Automatically Retrofitting JIT Compilers

Laurence Tratt
https://tratt.net/laurie/

2026-03-18

yk

```
$ cd yklua && git diff 6c992afd -- src | diffstat -f 0
 Makefile  |   17       16 +    1 -     0 !
 ldo.c     |   60       49 +   11 -     0 !
 lfunc.c   |   28       28 +    0 -     0 !
 lfunc.h   |    7        7 +    0 -     0 !
 lgc.c     |   11       11 +    0 -     0 !
 lmem.c    |    6        6 +    0 -     0 !
 lobject.h |   18       18 +    0 -     0 !
 loslib.c  |   15       15 +    0 -     0 !
 lparser.c |  114      114 +    0 -     0 !
 lparser.h |    3        3 +    0 -     0 !
 lstate.c  |    6        6 +    0 -     0 !
 lstate.h  |    7        7 +    0 -     0 !
 ltable.c  |   18       11 +    7 -     0 !
 luac.c    |   11       11 +    0 -     0 !
 luaconf.h |    2        1 +    1 -     0 !
 lundump.c |   10       10 +    0 -     0 !
 lvm.c     |   95       90 +    5 -     0 !
 17 files changed, 403 insertions(+), 25 deletions(-)
```

- *VM*:

- *VM*: System containing one or more language implementations

- *VM*: System containing one or more language implementations
- *Interpreter*:

- *VM*: System containing one or more language implementations
- *Interpreter*: Simple language implementation

- *VM*: System containing one or more language implementations
- *Interpreter*: Simple language implementation*

*https://tratt.net/laurie/blog/2023/distinguishing_an_interpreter_from_a_compiler.html

- *VM*: System containing one or more language implementations
- *Interpreter*: Simple language implementation*
- *JIT compiler*:

*https://tratt.net/laurie/blog/2023/distinguishing_an_interpreter_from_a_compiler.html

- *VM*: System containing one or more language implementations
- *Interpreter*: Simple language implementation*
- *JIT compiler*: Language implementation that observes a running program, optimises, and compiles portions to machine code

*https://tratt.net/laurie/blog/2023/distinguishing_an_interpreter_from_a_compiler.html

I: Why?

My program is slow

Drop in a faster language implementation

Cinder; IronPython; Jython;

Cinder; IronPython; Jython;
Nuitka; Psyco; Pyjion;

Cinder; IronPython; Jython;
Nuitka; Psyco; Pyjion;
PyPy; Pyston; S6;

Cinder; IronPython; Jython;
Nuitka; Psyco; Pyjion;
PyPy; Pyston; S6;
Shed Skin; Stackless; Starkiller;

Cinder; IronPython; Jython;
Nuitka; Psyco; Pyjion;
PyPy; Pyston; S6;
Shed Skin; Stackless; Starkiller;
TrufflePython; Unladen Swallow;

Cinder; IronPython; Jython;
Nuitka; Psyco; Pyjion;
PyPy; Pyston; S6;
Shed Skin; Stackless; Starkiller;
TrufflePython; Unladen Swallow;
WPython; Zippy

JIT compiling VMs are:

JIT compiling VMs are:

hard

JIT compiling VMs are:

hard
expensive

JIT compiling VMs are:

hard
expensive
maybe incompatible

JIT compiling VMs are:

hard
expensive
maybe incompatible
difficult to evolve

Automation:

Automation: RPython and Truffle

Automation: RPython and Truffle

Host:

Automation: RPython and Truffle

Host: RPython / Java

Automation: RPython and Truffle

Host: RPython / Java

Guest:

Automation: RPython and Truffle

Host: RPython / Java
Guest: Lua, Ruby, Python, CPU sim, etc.

II: How?

