# Using and Misusing Someone Else's Language

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

2020-01-13

# Overview

# Overview

- Language designers curse...

# Overview

- Language designers curse...
- ...to meta-tracing...

# Overview

- Language designers curse...
- ...to meta-tracing...
- ...to using Rust for *Yk*.

Why?

# Why?

- Language implementation is arduous.

# Why?

- Language implementation is arduous.
- ‘Semantic mismatch’s are real – and common.

# Why?

- Language implementation is arduous.
- ‘Semantic mismatch’s are real – and common.
- Bad design or bad implementation?

# Why?

- Language implementation is arduous.
- ‘Semantic mismatch’s are real – and common.
- Bad design or bad implementation?
- ‘Good enough’ performance is good enough.

# Why?

- Language implementation is arduous.
- ‘Semantic mismatch’s are real – and common.
- Bad design or bad implementation?
- ‘Good enough’ performance is good enough.
- Let’s free language designers to experiment!

# Traditional VMs

# Hand-written interpreter

# Traditional VMs

Hand-written interpreter

& hand-written JIT.

# Hand-written interpreter

# Meta-tracing

Hand-written interpreter

& get a JIT for free.

# Meta-tracing JITs

## *FL* Interpreter

```
pc = 0; stack = []
vars = {...}
while True:
  if control_point(pc):
    continue
  instr = load_instruction(pc)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    pc += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    pc += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    pc += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    pc += 1
```

```
elif instr == INSTR_IF:
  result = stack.pop()
  if result == True:
    pc += 1
  else:
    pc +=
      read_jump_if_instruction()
elif instr == INSTR_ADD:
  lhs = stack.pop()
  rhs = stack.pop()
  if isinstance(lhs, int)
   and isinstance(rhs, int):
    stack.push(lhs + rhs)
  else: ...
  pc += 1
```

# Meta-tracing JITs

## *FL* Interpreter

```
pc = 0; stack = []
vars = {...}
while True:
  if control_point(pc):
    continue
  instr = load_instruction(pc)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    pc += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    pc += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    pc += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    pc += 1
```

# Meta-tracing JITs

## *FL* Interpreter

```
pc = 0; stack = []
vars = {...}
while True:
  if control_point(pc):
    continue
  instr = load_instruction(pc)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    pc += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    pc += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    pc += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    pc += 1
```

## User program (lang *FL*)

```
if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3
```

# Meta-tracing JITs

## *FL* Interpreter

```
pc = 0; stack = []
vars = {...}
while True:
  if control_point(pc):
    continue
  instr = load_instruction(pc)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    pc += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    pc += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    pc += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    pc += 1
```

## Initial trace

```
v0 = <pc>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
...
```

# What language?

# What language?

- RPython?

# What language?

- RPython?
- C/C++?

# What language?

- RPython?
- C/C++?
- Java?

# What language?

- RPython?
- C/C++?
- Java?
- Rust!

# Rust for meta-tracing: pros

# Rust for meta-tracing: pros

- Good balance between performance and safety.

# Rust for meta-tracing: pros

- Good balance between performance and safety.
- Vibrant & growing community.

# Rust for meta-tracing: pros

- Good balance between performance and safety.
- Vibrant & growing community.
- High quality, modern, compiler.

# Rust for meta-tracing: pros

- Good balance between performance and safety.
- Vibrant & growing community.
- High quality, modern, compiler.
- Doesn’t impose an inappropriate GC strategy.

# Rust for meta-tracing: cons

# Rust for meta-tracing: cons

- Incomplete specification.

# Rust for meta-tracing: cons

- Incomplete specification.
- Upstream struggling with community growth?

# Rust for meta-tracing: cons

- Incomplete specification.
- Upstream struggling with community growth?
- Language big and still growing.

# Rust for meta-tracing: cons

- Incomplete specification.
- Upstream struggling with community growth?
- Language big and still growing.
- Internal compiler churn.

# Rust for meta-tracing: cons

- Incomplete specification.
- Upstream struggling with community growth?
- Language big and still growing.
- Internal compiler churn.
- Difficult to use for GC'd languages.

# Language spec

Language reference:

# Language spec

Language reference: “**Warning:** The following list is not exhaustive. There is no formal model of Rust’s semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. The following list is just what we know for sure is undefined behavior.”

# Language spec

Language reference: “**Warning:** The following list is not exhaustive. There is no formal model of Rust’s semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. The following list is just what we know for sure is undefined behavior.”

UnsafeCell:

# Language spec

Language reference: “**Warning:** The following list is not exhaustive. There is no formal model of Rust’s semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. The following list is just what we know for sure is undefined behavior.”

UnsafeCell: “The precise Rust aliasing rules are somewhat in flux, but the main points are not contentious”

# Language spec

# Language spec

Stacked Borrows is excellent work!

# Language spec

Stacked Borrows is excellent work!

1. Will it become official?

# Language spec

Stacked Borrows is excellent work!

1. Will it become official?
2. Will Miri be the only sanitiser?

# Garbage collection: outlines

# Garbage collection: outlines

- Implementing GC'd languages without GC: painful, slow.

# Garbage collection: outlines

- Implementing GC'd languages without GC: painful, slow.
- Rust once had a ref-counted GC!

# Garbage collection: outlines

- Implementing GC'd languages without GC: painful, slow.
- Rust once had a ref-counted GC!
- Many exotic Rust GC libraries.

# Garbage collection

# Garbage collection

- Unlikely we can retrofit ‘perfect’ GC.

# Garbage collection

- Unlikely we can retrofit ‘perfect’ GC.
- Conservative GC probably always neccessary.

# Garbage collection

- Unlikely we can retrofit ‘perfect’ GC.
- Conservative GC probably always neccessary.
- Our solution: rustgc.

