding threaded function, which is available via the `threadedPC` field. This enables execution to resume in threaded code, which is easier to single-step and debug.

4.5. Threaded Code

Threaded code is a code execution technique that strikes a balance between time and space trade-offs. It involves a method being compiled into a sequence of threaded words where each word is a direct address pointing to a block of code that implements an operation. Each code block performs its task and then performs a tail call or indirect jump to the next code block [7][8]. In Zag, these code blocks are threaded functions as described in section 4.1. Figure 1b showcases a sequence of threaded words emitted for an expression that adds two numbers and stores the result into a local variable. This contrasts Figure 1a which showcases the bytecodes that are emitted with a bytecode interpretation model.

```
PUSH_LITERAL 1
PUSH_LITERAL 2
ADD
STORE_LOCAL x
```

(a) Bytecode Interpretation

```
&pushLiteral, 1
&pushLiteral, 2
&add
&storeLocal, x
```

(b) Threaded Code Representation

Figure 1: Side-by-side comparison of bytecode and threaded code for `x := 1 + 2`.

Aside from the obvious size difference of a bytecode and a threaded word (pointer size), the main difference between the models lie in how each word executes its functionality and later how it passes control to the next. For a bytecode interpreter, execution proceeds in a large switch loop that repeatedly fetches a bytecode instruction, decodes it to determine what operation to perform, and then dispatches control to the appropriate handler code for that bytecode. This model is simple and portable, but it incurs runtime overhead from instruction fetch, decoding and branching on every operation.

For threaded code, there is a program counter, `pc`, that is used to traverse over all the threaded words by always pointing to the next threaded word from the one currently being executed. Figure 6 showcases a threaded function `pushLiteral0(...)` which implements the threaded word `pushLiteral0` (simply an address to the threaded function). To execute the next threaded word, the function does a tail call at the end using the value that the current `pc` holds, and passing in an

incremented pc value, along with other threaded function arguments. There is no instruction decoding here or branching, just simple direct execution from one block to the next. This makes threaded code both faster than traditional bytecode interpretation and simpler to generate than full native code, while still being compact and flexible enough to support debugging or analysis tools.

```
pub fn pushLiteral0(pc: PC, sp: SP, process: *Process, context: *Context, sig:
    Signature) SP {
        const newSp = sp.push(Object.from(0));
        return @call(tailCall, pc.prim(), .{ pc.next(), newSp, process, context,
            undefined });
    }
```

Listing 6: Threaded Function for pushLiteral0

4.6. Continuation Passing Style (CPS)

Continuation Passing Style (CPS) is an execution technique in which, instead of returning results in the traditional way, a function calls another function, referred to as a callback, which handles the continuation of the program [9]. This model aligns naturally with functional programming's emphasis on composition and side-effect-free evaluation [6].

For CPS execution, to pass control, the current function directly does a tail call to the next function (it's name is known) while adhering to the threaded function signature (part of the calling convention as explained in section 4.1). Before making the tail call, the current method may need to store return addresses in the context if it expects execution to resume from those points later. Importantly, these saved addresses do not refer to a point within the current CPS function itself, but rather to another function. The continuation is maintained through the stack and context parameters. Also note that the pc parameter is only used when we want to fall back into debugging (threaded) mode.

One of the principles of CPS execution is that the system stack has no net change - any values pushed on it during execution of the function will be removed before doing a tail-call to the next block of code. This does not prohibit changes to other data structures in the program, such as a language-specific stack or heap.

The example below shows a CPS-style Fibonacci implementation, where control is passed using tail calls to named functions, and return points are explicitly stored using `context.setReturnBoth(...)`.

```
pub fn fibCPS(pc:PC, sp:Stack, process:*Process, context:ContextPtr, selector:Object)
    Stack {
    if (!fibSym.equals(selector)) tailcall dnu(pc,sp,process,context,selector);
    if (inlined.p5N(sp[0],Object.from(2))) {
        sp[0] = Object.from(1);
        tailcall context.npc(context.tpc,sp,process,context,selector);
    }
    const newContext = context.push(sp,process,fibThread.asCompiledMethodPtr(),0,2,0);
    const newSp = newContext.sp();
    newSp[0]=inlined.p2L(sp[0],1)
    catch tailcall pc[10].prim(pc+11,newSp+1,process,context,fibSym);
    newContext.setReturnBoth(fibCPS1, pc+13); // after first callRecursive (line above)
    tailcall fibCPS(fibCPST+1,newSp,process,newContext,fibSym);
}
```

