Toronto Metropolitan University

esug EUROPEAN SMALLTALK USER GROUP

Smalltalk JIT Compilation

LLVM Experimentation

Janat Baig (Presenter) & Dave Mason

Contents

Intro to JIT Compilers

- There are two main implementation techniques: Compilation and Interpretation
- Compilers produce code to be executed by hardware - typically ahead of time
- Interpreters do this execution in software
- Just In Time (JIT) is a technique that can improve performance for interpreters by compiling to more efficient code dynamically at runtime
- JIT compiler generated code is typically 10x faster than interpretation

Basic Example

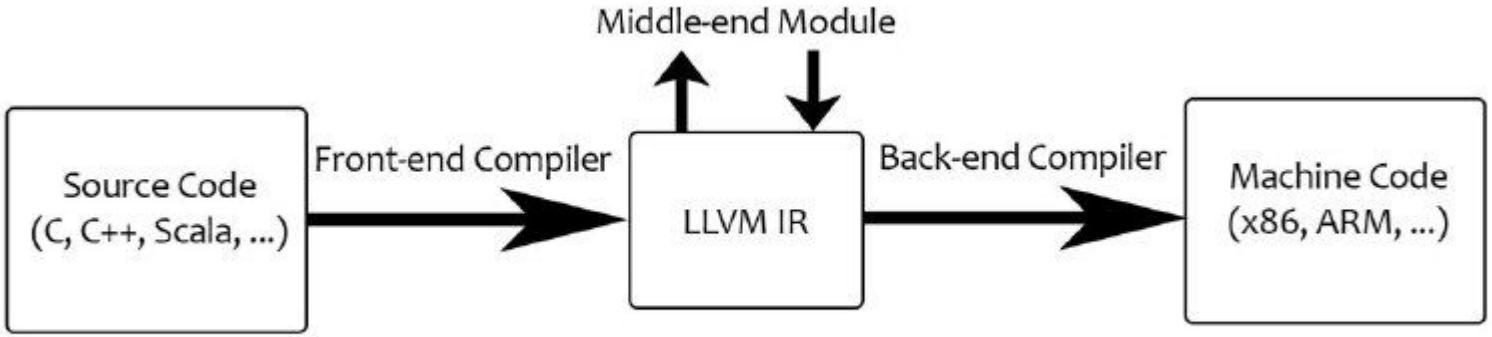
1. Method X is invoked multiple times
2. JIT labels method X as a hotspot
3. Compiles method X + applies optimizations from runtime information
4. Benefit? Don't have to scan, parse and interpreter method X every time it is invoked

JIT Compilers In Use

Used in lots of systems!

Intro to LLVM

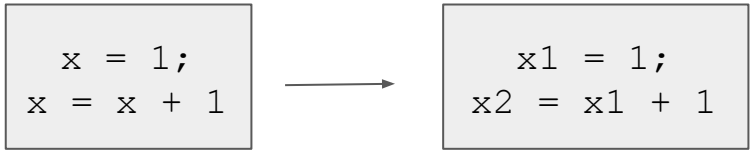
- LLVM is a toolset for building compilers
- Can generate machine code for a plethora of architectures through the LLVM compiler (LLC)
- LLC also performs a multitude of optimizations on it's IR
- LLVM intermediate representation (IR) is a register-based form that uses Single Static Assignment (SSA)



Single Static Assignment (SSA)

Since the LLVM IR uses SSA form, we need to ensure that we adhere to its restrictions:

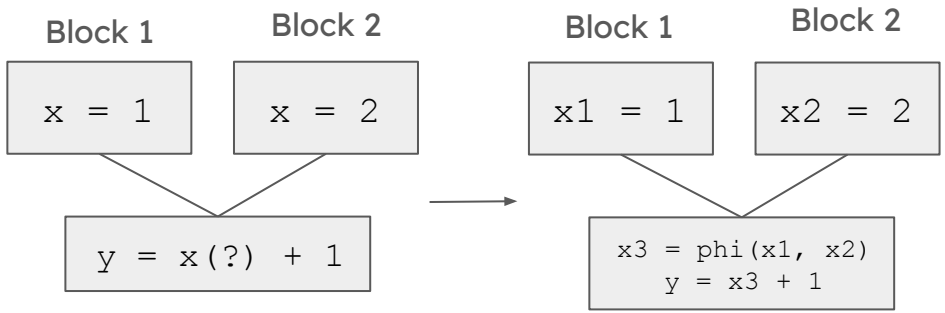
1. **Unique assignment #s**
2. Phi Node at converging paths