The C interpreter is the source of truth

Generate a *meta-tracing* JIT compiler
from C interpreters

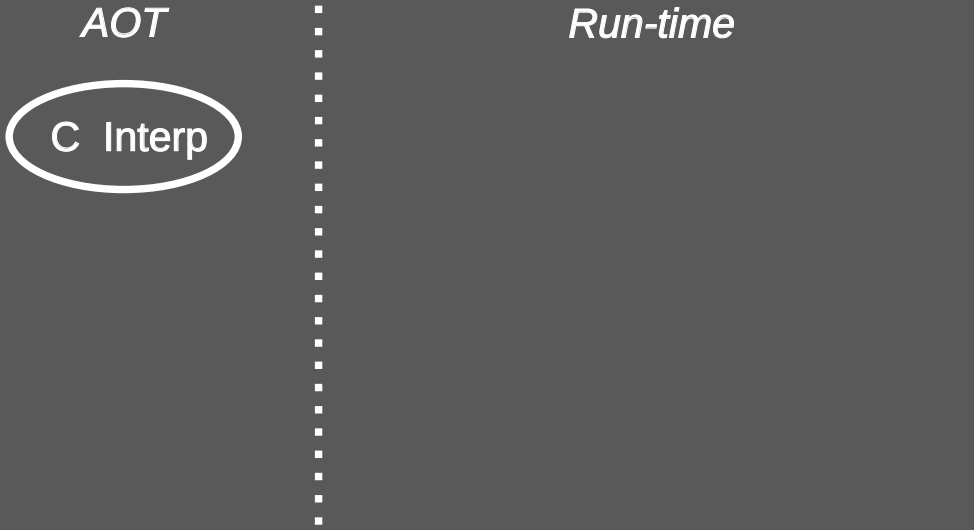
Tracing: manually record *hot loops* at run-time

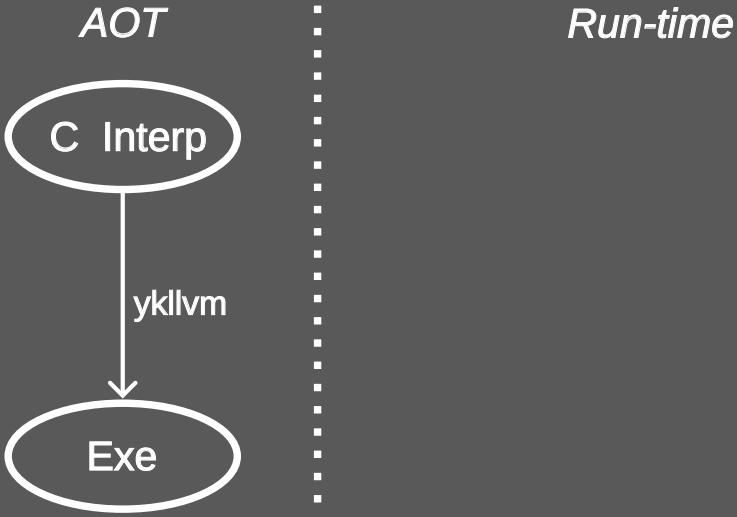
Tracing: manually record *hot loops* at run-time

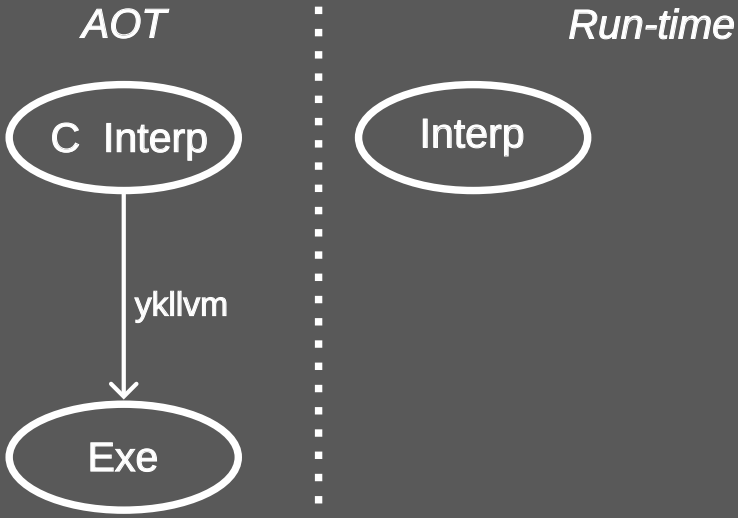
Meta-tracing: record the interpreter executing loops at run-time