Listing 7: CPS Example

A critical advantage of CPS over traditional threaded code is that CPS calls know exactly where to go next—eliminating the need for memory indirection through pc. This has two implications:

1. Reduced Memory Accesses: Threaded functions rely on pc to read the next function's address, resulting in an extra memory access. CPS functions skip this entirely, tail-calling a directly known function or label.
2. LLVM Optimization Potential: Since the next call target is statically known, CPS functions can be inlined and optimized more aggressively by LLVM. Control flow becomes more analyzable, allowing interprocedural optimizations that are hard to achieve with indirect jumps.

There's a subtlety here: CPS-style execution can still occur within a threaded context. In such cases, we use the context to obtain the next function rather than following pc. For instance:

```
return @call(tailCall, process.check(context.npc.f), .{ context.tpc, newSp, process,
    context, extra });
```

Listing 8: A threaded CPS call

This is often referred to as a threaded CPS call—the execution model is still CPS, but the surrounding infrastructure is built with threaded code conventions.

Finally, while Zag's threaded code does use tail calls (and hence resembles CPS), not all threaded code qualifies as CPS. Historically, early threaded interpreters (e.g., in FORTH or FORTRAN contexts) used simple direct function calls and hardware stack manipulation, without continuations. CPS, by contrast, explicitly passes continuations and avoids reliance on hardware stack state—making it more amenable to modern compiler optimizations and safe execution models.

5. Compiler Pipeline Overview

Let's say that we start off with the expression: `42 fibonacci`.

5.1. Source Code to Zag AST

The Zag Abstract Syntax Tree is semantically equivalent to Smalltalk source, but simpler. It contains Method and BlockClosure objects, both of which have bodies composed of a sequence of expressions, optionally terminated with a return. Expressions include: variable reference, variable assignment, send/cascade, literals (including Array literals), self, super, and thisContext.

The Zag AST can be generated directly from source code, and this is eventually how it will normally be created, but as this experiment is based from within Pharo, it is more convenient to extract the Zag AST from Pharo's native AST for a method. Regardless of the source, or whether generating for a threaded or native target, the Z-AST is then transformed into basic blocks by the AST compiler as described in the following subsection. Figure 2 showcases the Zag AST for `Fibonacci`.

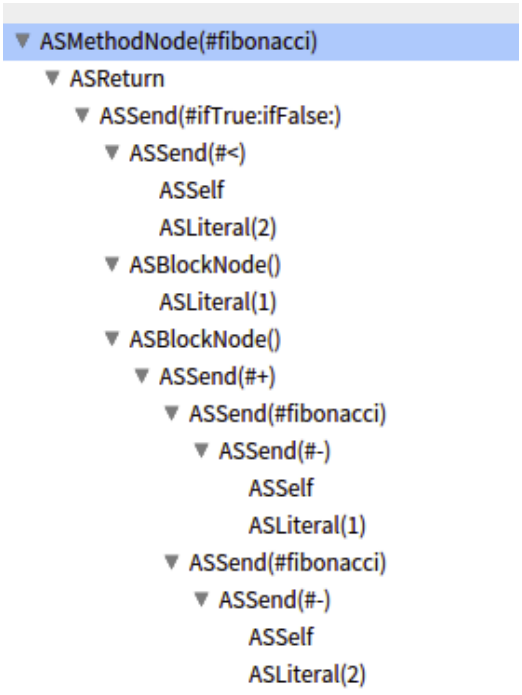

Figure 2: Zag AST for `fibonacci`

5.2. Image Exporting

The image exporter works in two phases. First, it expects a message send, which is `fibonacci` in our case. In order to execute this send, the image exporter needs to work in two phases to create an image file specific to this input.