Single Static Assignment (SSA)

Since the LLVM IR uses SSA form, we need to ensure that we adhere to its restrictions:

1. Unique assignment #s
2. **Phi Node at converging paths**



Textual LLVM IR

LLVM has an binary API and a textual form.

Why are we experimenting with emitting textual IR?

1. Textual IR is easy to verify by eye

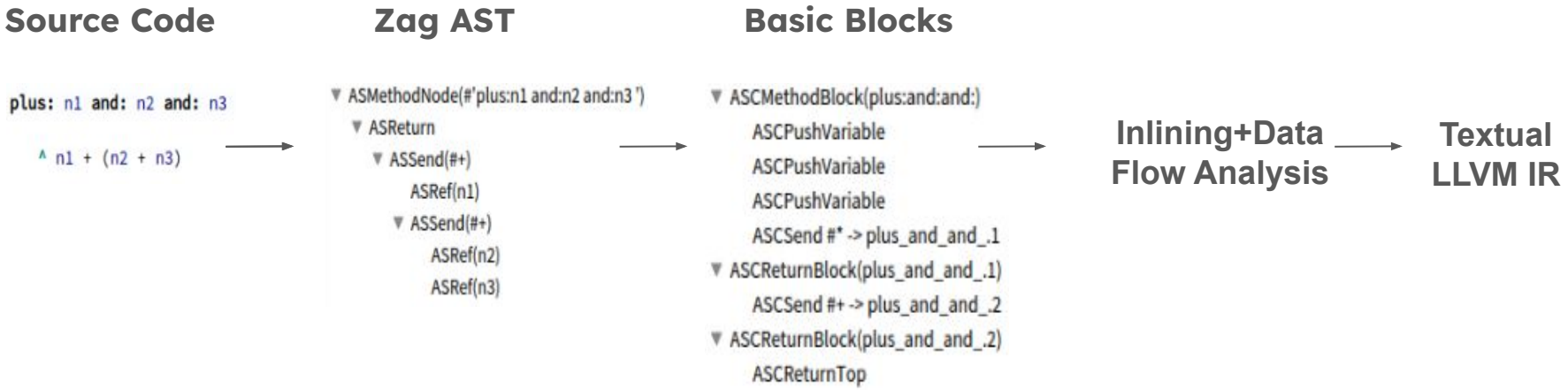
2. Debugging textual IR is simpler

3. Advantage of LLVM tooling for textual IR

Overall Compile Strategy

- convert AST to intermediate form
  - compile-time stack mimics run-time stack
  - sequence of basic blocks ending in send or return
- (optional) inlining
  - replaces some sends with embedded operations
  - target blocks become inline blocks
- for LLVM target
  - do flow analysis for SSA within ReturnBlocks
    - all values live in registers within ReturnBlocks
    - between ReturnBlocks only the stack/context is persistent

Steps to Emit LLVM IR



Generating Zag AST

Example 1, 1st Stage (Pharo hosted)