# Garbage collection

- Unlikely we can retrofit ‘perfect’ GC.
- Conservative GC probably always neccessary.
- Our solution: rustgc.
- Boehm is underappreciated.

# Garbage collection

- Unlikely we can retrofit ‘perfect’ GC.
- Conservative GC probably always neccessary.
- Our solution: rustgc.
- Boehm is underappreciated.
- Fork rustc for semi-precise GC.

# Garbage collection

- Unlikely we can retrofit ‘perfect’ GC.
- Conservative GC probably always neccessary.
- Our solution: rustgc.
- Boehm is underappreciated.
- Fork rustc for semi-precise GC.
- Safe enough (probably…) with stacked borrows.

# Garbage collection

- Unlikely we can retrofit ‘perfect’ GC.
- Conservative GC probably always neccessary.
- Our solution: rustgc.
- Boehm is underappreciated.
- Fork rustc for semi-precise GC.
- Safe enough (probably…) with stacked borrows.
- Impose restrictions on finalisers for safety.

Yk

# *Yk*

- Our new Rust meta-tracer: *Yk*.

# *Yk*

- Our new Rust meta-tracer: *Yk*.
- Still ‘hello world’ days!

# *Yk*

- Our new Rust meta-tracer: *Yk*.
- Still ‘hello world’ days!
- Major aim: reduce warm-up.

# *Yk*

- Our new Rust meta-tracer: *Yk*.
- Still ‘hello world’ days!
- Major aim: reduce warm-up.
- Trace basic blocks; use Intel PT (or equivalent).

# *Yk*

- Our new Rust meta-tracer: *Yk*.
- Still ‘hello world’ days!
- Major aim: reduce warm-up.
- Trace basic blocks; use Intel PT (or equivalent).
- Combination of rustc fork and ‘normal’ libraries.

ykrustc

# ykrustc

- A(nother) rustc fork.

# ykrustc

- A(nother) rustc fork.
- rustc: HIR $\rightarrow$ MIR $\rightarrow$ machine code.

# ykrustc

- A(nother) rustc fork.
- rustc: HIR $\rightarrow$ MIR $\rightarrow$ machine code.
- ykrustc: HIR $\rightarrow$ MIR $\rightarrow$ machine code

# ykrustc

- A(nother) rustc fork.
- rustc: HIR $\rightarrow$ MIR $\rightarrow$ machine code.
- ykrustc: HIR $\rightarrow$ MIR $\rightarrow$ machine code
  compile-time: MIR $\rightarrow$ SIR

# ykrustc

- A(nother) rustc fork.
- rustc: HIR $\rightarrow$ MIR $\rightarrow$ machine code.
- ykrustc: HIR $\rightarrow$ MIR $\rightarrow$ machine code
  compile-time: MIR $\rightarrow$ SIR
  run-time: $\underbrace{\text{SIR} \rightarrow \text{TIR}}_{\text{JIT}} \rightarrow$ machine code.

# Interpreter loop

```
while {
    if    x < 0 {
           x =    y +    z;
    }
    print(   x);
}
```

# Interpreter loop

```
#[interp_step]
fn interp(st: &mut InterpCtx) {
    if st.x < 0 {
        st.x = st.y + st.z;
    }
    print(st.x);
}
```

# MIR

```
bb0: {
    StorageLive(_2);
    StorageLive(_3);
    _3 = ((*_1).0: isize);
    _2 = Lt(move _3, const 0_isize);
    StorageDead(_3);
    switchInt(_2) -> [false: bb1, otherwise: bb2];
}
bb1: {
    goto -> bb4;
}
bb2: {
    StorageLive(_4);
    _4 = ((*_1).1: isize);
    StorageLive(_5);
    _5 = ((*_1).2: isize);
    _6 = CheckedAdd(_4, _5);
    assert(!move (_6.1: bool), "attempt to compute `{} + {}`, which would overflow", move _4, move
}
bb3: {
    ((*_1).0: isize) = move (_6.0: isize);
    StorageDead(_5);
    StorageDead(_4);
    goto -> bb4;
}

...
```

# SIR

```
bb0:
  $2 = *$1
  $3 = $2 < 0isize
  $4 = $3
  switch_int $4, [0], [1], 2
bb1:
  goto bb4
bb2:
  $5 = *($1)+8
  $6 = *($1)+16
  $7 = $5 + $6 (checked)
  $8 = $7
  assert $8+8, 3
bb3:
  *$1 = $8
  goto bb4
bb4:
  $9 = *$1
  $10 = call print($9) -> bb5
bb5:
  $0 = ()
  return
```

# TIR (for args -1, 1, 2)

```
$2 = *$1
$3 = $2 < 0isize
dead($2)
$4 = $3
dead($3)
guard($4, other_integer([0]), <interp, 0>, [])
dead($4)
$5 = *($1)+8
$6 = *($1)+16
$7 = $5 + $6 (checked)
dead($5)
dead($6)
$8 = $7
dead($7)
guard($8+8, bool(false), <interp, 2>, [$8])
*$1 = $8
dead($8)
$9 = *$1
$10 = call(print, [$9])
dead($10)
dead($9)
```

# Status

# Status

- Roughly: ‘Hello world’ for BF.

# Status

- Roughly: ‘Hello world’ for BF.
- We’re now in the happy phase of implementation!

# Status

- Roughly: ‘Hello world’ for BF.
- We’re now in the happy phase of implementation!
- Next: SOM.

# Status

- Roughly: ‘Hello world’ for BF.
- We’re now in the happy phase of implementation!
- Next: SOM.
- Long-term: a ‘Rust language implementation kit’? Pair e.g. with our parsing toolkit.

## Thanks for listening

https://github.com/softdevteam/ykrustc/wiki