5.2.1. Phase 01

In the first pass, the image exporter takes all the classes that are needed to compile our Zag ASTs to threaded code - basically getting hold of all the classes part of the Zag AST to threaded code compiler. It will then bootstrap itself by compiling these classes into threaded code. This chunk of the compiler now in threaded code is separately loaded into the Image being generated. We need the compiler in threaded code form because we need to compile code dynamically (because of the way the execution model is described in section 3.1 we need to be able to translate on the fly AST-> threaded and that requires us to have the compiler). [Explain why this phase must come before the second]

5.2.2. Phase 02

In the second phase, the image exporter will track and store all the classes and methods necessary to execute the input `42 fibonacci`. For example, classes and methods may belong to the `SmallInteger` class and its parent classes, amongst many others. The image exporter will take these classes and methods, convert them all into Zag AST form (described in section 3.1), and then organize them into a class table to include into the Zag image we are currently generating. Any objects that are allocated (e.g, class tables) are added to the .heap file that is part of the image.

5.3. Loading into the Zag Runtime

When we export the image, we need to load it into the Zag runtime. The image is then read in by Zag and all the symbols and dispatch tables and all the classes represented in AST format are loaded at the proper addresses for them to be used by the runtime.

5.4. Starting ThreadedCode + Dispatch

To commence execution, the runtime looks up in the dispatch tables for a compiled method for the symbol (selector's value) for the class of the object stored in the target field. If there is no entry for that specific selector, then the runtime begins executing the threaded compiler code (which was imported in). The compiler reaches into the class table and takes the AST form of the desired method, and converts it into a compiled method (threaded/LLVM form). It then patches the dispatch table to include the address of this newly compiled method. And it goes on executing the instructions within the compiled method, repeating this dispatch process as required.

```
1  &embedded.verifySelector,
2  ":recurse",
3  &embedded.dup, // self
4  &embedded.pushLiteral, Object.from(2),
5  &embedded.p5, // <=
6  &embedded.ifFalse,"label3",
7  &embedded.drop, // self
8  &embedded.pushLiteral1,
9  &embedded.returnNoContext,
```

Listing 9: Threaded Code (Part 1)

```
10 ":label3",
11 &embedded.pushContext,"^",
12 &embedded.pushLocal0, // self
13 &embedded.pushLiteral1,
14 &embedded.p2, // -
15 &embedded.callRecursive, "recurse",
16 &embedded.pushLocal0, //self
17 &embedded.pushLiteral2,
18 &embedded.p2, // -
19 &embedded.callRecursive, "recurse",
20 &embedded.p1, // +
21 &embedded.returnTop,
```

Listing 10: Threaded Code (Part 2)

Figure 9&10. Threaded code implementation of the Fibonacci algorithm, split across two columns for readability.

5.5. JIT Compiling Methods

Section 4.2 discusses the various fields that a `compiledMethod` contains - notiable the `executeFn` and `jitted` field. If the runtime detects that the method has been invoked multiple times (e.g., through recursion or loops), it updates `executeFn` to point to the `jitted` function, allowing execution to proceed from the optimized machine code instead.

6. Integrating LLVM

Zag integrates the LLVM compiler infrastructure to enable native code generation at runtime through Just-In-Time (JIT) compilation. LLVM is primarily written in C++ and provides bindings for various front-end languages. Since the Zag runtime is implemented in Zig, which interoperates seamlessly with C, we use the LLVM-C API for integration. To facilitate this, we rely on the open-source project `llvm-zig` (https://github.com/kassane/llvm-zig), which offers Zig bindings that expose the full LLVM-C interface.

Although the Zig bindings are thin wrappers over the C API, they are expressive enough to build complete IR modules, manage contexts, and orchestrate LLVM's JIT execution engine from within the runtime. These bindings rely on the developer having the LLVM binaries installed locally, as Zig inspects the installed version to extract the available function signatures and generate Zig-compatible interfaces. This allows us to construct LLVM IR modules directly from the runtime, emit native code, and obtain executable function pointers compatible with our execution model.

6.1. JIT Execution Model

To support JIT compilation, all LLVM-generated functions must conform to the `threadedFn` calling convention used by Zag. This means that each LLVM function must accept the same parameters as a threaded code block: the program counter `pc`, stack pointer `sp`, `process`, and context. This uniformity ensures that the compiled LLVM code can be invoked natively via a function `pointer` and integrate smoothly with the rest of the runtime. Furthermore, each LLVM function ends with a tail call to the next LLVM function, passing along these continuation parameters. This fits naturally with the CPS-style execution model employed in Zag, in which control is passed explicitly rather than relying on the native call stack.