Source Code

```
plus: n1 and: n2 and: n3

    ^ n1 + (n2 + n3)
```

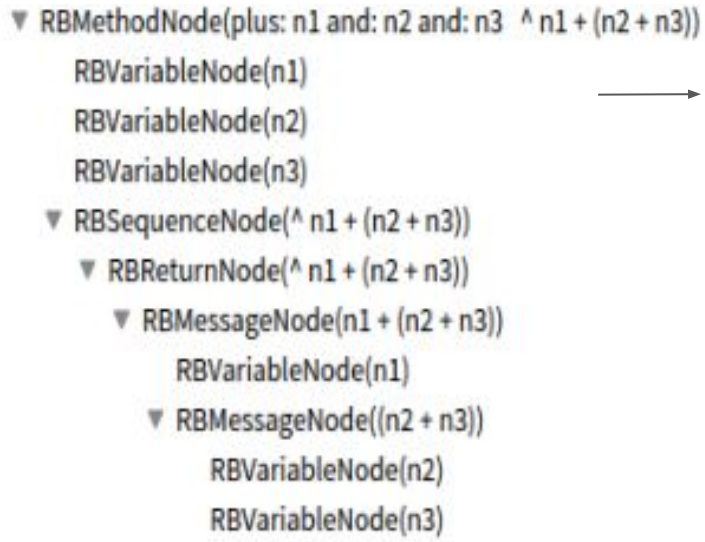

Pharo AST

- RBMethodNode(plus: n1 and: n2 and: n3 ^ n1 + (n2 + n3))
  - RBVariableNode(n1)
  - RBVariableNode(n2)
  - RBVariableNode(n3)
  - RBSequenceNode(^ n1 + (n2 + n3))
    - RBReturnNode(^ n1 + (n2 + n3))
      - RBMessageNode(n1 + (n2 + n3))
        - RBVariableNode(n1)
        - RBMessageNode((n2 + n3))
          - RBVariableNode(n2)
          - RBVariableNode(n3)

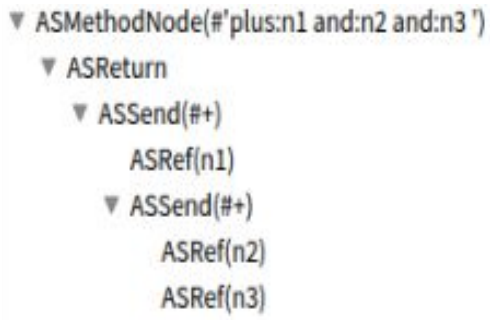
Generating the Zag AST (Cont.)

Example 1, 2nd Stage (Pharo hosted)

Pharo AST



Zag AST



Generating Basic Blocks

Example 1, 3rd Stage

Zag AST

- ASMethodNode(#'plus:n1 and:n2 and:n3 ')
  - ASReturn
    - ASSend(#+)
      - ASRef(n1)
      - ASSend(#+)
        - ASRef(n2)
        - ASRef(n3)

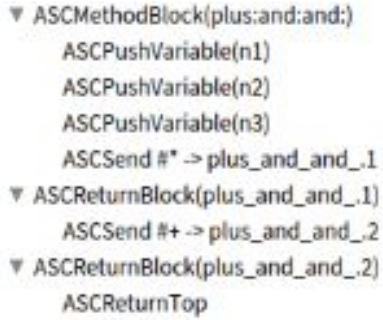
Basic Blocks

- ASCMethodBlock(plus:and:and:)
  - ASCPushVariable
  - ASCPushVariable
  - ASCPushVariable
  - ASCSend #* -> plus_and_and_.1
- ASCReturnBlock(plus_and_and_.1)
  - ASCSend #+ -> plus_and_and_.2
- ASCReturnBlock(plus_and_and_.2)
  - ASCReturnTop

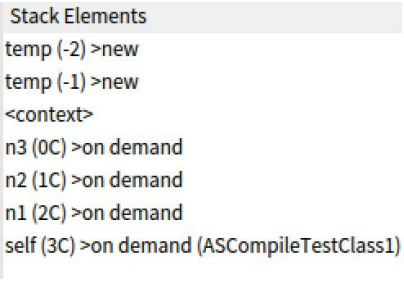
Flow Analysis

Example 1, 4th Stage

Basic Blocks



Inlining + Flow Analysis



Sample Stack

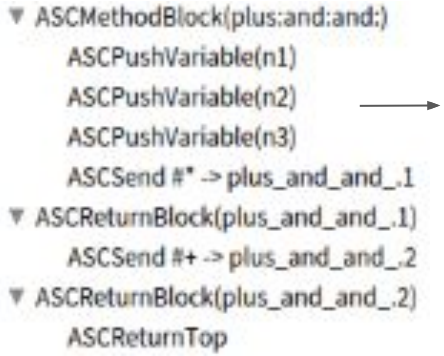
Data flow analysis

- On Demand: on the stack already (self parameters, read-only unless locals or temps)
- New: Not on stack, only spilled to stack when a call is invoked
- Phi: special case when the value came from 2 different paths (none in this ex.)