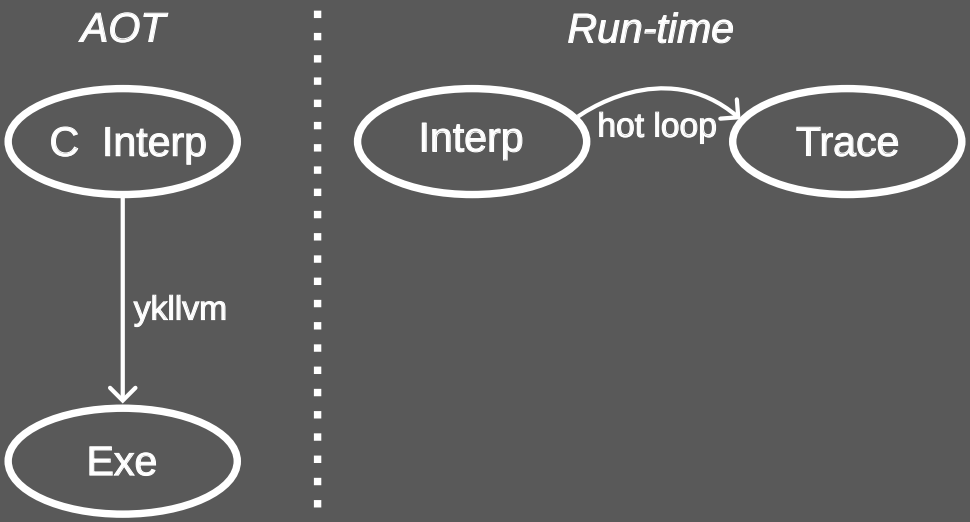
AOT

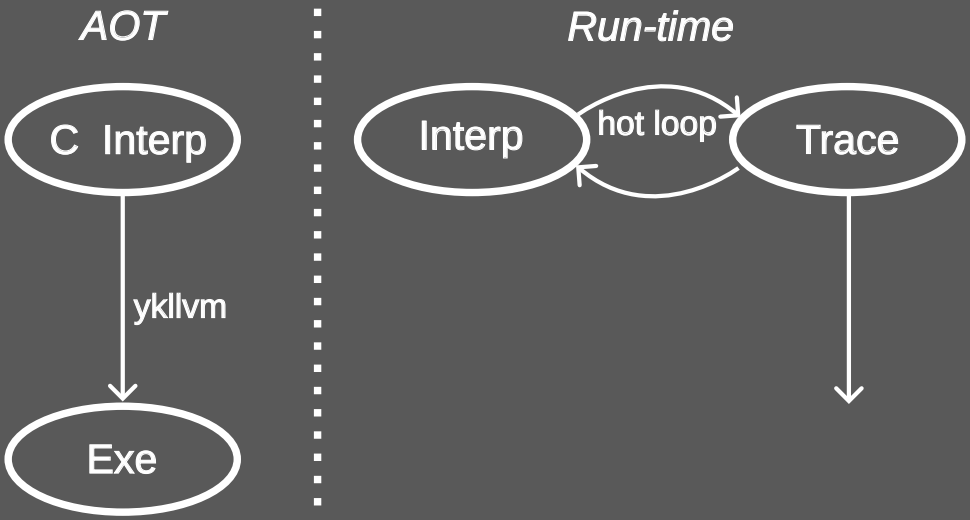
Run-time

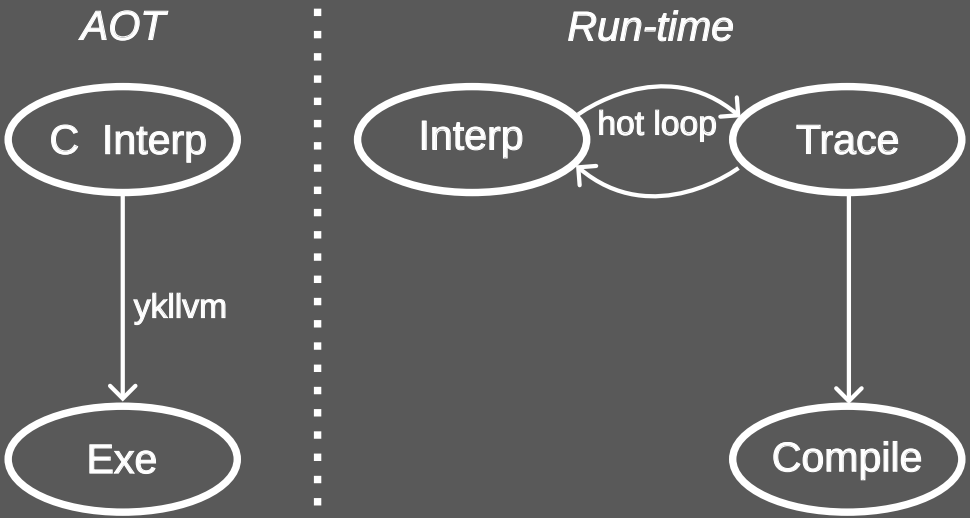

AOT
Run-time
C Interp

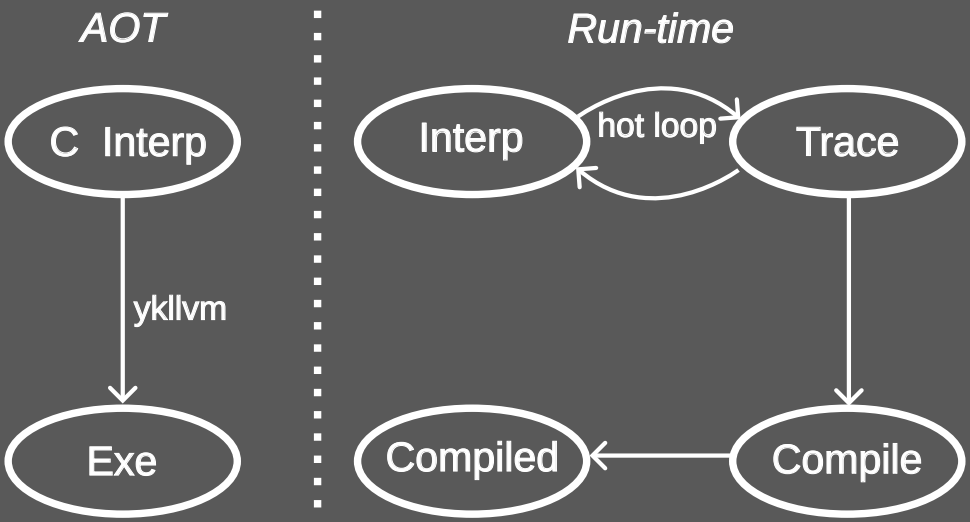

AOT
Run-time
C Interp
yklllvm
Exe

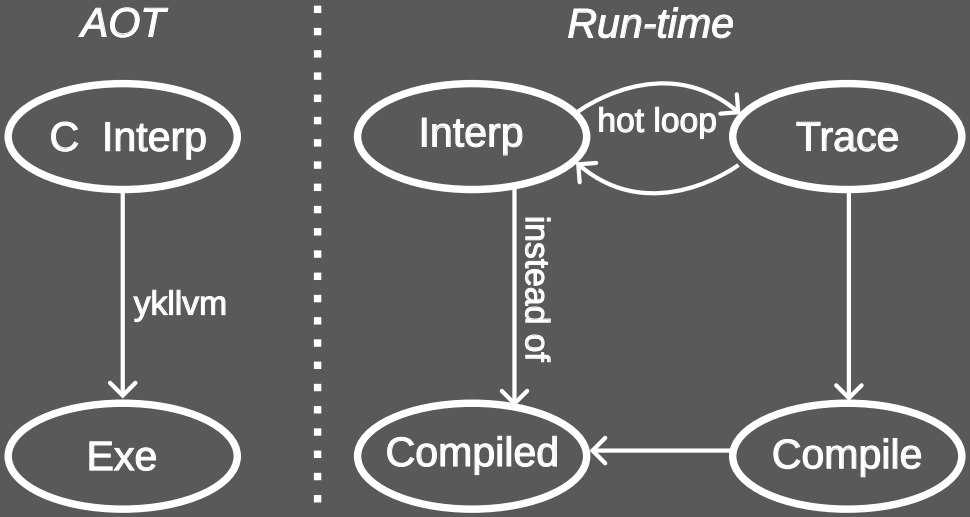

AOT
Run-time
C Interp
Interp
yklIvm
Exe

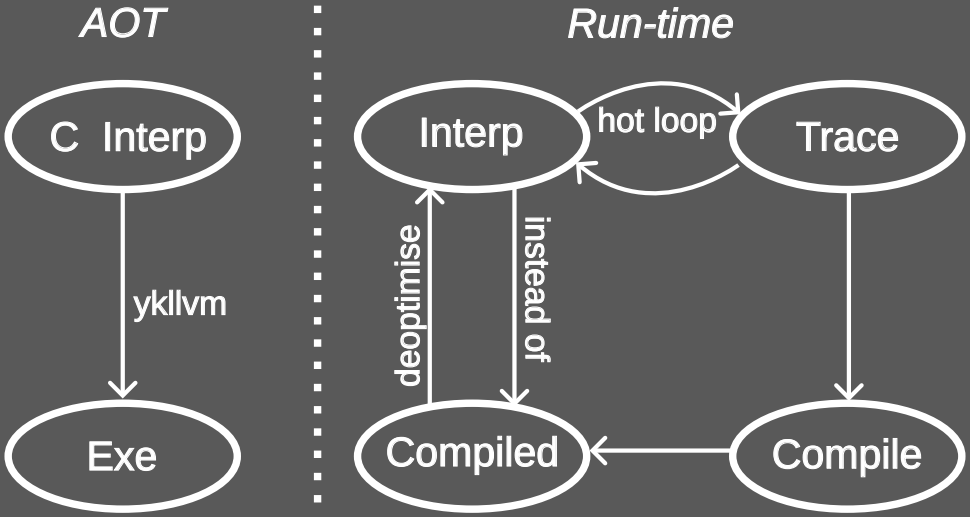

AOT
Run-time
C Interp
ykllvm
Exe
Interp
hot loop
Trace


AOT
Run-time
C Interp
yk|lvm
Exe
Interp
hot loop
Trace


AOT
Run-time
C Interp
ykllvm
Exe
Interp
hot loop
Trace
Compile


AOT
Run-time
C Interp
yklllvm
Exe
Interp
hot loop
Trace
Compiled
Compile


AOT
Run-time
C Interp
yklIvm
Exe
Interp
hot loop
Trace
instead of
Compiled
Compile


AOT
Run-time
C Interp
yk|lvm
Exe
Interp
hot loop
Trace
deoptimise
instead of
Compiled
Compile

```
while (true) {




}
```

```