To make this integration possible, Zag uses LLVM's ORC JIT API, which provides a flexible framework for compiling and executing LLVM IR at runtime. The JIT pipeline begins by creating a new LLVM context and module using functions like `LLVMModuleCreateWithName` and `LLVMContextCreate`. These objects form the container for all IR definitions. Next, a builder is created via `LLVMCreateBuilder`, which emits instructions into basic blocks within LLVM functions. The function that will represent the Smalltalk method that contains the IR code is declared using `LLVMAddFunction`, with its type defined using `LLVMFunctionType` to match the `threadedFn` signature.

Once the IR has been fully constructed, the module must be compiled using the ORC JIT infrastructure. A JIT instance is initialized through `LLVMOrcCreateLLJITBuilder` and `LLVMOrcCreateLLJIT`. The IR module is then wrapped into a thread-safe module using `LLVMOrcCreateNewThreadSafeModule`, and added to the JIT using `LLVMOrcLLJITAddLLVMIRModule`. Finally, the compiled function can be looked up via `LLVMOrcLLJITLookup`, returning an address that can be cast to a native Zig function pointer using (@ptrFromInt).

Listing 11 showcases a textual dump of a dummy LLVM IR function that adheres to the threadead calling convention by passing in the same threaded parameters along with the same return type of `sp`. Note that the return here at the end is simply a direct one instead of a tail call in LLVM (`musttail`) for simplicity. While executing, the runtime would actually use a tail call to pass control to the next CPS block properly.

```
--- IR DUMP ---
; ModuleID = 'module'
source_filename = "module"

%TagObject = type <{ i3, i5, i56 }>

define ptr @pushLiteral(ptr %pc, ptr %sp, ptr %process, ptr %context, i64 %signature) {
entry:
  %newSp = getelementptr %TagObject, ptr %sp, i3 -1
  %object_val = load %TagObject, ptr %pc, align 1
  store %TagObject %object_val, ptr %newSp, align 1
  ; Suppose to be a tail call but is instead a direct return for simplicity
  ret ptr %newSp
}
--- END IR ---
```

Listing 11: Example LLVM IR Function Emitted

Note that Listing 15 in the appendix provides LLVM-C API source code illustrating how to generate a generic LLVM IR function, invoke the JIT compilation pipeline, and retrieve a function pointer to execute the compiled code natively using Zig.

6.2. Threaded code dependence

The generation of an LLVM function in Zag is tightly coupled to the method's existing threaded code implementation. Specifically, LLVM IR emission relies on traversing the already-compiled threaded code sequence, interpreting each threaded word one by one and emitting the corresponding LLVM instructions through our Zig-based LLVM-C API bindings.

For example, consider the method `returnALiteral`, which simply returns the value 42. Its corresponding threaded code might look like this:

```
&pushLiteral,
42,
&returnTop
```

Listing 12: Threaded code for returnAFunction

6.3. The LLVM driver

Now that we have the threaded code version of the returnALiteral function, we can begin generating a parallel LLVM function. This process is initiated by invoking the LLVM driver from within Zag, which sets up the necessary components for IR generation.

The first step is to create an instance of the ZagLLVMOutput class, which is responsible for orchestrating the emission of LLVM IR. This is done by sending the `initialize` message to the class, which allocates memory on the heap for the object using a heap header, followed by additional space for its instance variables and (any) local data.

During initialization, the following instance variables are set up:

1. `sp`, which represents the current runtime stack pointer.
2. `gen`, which is the IR generator object containing the LLVM builder, context, and module.
3. `driver`, the LLVM driver

Once initialized, the LLVM driver retrieves the first threaded opcode (e.g., pushLiteral) and sends it as a message to the `ZagLLVMOutput` instance. That instance then delegates to the gen object, which emits calls to LLVM primitives. If the implementation of any message send does not exist, the runtime will use the threaded code compiler to compile the corresponding method definition from its AST in the class table. The compiled method (a `compiledMethod` object) is then inserted into the dispatch table. Here's an example of what the pushLiteral method might look like:

```
pushLiteral
    | literal |
    literal := driver nextObject.
    sp := gen register: sp plus: -8.
    gen store: literal at: sp
```

Listing 13: The pushLiteral message implementation

In this method, the literal value (e.g., 42) is retrieved by sending the nextObject message to the driver. The stack pointer is then adjusted using the `@register:plus:` primitive, and the value is stored using `@store:at:`. These primitives correspond to LLVM builder function calls made through the LLVM-C API, accessed using our Zig bindings.

