How Can An Existing Language Implementation Be (semi-)Automatically Sped Up?

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

2022-04-21

Meta-trace existing interpreters
(e.g. CRuby)

Program

```
x = x + 1
if true
  x = x + 1
end
...
```

Program

```
x = x + 1
if true
  x = x + 1
end
...
```

```
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Program

```
x = x + 1
if true
  x = x + 1
end
...
```

```
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Program	Interpreter	Trace

```
x = x + 1
if true
  x = x + 1
end
...
```

```
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Program	Interpreter	Trace

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Program	Interpreter	Trace

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Program	Interpreter	Trace

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

```
instr = load_instr(0)
```

Program

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
```

Program

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
```

Program

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
```

Program

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
instr = load_instr(1)
guard(instr, INSTR_INT)
stack.push(1)
pc += 1
```

Program

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
instr = load_instr(1)
guard(instr, INSTR_INT)
stack.push(1)
pc += 1
...
```

Program

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
instr = load_instr(1)
guard(instr, INSTR_INT)
stack.push(1)
pc += 1
...
instr = load_instr(5)
guard(instr, INSTR_JMP_NOT_TRUE)
```

Program

```
x = x + 1
if true
  x = x + 1
end
...

VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
instr = load_instr(1)
guard(instr, INSTR_INT)
stack.push(1)
pc += 1
...
instr = load_instr(6)
guard(instr, INSTR_JMP_NOT_TRUE)
guard(!stack.pop(), false)
```

Program

```
x = x + 1
if true
  x = x + 1
end
...
```

```
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
instr = load_instr(1)
guard(instr, INSTR_INT)
stack.push(1)
pc += 1
...
instr = load_instr(6)
guard(instr, INSTR_JMP_NOT_TRUE)
guard(!stack.pop(), false)
pc += 1
```

Program

```
x = x + 1
if true
  x = x + 1
end
...
```

```
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
instr = load_instr(1)
guard(instr, INSTR_INT)
stack.push(1)
pc += 1
...
instr = load_instr(6)
guard(instr, INSTR_JMP_NOT_TRUE)
guard(!stack.pop(), false)
pc += 1
```

Program

```
x = x + 1
if true
  x = x + 1
end
...
```

```
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
VAR_GET(true)
JMP_NOT_TRUE(5)
VAR_GET(x)
INT(1)
ADD()
VAR_SET(x)
...
```

Interpreter

```
let pc = 0;
let stack = Vec::new();
let vars = HashMap::new();
loop {
  let instr = load_instr(pc);
  if instr == INSTR_VAR_GET {
    stack.push(
     vars[read_var_name_from_instr()]);
  } else if instr == INSTR_VAR_SET {
    vars[read_var_name_from_instr()]
     = stack.pop();
  } else if instr == INSTR_INT {
    stack.push(read_int_from_instr());
  } else if instr == INSTR_ADD {
    stack.push(stack.pop() + stack.pop());
  } else if instr == INSTR_JMP_NOT_TRUE {
    if !stack.pop() {
      pc += read_pc_from_instr();
      continue;
    }
  }
  pc += 1;
}
```

Trace

```
instr = load_instr(0)
guard(instr, INSTR_VAR_GET)
stack.push(vars["x"])
pc += 1
instr = load_instr(1)
guard(instr, INSTR_INT)
stack.push(1)
pc += 1
...
instr = load_instr(6)
guard(instr, INSTR_JMP_NOT_TRUE)
guard(!stack.pop(), false)
pc += 1
```

How can an existing language implementation be meta-traced?

How can an existing language implementation be meta-traced?

Meta-trace LLVM IR

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR

```
%0 = load i64, ...
%1 = load i64, ...
%2 = cmp i64 %1, i64 %2
guard(%2, ...)
%4 = add i64 %1, 2
```

How can meta-tracing be sped up?

How can meta-tracing be sped up?

Trace basic blocks

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR

```
bb1
bb3
```

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR

```
bb1
bb3
```

⇓

```
%0 = load i64, ...
%1 = load i64, ...
%2 = cmp i64 %1, i64 %2
guard(%2, ...)
%4 = add i64 %1, 2
```

How can basic block tracing be sped up?

How can basic block tracing be sped up?

Intel PT (or equivalent)

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR

```
0x9b8247a0200
0x9b8247a0300
```

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR + mapping

```
0x9b8247a0200
0x9b8247a0300
```

⇓

```
bb1
bb3
```

```
bb1:
  %0 = load i64, ...
  %1 = load i64, ...
  %2 = cmp i64 %1, i64 %2
  br %2, label %bb2, label %bb3

bb2:
  %3 = add i64 %1, 1
  ...

bb3:
  %4 = add i64 %1, 2
   ...
```

Executable = machine code + LLVM IR + mapping

```
0x9b8247a0200
0x9b8247a0300
```

⇓

```
bb1
bb3
```

⇓

```
%0 = load i64, ...
%1 = load i64, ...
%2 = cmp i64 %1, i64 %2
guard(%2, ...)
%4 = add i64 %1, 2
```

How can `guard`s be deoptimised?

How can `guard`s be deoptimised?

`llvm.experimental.deoptimize`

How can language level optimisations be expressed?

How can language level optimisations be expressed?

C/LLVM annotations

```
__attribute__((yk_outline)) void f(void) {
  ...
}
```

```
YkMT *mt = yk_mt_new();
YkLocation *yklocs = calloc(prog_len, sizeof(YkLocation));
for (YkLocation *ykloc = yklocs; ykloc < yklocs + prog_len; ykloc++)
  *ykloc = yk_location_new();

while (instr < prog_end) {
  YkLocation *loc = NULL;
  if (*instr == ']')
    loc = &yklocs[instr - prog];
  yk_mt_control_point(mt, loc);
  switch (*instr) {
    case '>': {
      if (cell++ == cells_end)
        errx(1, "out of memory");
        break;
      }
      case '<': {
        if (cell > cells)
          cell--;
        break;
      }
      ...
    }
  }
}
```

```
YkMT *mt = yk_mt_new();
YkLocation *ykblocs = calloc(prog_len, sizeof(YkLocation));
for (YkLocation *ykloc = ykblocs; ykloc < ykblocs + prog_len; ykloc++)
  *ykloc = yk_location_new();

while (instr < prog_end) {
  YkLocation *loc = NULL;
  if (*instr == ']')
    loc = &ykblocs[instr - prog];
  yk_mt_control_point(mt, loc);
  switch (*instr) {
    case '>': {
      if (cell++ == cells_end)
        errx(1, "out of memory");
        break;
      }
      case '<': {
        if (cell > cells)
          cell--;
        break;
      }
      ...
    }
  }
}
```

```
YkMT *mt = yk_mt_new();
YkLocation *yklocs = calloc(prog_len, sizeof(YkLocation));
for (YkLocation *ykloc = yklocs; ykloc < yklocs + prog_len; ykloc++)
  *ykloc = yk_location_new();

while (instr < prog_end) {
  YkLocation *loc = NULL;
  if (*instr == ']')
    loc = &yklocs[instr - prog];
  yk_mt_control_point(mt, loc);
  switch (*instr) {
    case '>': {
      if (cell++ == cells_end)
        errx(1, "out of memory");
        break;
      }
      case '<': {
        if (cell > cells)
          cell--;
        break;
      }
      ...
    }
  }
}
```

```
YkMT *mt = yk_mt_new();
YkLocation *yklocs = calloc(prog_len, sizeof(YkLocation));
for (YkLocation *ykloc = yklocs; ykloc < yklocs + prog_len; ykloc++)
  *ykloc = yk_location_new();

while (instr < prog_end) {
  YkLocation *loc = NULL;
  if (*instr == ']')
    loc = &yklocs[instr - prog]
  yk_mt_control_point(mt, loc);
  switch (*instr) {
    case '>': {
      if (cell++ == cells_end)
        errx(1, "out of memory");
        break;
      }
      case '<': {
        if (cell > cells)
          cell--;
        break;
      }
      ...
    }
  }
}
```

We want to use the strengths of existing systems (e.g. LLVM)...

We want to use the strengths of existing systems (e.g. LLVM)...

...but that also puts as at the mercy of their weaknesses.

https://github.com/ykjit/