Instruction *code = ...;

while (true) {




}
```

```
Instruction *code = ...;
int pc = 0;

while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {




  }
}
```

```
Instruction *code = ...;
int pc = 0;

while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP:




  }
}
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP: push(lookup(GET_OPVAL()));




  }
}
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP: push(lookup(GET_OPVAL())); pc++; break;




  }
}
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP: push(lookup(GET_OPVAL())); pc++; break;
    case OP_ADD:



  }
}
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP: push(lookup(GET_OPVAL())); pc++; break;
    case OP_ADD: push(pop() + pop()); pc++; break;



  }
}
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP: push(lookup(GET_OPVAL())); pc++; break;
    case OP_ADD: push(pop() + pop()); pc++; break;
    case OP_JLE:


  }
}
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP: push(lookup(GET_OPVAL())); pc++; break;
    case OP_ADD: push(pop() + pop()); pc++; break;
    case OP_JLE:
      if (pop() <= 0) pc += GET_OPVAL(i) else pc++;
      break;
  }
}
```

Guest

```
if x:
  y = y + 3
```

Guest

```
if x:
  y = y + 3
```

Tracing

```
OP_LOOKUP("x")
OP_JEQ(true, label)
OP_LOOKUP("y")
OP_INT(3)
OP_ADD
OP_SET("y")
```

Guest

```
if x:
  y = y + 3
```

Tracing

```
OP_LOOKUP("x")
OP_JEQ(true, label)
OP_LOOKUP("y")
OP_INT(3)
OP_ADD
OP_SET("y")
```

Meta-tracing

```
push(lookup("x")); pc++
guard true, pop(); pc++
push(lookup("y")); pc++
push(3); pc++
push(pop() + pop()); pc++
set("y", pop()); pc++
```

How does yk convert C code into a meta-tracing compiler?