6.4. LLVM Primitive Calls

The previous section mentioned primitives like `@register:plus:`. Section 4.3 touch upon how these primitives are implemented within Zag. All LLVM primitives reside inside of the LLVM primitive module, and they are responsible for emitting IR builder calls when invoked. And example of a LLVM primitive implementation is shown below:

```
pub const @"register:plus:asName:" = struct {
    pub fn primitive(pc: PC, sp: SP, process: *Process, context: *Context, extra: Extra) Result {
        if (noLLVM) return @call(tailCall, Extra.primitiveFailed, .{ pc, sp, process, context,
            extra });
        const self = sp.at(4);
        const instVars = self.instVars();
        const builder = BuilderRef.asLLVM(instVars[0]) catch return @call(tailCall, Extra.
            primitiveFailed, .{ pc, sp, process, context, extra });
        const registerToModify = ValueRef.asLLVM(sp.third) catch return @call(tailCall, Extra.
            primitiveFailed, .{ pc, sp, process, context, extra });
        const offset = sp.next.to(i64);
        const name = sp.top.arrayAsSlice(u8) catch return @call(tailCall, Extra.primitiveFailed,
            .{ pc, sp, process, context, extra });
        const newSp = sp.unreserve(3);
        const module = ModuleRef.asLLVM(instVars[1]) catch return @call(tailCall, Extra.
            primitiveFailed, .{ pc, sp, process, context, extra });
        const tagObjectTy = core.LLVMGetTypeByName(module, "TagObject");
        sp.top = ValueRef.asObject(singleIndexGEP(@ptrCast(builder), tagObjectTy, registerToModify
            , offset, name));
        return @call(tailCall, process.check(context.npc.f), .{ context.tpc, newSp, process,
            context, undefined });
    }
};
```

Listing 14: The LLVM `@register:plus:asName` primitive

The primitive `register:plus:asName:` is responsible for emitting a getelementptr (GEP) instruction in LLVM IR using the builder associated with the current generator instance. It operates on values already present on the stack, extracting the LLVM builder, the base register to offset, the offset amount, and the symbolic name to assign to the resulting value. These are used to compute a new pointer value via a single-index GEP, which is then stored back on the stack as a tagged object.

7. Future Work

This thesis represents the initial stages of integrating a JIT compiler into the Zag Smalltalk VM using LLVM. At this stage, the system lays the foundational infrastructure for JIT compilation, with a small but functional set of LLVM primitives that map Smalltalk message sends to IR builder instructions via Zig bindings. In the future, the JIT backend can be significantly expanded by implementing a broader range of LLVM primitives to support more language features and execution patterns.

Once the full system is operational and capable of compiling and executing a larger subset of Smalltalk programs, benchmark-driven evaluation can be conducted to assess the performance benefits of JIT-compiled code over traditional threaded execution. This would provide concrete data to inform further optimization and system design decisions. Additionally, future work may explore incorporating LLVM optimization passes that are tailored to dynamic languages.

8. Conclusion

This thesis set out to explore the effort involved in how LLVM could be integrated into a dynamic language runtime like Zag Smalltalk, and the work done so far shows that it's not only possible, but fits naturally with the system's design — once some foundational design decisions are made. The process of leveraging LLVM revealed both opportunities and constraints unique to dynamic languages. The challenge wasn't just generating LLVM IR, but figuring out how to embed that process within the runtime without disrupting it.

This led to the design of a custom LLVM driver that could interleave with Zag's existing execution model — one built around threaded functions and continuation-passing style. That meant thinking carefully about how execution context, stack layout, and method dispatch interact with native code generation. LLVM functions were deliberately constructed to match the threaded function signature, and follow the CPS execution technique. This design choice allowed the runtime to transition smoothly between threaded and JIT-executed code while maintaining the semantics and control flow expected by the rest of the system.