Generating Textual LLVM IR

Example 1, 5th Stage

Basic Blocks




```
define ptr @plus_and_and_(ptr noundef %.pc, ptr noundef %.sp, ptr noundef %.process, ptr
noundef %.context, i64 %.signature) #1 {
** push values and setup for send **
%30 = musttail call ptr %29(i64 %28, ptr nonnull align 8 %.sp.1, ptr nonnull align 8
%.process, ptr nonnull align 8 %.context.1, i64 %27)
ret ptr %30}
define ptr @plus_and_and_.1(ptr noundef %.pc, ptr noundef %.sp, ptr noundef %.process,
ptr noundef %.context, i64 %.signature) #1 {
** reset for next send **
%3 = musttail call ptr %2(i64 %1, ptr nonnull align 8 %.sp, ptr nonnull align 8
%.process, ptr nonnull align 8 %.context, i64 %4)
ret ptr %3}
define ptr @plus_and_and_.2(ptr noundef %.pc, ptr noundef %.sp, ptr noundef %.process,
ptr noundef %.context, i64 %.signature) #1 {
** setup for return **
%5 = musttail call ptr %2(i64 %3, ptr nonnull align 8 %.sp, ptr nonnull align 8
%.process, ptr nonnull align 8 %.context, i64 undef)
ret ptr %5}
```

Generating Textual LLVM IR

Example 1, 4th Stage

Basic Blocks

- ASCMethodBlock(plus:and:and:)
  - ASCPushVariable(n1)
  - ASCPushVariable(n2)
  - ASCPushVariable(n3)
  - ASCSend #* -> plus_and_and_.1
- ASCReturnBlock(plus_and_and_.1)
  - ASCSend #+ -> plus_and_and_.2
- ASCReturnBlock(plus_and_and_.2)
  - ASCReturnTop

```
define ptr @plus_and_and_(ptr noundef %.pc, ptr noundef %.sp, ptr noundef %.process, ptr noundef %.context, i64
%.signature) #1 {
%17 = getelementptr inbounds %zag.execute.Stack, ptr %.sp, i64 0, i32 2
%n1 = load i64, ptr %17, align 8 ; n1
%19 = getelementptr inbounds %zag.execute.Stack, ptr %.sp, i64 0, i32 1
%n2 = load i64, ptr %19, align 8 ; n2
%n3 = load i64, ptr %.sp, align 8 ; n3
; ** create context pointed by %.context.1 **
; make space to spill to memory
%.sp.1 = getelementptr inbounds %zag.execute.Stack, ptr %context.1, i64 0, i32 -3
%22 = getelementptr inbounds %zag.execute.Stack, ptr %.sp.1, i64 0, i32 2
store i64 %n1, ptr %22, align 8 ; n1 in reserved space
%23 = getelementptr inbounds %zag.execute.Stack, ptr %.sp.1, i64 0, i32 1
store i64 %n2, ptr %23, align 8 ; n2 in reserved space
store i64 %n3, ptr %sp.1, align 8 ; n3 in reserved space
; ** get native return address (@plus_and_and_.1) into %24 **
%25 = getelementptr inbounds %zag.context.Code, ptr %.tpc, i64 0, i32 1 ; threaded return into %25
%26 = getelementptr inbounds %zag.context.Context, ptr %context.1, i64 0, i32 3
store ptr %24, ptr %26, align 8 ; native PC
%27 = getelementptr inbounds %zag.context.Context, ptr %context.1, i64 0, i32 2
store ptr %25, ptr %27, align 8 ; threaded PC
; ** set %28 to the dispatch send address **
; ** set %29 to the symbol value for #* **
%30 = musttail call ptr %28(ptr %25, ptr nonnull align 8 %.sp.1, ptr nonnull align 8 %.process, ptr nonnull align
8 %.context.1, i64 %29)
ret ptr %30}
```

Stack Elements
n3 (0C) >on demand
n2 (1C) >on demand
n1 (2C) >on demand
self (3C) >on demand (ASCompileTestClass1)