How does yk convert C code into a meta-tracing compiler?

ykllvm

```
cc -o vm.o vm.c
ld -o vm vm.o
```

```
cc -o vm.o vm.c
ld -o vm vm.o

`yk-config release --cc` -o vm.o vm.c
`yk-config release --cc --ldflags` -o vm vm.o
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {

  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP:

      push(lookup(GET_OPVAL())); pc++; break;
    case OP_ADD:

      push(pop() + pop()); pc++; break;
  }
}
```

```
Instruction *code = ...;
int pc = 0;
Value **stack = ...;
while (true) {
  __yk_record(0);
  Instruction i = code[pc];
  switch (GET_OPCODE(i)) {
    case OP_LOOKUP:
      __yk_record(1);
      push(lookup(GET_OPVAL())); pc++; break;
    case OP_ADD:
      __yk_record(2);
      push(pop() + pop()); pc++; break;
  }
}
```

Convert LLVM’s IR for the interpreter into yk IR

```
359:                                              ; preds = %341
  %360 = load ptr, ptr %355, align 8, !tbaa !11
  %361 = load ptr, ptr %352, align 8, !tbaa !11
  %362 = call ptr @luaH_getshortstr(ptr noundef %361, ptr noundef %360) #18
  %363 = getelementptr inbounds nuw i8, ptr %362, i64
  %364 = load i8, ptr %363, align 8, !tbaa !9
  %365 = and i8 %364, 15
  %366 = icmp eq i8 %365, 0
  br i1 %366, label %371, label %367
367:                                              ; preds = %359
  %368 = load i64, ptr %362, align 8, !tbaa !11
  store i64 %368, ptr %345, align 8, !tbaa !11
  %369 = load i8, ptr %363, align 8, !tbaa !9
  %370 = getelementptr inbounds nuw i8, ptr %345, i64 8
  store i8 %369, ptr %370, align 8, !tbaa !9
  br label %3329
371:                                              ; preds = %341, %359
  %372 = phi ptr [ %362, %359 ], [ null, %341 ]
  store ptr %203, ptr %171, align 8, !tbaa !11
  %373 = load ptr, ptr %172, align 8, !tbaa !11
  store ptr %373, ptr %13, align 8, !tbaa !11
  call void @luaV_finishget(ptr noundef nonnull %0, ptr noundef nonnull %352, ptr n
  %374 = load volatile i32, ptr %173, align 8, !tbaa !11
  br label %3329
```

```
 bb47:
  %47_0: ptr = ptr_add %46_2, 8
  %47_1: i8 = load %47_0
  %47_2: i8 = and %47_1, 15i8
  %47_3: i1 = eq %47_2, 0i8
  condbr %47_3, bb49, bb48 [safepoint: 396i64, (%0_0, %0_2, %0_5, %0_6, ...
bb48:
  %48_0: i64 = load %46_2
  *%45_3 = %48_0
  %48_2: i8 = load %47_0
  %48_3: ptr = ptr_add %45_3, 8
  *%48_3 = %48_2
  br bb812
bb49:
  %49_0: ptr = phi bb47 -> %46_2, bb45 -> 0x0
  *%6_10 = %29_0
  %49_2: ptr = load %11_5
  *%0_21 = %49_2
  call luaV_finishget(%0_0, %45_10, %45_13, %45_3, %49_0) [safepoint: 397i64,
  br bb50
```

Executable = normal machine code + serialised IR

How does yk optimise a program?

How does yk optimise a program?

1. Inlining.

How does yk optimise a program?

1. Inlining.
2. Standard(ish) compiler optimisations.

How does yk optimise a program?

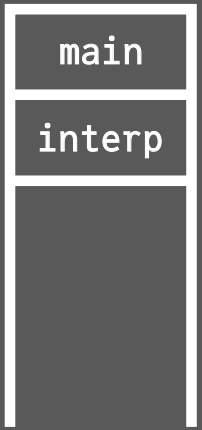
1. Inlining.
2. Standard(ish) compiler optimisations.
3. Interpreter hints.