While the current implementation only covers a small number of primitives, it lays down a working foundation for a full JIT pipeline. Going forward, more LLVM primitives can be added, and once the system is robust enough to compile a meaningful subset of Smalltalk programs, performance testing can begin. This would allow us to evaluate the speedups gained from native code generation versus threaded execution.

Ultimately, this project shows that LLVM integration is viable, and that dynamic language VMs don't have to choose between flexibility and performance — with careful design, they can have both.

9. Appendix

```
const std = @import("std");
const llvm = @import("llvm");
const target = llvm.target;
```

```
const types = llvm.types;
const core = llvm.core;
const analysis = llvm.analysis;
const orc = llvm.orc;
const lljit = llvm.jit;
const engine = llvm.engine;
const target_machine = llvm.target_machine;

pub fn main() !void {

    // LLVM target specific setup logic
    target.initNativeTarget();

    // Create downward growing stack - smaller indices for more pushes
    var stack: [5]i64 = .{ 0, 0, 0, 4, 5 };
    std.debug.print("Stack before: {any}\n", .{stack});
    _ = &stack[3]; // top is at index 3 = value 4

    // Create LLVM module (context created by default)
    const module: types.LLVMModuleRef = core.LLVMModuleCreateWithName("module");
    const builder: types.LLVMBuilderRef = core.LLVMCreateBuilder();

    // Fill module with IR
    _ = try populateModule(module, builder);

    // Create the LLJIT Builder
    const jitBuilder = lljit.LLVMOrcCreateLLJITBuilder();

    // Create the LLJIT Instance
    var jit: types.LLVMOrcLLJITRef = undefined;
    if (lljit.LLVMOrcCreateLLJIT(&jit, jitBuilder) != null) {
        return error.LLJITCreationFailure;
    }

    // Add IR module to JIT instance
    const dylib = lljit.LLVMOrcLLJITGetMainJITDylib(jit);
    if (dylib == null) {
        return error.JITDylibRetrievalFailure;
    }
    const threadRef = orc.LLVMOrcCreateNewThreadSafeContext();
    const threadSafeModule = orc.LLVMOrcCreateNewThreadSafeModule(module, threadRef);

    if (lljit.LLVMOrcLLJITAddLLVMIRModule(jit, dylib, threadSafeModule) != null) {
        return error.AddModuleToJITFailure;
    }

    // Look up synbol to execute
    var result: orc.LLVMOrcExecutorAddress = 0;
    if (lljit.LLVMOrcLLJITLookup(jit, &result, "stack_add_top_two") != null) {
        return error.SymbolLookupFailure;
    }

    const llvmCompiledFn: *const fn (*i64) callconv(.C) *i64 = @ptrFromInt(result);
    const newSp = stackAddTopTwoFn(&stack[3]);
    const valueAtNewSp: *const i64 = @ptrCast(newSp);
    std.debug.print("Value at stack pointer: {}\n", .{valueAtNewSp.*});
}
```

Listing 15: LLVM Builder Calls

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] Zig Software Foundation, Zig programming language, https://ziglang.org/, 2025.

[3] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation, in: International symposium on code generation and optimization, 2004. CGO 2004., IEEE, San Jose, CA, USA, 2004, pp. 75–86.

[4] Sun Microsystems, Inc., The java hotspot performance engine architecture, 2001. URL: https://www.oracle.com/java/technologies/whitepaper.html#:~:text=Rather%20than%20compiling%20method%20by,optimizer%20on%20the%20hot%20spots.

[5] S. Ducasse, D. Pollet, A. Black, O. Nierstrasz, Pharo by Example 9, Square Bracket Associates, 2022. URL: https://books.pharo.org/pharo-by-example9/pdf/2022-03-26-index.pdf, accessed: 2025-04-27.

[6] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983. Chapter 26, "The Virtual Machine".

[7] D. A. Turner, Some principles of functional programming languages, Software: Practice and Experience 11 (1981) 671–681. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.4380110908. doi:10.1002/spe.4380110908.

[8] J. R. Bell, Threaded code, 1973. URL: https://dl.acm.org/doi/pdf/10.1145/362248.362270.

[9] A. W. Appel, T. Jim, Continuation-passing, closure-passing style, in: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89), Association for Computing Machinery, New York, NY, USA, 1989, pp. 293–302. URL: https://doi.org/10.1145/75277.75303. doi:10.1145/75277.75303.