Initial Stack

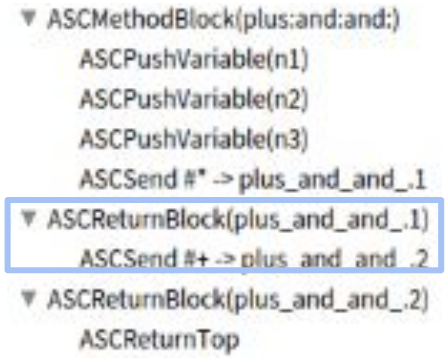
Stack Elements
temp (-7) >new
temp (-6) >new
temp (-5) >new
n3 (0C) >on demand
n2 (1C) >on demand
n1 (2C) >on demand
self (3C) >on demand (ASCompileTestClass1)

Before Send

Generating Textual LLVM IR

Example 1, 4th Stage

Basic Blocks

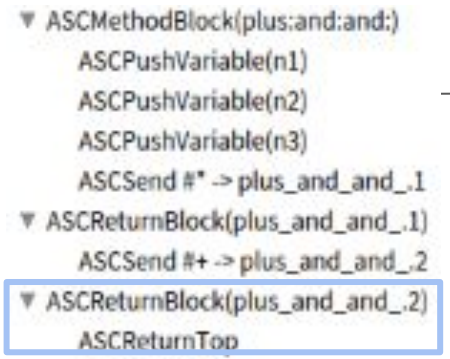



```
define ptr @plus_and_and_.1(ptr noundef %.tpc, ptr noundef %.sp, ptr noundef
%.process, ptr noundef %.context, i64 %.signature) #1 {
; ** get native return address (@plus_and_and_.2) into %1 **
%2 = getelementptr inbounds %zag.context.Code, ptr %.tpc, i64 0, i32 1
%3 = getelementptr inbounds %zag.context.Context, ptr %.context, i64 0, i32 3
store ptr %1, ptr %3, align 8 ; native PC
%4 = getelementptr inbounds %zag.context.Context, ptr %.context, i64 0, i32 2
store ptr %2, ptr %4, align 8 ; threaded PC
; ** set %5 to the dispatch send address **
; ** set %6 to the symbol value for #+ **
%8 = musttail call ptr %5(ptr undef, ptr nonnull align 8 %.sp, ptr nonnull align
8 %.process, ptr nonnull align 8 %.context, i64 %6)
ret ptr %7}
```

Generating Textual LLVM IR

Example 1, 4th Stage

Basic Blocks




```
define ptr @plus_and_and_.2(ptr noundef %.pc, ptr noundef %.sp, ptr noundef %.process, ptr
noundef %.context, i64 %.signature) #1 {
%1 = getelementptr inbounds %zag.context.Context, ptr %.context, i64 0, i32 3
%2 = load ptr, ptr %1, align 8 ; native PC
%3 = getelementptr inbounds %zag.context.Context, ptr %.context, i64 0, i32 2
%4 = load ptr, ptr %3, align 8 ; threaded PC
%5 = getelementptr inbounds %zag.context.Context, ptr %.context, i64 0, i32 4
%.ccontext = load ptr, ptr %5, align 8 ; caller's context
%6 = load i64, ptr %.sp, align 8 ; result value
%.sp.1 = getelementptr inbounds %zag.context.Context, ptr %.context, i64 0, i32 9
store i64 %6, ptr %.sp.1, align 8 ; result value
%7 = musttail call ptr %2(ptr %3, ptr nonnull align 8 %.sp.1, ptr nonnull align 8 %.process
ptr nonnull align 8 %.ccontext, i64 undef)
ret ptr %7}
```

Stack Elements
temp (-5) >new
n3 (0C) >on demand
n2 (1C) >on demand
n1 (2C) >on demand
self (3C) >on demand (ASCompileTestClass1)

Stack Elements
temp (-1) >on demand (ASCompileTestClass1)

Result of computation on the top of the stack

Future Work

1. Connecting the LLVM IR Builder into the runtime and generating the JIT'ed code on the fly

2. Benchmarking results

Thank you for listening