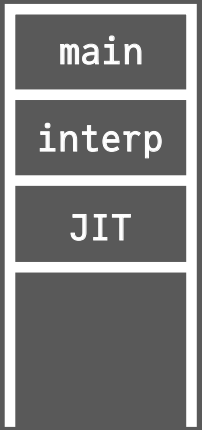
```
push(lookup("x")); pc++
guard true, pop(); pc++
push(lookup("y")); pc++
push(3); pc++
push(pop() + pop()); pc++
set("y", pop()); pc++
```

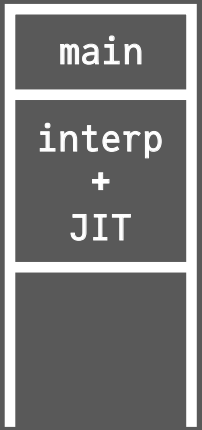
```
push(lookup("x")); pc++
guard true, pop(); pc++
push(lookup("y")); pc++
push(3); pc++
push(pop() + pop()); pc++
set("y", pop()); pc++
```

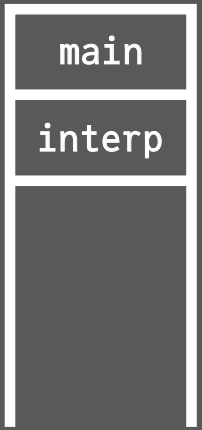
```
push(lookup("x")); pc++
guard true, false; pc++
push(lookup("y")); pc++
push(3); pc++
push(pop() + pop()); pc++
set("y", pop()); pc++
```


main


main
interp


main
interp
JIT


main
interp
+
JIT


main
interp

```
 bb47:
  %47_0: ptr = ptr_add %46_2, 8
  %47_1: i8 = load %47_0
  %47_2: i8 = and %47_1, 15i8
  %47_3: i1 = eq %47_2, 0i8
  condbr %47_3, bb49, bb48 [safepoint: 396i64, (%0_0, %0_2, %0_5, %0_6, ...
bb48:
  %48_0: i64 = load %46_2
  *%45_3 = %48_0
  %48_2: i8 = load %47_0
  %48_3: ptr = ptr_add %45_3, 8
  *%48_3 = %48_2
  br bb812
bb49:
  %49_0: ptr = phi bb47 -> %46_2, bb45 -> 0x0
  *%6_10 = %29_0
  %49_2: ptr = load %11_5
  *%0_21 = %49_2
  call luaV_finishget(%0_0, %45_10, %45_13, %45_3, %49_0) [safepoint: 397i64,
  br bb50
```

```
bb47:
  %47_0: ptr = ptr_add %46_2, 8
  %47_1: i8 = load %47_0
  %47_2: i8 = and %47_1, 15i8
  %47_3: i1 = eq %47_2, 0i8
  condbr %47_3, bb49, bb48 [safepoint: 396i64, (%0_0, %0_2, %0_5, %0_6, ...
bb48:
  %48_0: i64 = load %46_2
  *%45_3 = %48_0
  %48_2: i8 = load %47_0
  %48_3: ptr = ptr_add %45_3, 8
  *%48_3 = %48_2
  br bb812
bb49:
  %49_0: ptr = phi bb47 -> %46_2, bb45 -> 0x0
  *%6_10 = %29_0
  %49_2: ptr = load %11_5
  *%0_21 = %49_2
  call luaV_finishget(%0_0, %45_10, %45_13, %45_3, %49_0) [safepoint: 397i64,
  br bb50
```

