Cheaply writing a fast interpreter

Code at https://github.com/ndmitchell/interpret

Neil Mitchell
@ndm_haskell
Given a language, we can:

- **Compile**
  - Static ASM code
  - Expensive
  - C, Rust

- **Interpret**
  - JIT/dynamic ASM
  - Cheap
  - Python
  - JavaScript
  - Java

The options: Interpret, Compile, Expensive, Cheap, Static ASM code, JIT/dynamic ASM, No ASM, cross-arch
Cheap interpreters

- Low cost of development and maintenance
- No Assembly (ASM) writing (may be some reading)
- Can do better! But at cost (v8, Lua)

An example: Starlark (aka deterministic Python)
- Used by Buck/Bazel build systems for config
- How would we go about writing an interpreter in Rust for Starlark?
Possible alternatives

- AST (abstract syntax tree) interpretation
- Bytecode (threaded?)
- Closure generation
- Intermediates: Native, Stack, Registers?
- Packed/Unpacked?
x = 100;
for (i = 1000; i != 0; i--) {
    x = x + 4 + x + 3;
    x = x + 2 + 4;
}
x

Deliberately use only +, to emphasise interpreter overhead
In reality, an expensive atoms might make all this noise
fn f(x: &Expr, vs: V) -> i64 {
  match x {
    Lit(i) => *i,
    Var(u) => vs[u],
    Add(x, y) =>
      f(x, vs) + f(y, vs),
    Assign(u, e) =>
      vs[u] = f(e, vs),
    ...
  }
}
What performance penalty?

Do the obvious things:
• Use unchecked array access
• Convert variables to indices
• No allocation
• Rust -O

(All these are always done in this talk)

What is the performance penalty?
What did it do?

\[
x = x + 4 + x + 3;
\]
\[
x = x + 2 + 4;
\]
\[
x = x + x + 13;
\]

Make add a noinline function call
More representative of real work
6.4x

6 minutes

1 minute
What does it do?

• Match on AST nodes
• Perform operations

Could we match on AST nodes only once?
• Yes! Generate closures once, run closures
• Closure = function pointer + data
Closures

type K = Box<dyn Fn(V) -> i64>;

fn f(x: &Expr) -> K {
    match x {
        Lit(i) => {
            let i = *i;
            box move |_| i;
        }
        Add(x, y) => {
            let x = f(x);
            let y = f(y);
            box move |v| x(v) + y(v)
        }
    }
}
Where do intermediates go?

With AST/Closure we reuse the native/Rust stack
\( f(x, ...) + f(y, ...) \)

What could we do instead? Explicit:

**Stack**
- Access the top
- PUSH 1
- ADD
- Pop top 2
- Push their sum

**Registers**
- Access by index
- \( r9 = 1 \)
- \( r7 = r2 + r9 \)
With a stack

```
PUSH  -1
GET   $i
ADD
SET   $i
```

```
loop {
    match tape.next() {
        PUSH => stack.push(tape.next()),
        ADD => stack.push(
            stack.pop() + stack.pop()),
        ...
    }
}
```
What happens on each op?

```rust
loop {
    match tape.next() {
        LOOKUP match[tape.next()]
        JUMP '
        ...
        BODY
    }
    JUMP 'loop
}
```
What would be optimal?

- Can’t generate new ASM on the fly
- The definition of a “Cheap” interpreter
- Must have a finite number of parameterisable chunks of ASM
- Must JUMP between them - but only one JUMP

Sometimes known as “direct threading”
static const Tape tape =
{&&push, 1, &&add, &&set, 8, ...};

push:
    stack.push(tape.next());
    goto tape.next();

add:
    stack.push(
        stack.pop() + stack.pop());
    goto tape.next();

set:
Rust

Faking computed Goto

- Tail calls are compiled to JUMP
- On x86_64, with -O
- Not guaranteed 😞 (can abstract it)
- But is compositional 😊

```rust
fn add(stack: Stack, tape: Tape) {
    stack.push(
        stack.pop() + stack.pop);
    let k = tape.next();
    k(stack, tape);
}
```
Even faster

Use registers

- Longer instructions, but fewer
- Less adjusting the stack

PUSH x
PUSH 1
ADD

r2 = 1
r3 = r1 + r2

5 words 3 instructions

3 + 4 words 1 + 1 instructions
What else?

Didn’t work

• Use compact tape instead of word-aligned
• A few percent slowdown
• A better register allocator (less registers)
• No difference on this particular benchmark

Would work

• Transform the code first (e.g. 2 + 4 => 6)
• Use “bigger” fragments (e.g. add3)
• Generate fresh assembly at runtime
Conclusion

- **6.4x penalty**
- **Lowest effort, cleanest code**

- **4.8x penalty**
- **More effort, but not *much* more**

- **1.4x penalty**
- **Requires register allocator**
- **Uses unsafe operations (register indexing)**
- **Much more effort, but much better result**