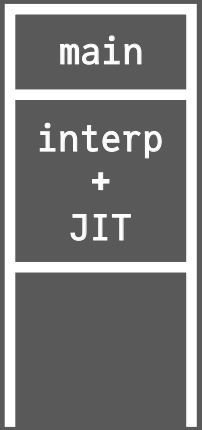
```
%0_0:
```

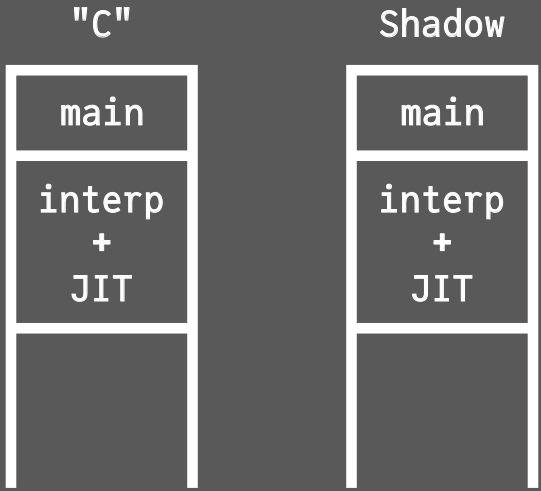
```
%0_0: [Stack(0x40)
```

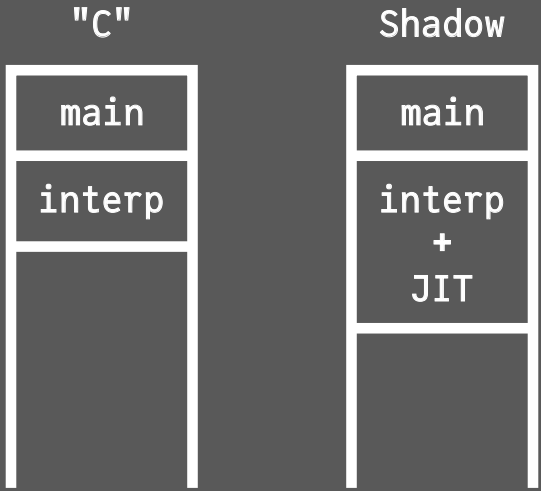
```
%0_0: [Stack(0x40), RAX]
```

```
%0_0: [Stack(0x40), RAX]
%0_1: [Const(9)]
```

```
%0_0: [Stack(0x40), RAX]
%0_1: [Const(9)]
%0_2: [RCX, RDX]
```


main
interp
+
JIT


"C"
main
interp
+
JIT
Shadow
main
interp
+
JIT


"C"
main
interp
Shadow
main
interp
+
JIT

```
while i > 0 do
  x = ...
  i = i - 1
end
print(x)
```

```
while i > 0 do
  x = ...
  i = i - 1
end
print(x)
```

Is this a loop?

```
while i > 0 do
  x = ...
  i = i - 1
end
print(x)
```

Is this a loop?

AOT:

```
while i > 0 do
  x = ...
  i = i - 1
end
print(x)
```

Is this a loop?

AOT: yes

```
while i > 0 do
  x = ...
  i = i - 1
end
print(x)
```

Is this a loop?

AOT: yes

Run-time:

```
while i > 0 do
  x = ...
  i = i - 1
end
print(x)
```

Is this a loop?

AOT: yes

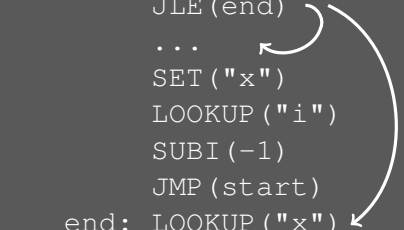
Run-time: maybe

```
        while i > 0 do
          x = ...
          i = i - 1
        end
        print(x)

start:  LOOKUP("i")
        JLE(end)
        ...
        SET("x")
        LOOKUP("i")
        SUBI(-1)
        JMP(start)
  end:  LOOKUP("x")
        PRINT()
```

```
while i > 0 do
  x = ...
  i = i - 1
end
print(x)
```

```
start: LOOKUP("i")
       JLE(end)
       ...
       SET("x")
       LOOKUP("i")
       SUBI(-1)
       JMP(start)
  end: LOOKUP("x")
       PRINT()
```

```
        while i > 0 do
          x = ...
          i = i - 1
        end
        print(x)

start:  LOOKUP("i")
        JLE(end)
        ...
        SET("x")
        LOOKUP("i")
        SUBI(-1)
        JMP(start)
  end:  LOOKUP("x")
        PRINT()
```



III: What’s next?

1. Support more LLVM IR (e.g. vectors).

1. Support more LLVM IR (e.g. vectors).
2. More optimisations (e.g. escape analysis).

1. Support more LLVM IR (e.g. vectors).
2. More optimisations (e.g. escape analysis).
3. More interpreters (WIP micropython).

https://github.com/ykjit/yk