17-350 / 17-650

Lecture 03: Fast Interpreter Implementation

Jonathan Aldrich
Sep 1, 2026

Carnegie Mellon University

S3D Software and Societal Systems Department

Lecture Outline

- Recap: the safe-Rust interpreter improvements from Lecture 2
- Remaining overhead: instruction size and dispatch
- Safe dispatch alternatives, true bytecode and threading
- Tradeoffs between `unsafe` and nightly build features
- `unsafe` for a compact tagged value representation
- Extensions and resources

Lecture 2 Recap

- Tree-walker to a linear bytecode interpreter
  - `Vec<Instr>` instead of boxed AST nodes
  - Better cache locality and less repeated tree matching
- Name lookup to direct slot indexing
  - `HashMap<String, Value>` lookup becomes `locals[i]`
  - Runtime access is array indexing
- All in **safe Rust**

Safe Bytecode From Lecture 2

```
#[derive(Clone, Copy)]
enum Instr { Const(i32), Add, Halt, }
fn run(code: &[Instr]) -> Result<i32, &'static str> {
        let mut stack = Vec::new();
        let mut pc = 0;
        loop {
                let instr = *code.get(pc).ok_or("program counter out of bounds")?;
                match instr {
                        Instr::Const(value) => stack.push(value),
                        Instr::Add => {
                                let rhs = stack.pop().ok_or("stack underflow")?;
                                let lhs = stack.pop().ok_or("stack underflow")?;
                                stack.push(lhs.checked_add(rhs).ok_or("integer overflow")?);
                        }
                        Instr::Halt => return stack.pop().ok_or("empty stack"),
                }
                pc += 1;
        }
}
```

What Overhead Remains?

- Fixed-size instructions
  - `Const(i32)` might need 8 bytes after enum tag + padding
  - `Add` and `Halt` could be 1 byte each in principle
  - In an `enum` representation, every `Instr` has the same size
- Basic instruction dispatch
  - If LLVM cooperates, dense `match` can become a jump table
  - But Rust does not guarantee that lowering
- The loop jumps back to the dispatch head after every instruction

Match Dispatch: Best Case

Assume LLVM compiles the `match` into a jump table:

```
tag = load pc.tag              ; load instruction tag
pc  = pc + sizeof(Instr)       ; advance instruction pointer
off = tag * PTR_SIZE           ; compute jump-table offset
dst = load jump_table[off]     ; load target address
jmp dst                        ; indirect jump
```

- Roughly 5 RISC-like dispatch instructions (maybe more with bounds checking)
- Then each case jumps back to the loop head
- Good when it happens, but it is a backend optimization choice

Safe Alternative: Function Table

```
struct VM { stack: Vec<i32>, pc: usize }

enum Action { Continue, Halt(i32) }
type Handler = fn(&mut VM, &[u8]) -> Result<Action, &'static str>;

const CONST: u8 = 0;
const ADD: u8 = 1;
const HALT: u8 = 2;

static DISPATCH: [Handler; 3] = [op_const, op_add, op_halt];

fn run(code: &[u8]) -> Result<i32, &'static str> {
        let mut vm = VM { stack: Vec::new(), pc: 0 };
        loop {
                let opcode = *code.get(vm.pc).ok_or("pc out of bounds")?;
                vm.pc += 1;
                let handler = DISPATCH.get(opcode as usize).ok_or("bad opcode")?;
                match handler(&mut vm, code)? {
                        Action::Continue => {}
                        Action::Halt(value) => return Ok(value),
                }
        }
}
```

Function Handlers

```
fn op_const(vm: &mut VM, code: &[u8]) -> Result<Action, &'static str> {
        let bytes = code.get(vm.pc..vm.pc + 4).ok_or("truncated const")?;
        vm.pc += 4;
        vm.stack.push(i32::from_le_bytes(bytes.try_into().unwrap()));
        Ok(Action::Continue)
}

fn op_add(vm: &mut VM, _code: &[u8]) -> Result<Action, &'static str> {
        let rhs = vm.stack.pop().ok_or("stack underflow")?;
        let lhs = vm.stack.pop().ok_or("stack underflow")?;
        vm.stack.push(lhs.checked_add(rhs).ok_or("integer overflow")?);
        Ok(Action::Continue)
}

fn op_halt(vm: &mut VM, _code: &[u8]) -> Result<Action, &'static str> {
        Ok(Action::Halt(vm.stack.pop().ok_or("empty stack")?))
}
```

Function Table Tradeoffs

- Advantage: the table lookup is guaranteed by the source structure
  - No need to hope LLVM chooses a jump table for `match`
- Similar dispatch shape:
  i. load opcode
  ii. increment instruction address
  iii. compute table offset
  iv. load function pointer
  v. indirect call
- But the last step is a **call**, not a local jump
  - ABI boundary: arguments, possible spills, prologue/epilogue
  - Harder to keep VM state in registers across handlers
  - Less opportunity for inlining and cross-opcode optimization

Optimize Further: True Bytecode

- Store opcodes as bytes, not Rust enum values
- Each instruction starts with 1 byte
  - Operand data follows only when needed
  - `Add` and `Halt`: 1 byte
  - `Const(i32)`: 1 opcode byte + encoded immediate
- WebAssembly uses LEB128-style variable-length integers
  - 7 payload bits per byte + 1 continuation bit
  - Many `i32.const` values fit in 1 byte of payload
- Benefit: denser code, better instruction-cache locality

Optimize Further: Direct Threading

- Compile bytecode to a sequence of handler addresses
  - Instruction stream cell = address of the next handler
- Dispatch becomes conceptually:

```
dst = load ip        ; load handler address
ip  = ip + PTR_SIZE ; advance threaded-code pointer
jmp dst             ; jump directly to handler
```

- Roughly 3 RISC-like dispatch instructions
- No opcode load, no jump-table offset, no jump-table load
- Avoid jump to top of loop by inlining dispatch after each instruction
- BUT stable, safe Rust has no computed `goto` / labels-as-values

Direct Threading with Tail Calls (nightly compiler build)

```rust
#![feature(explicit_tail_calls)]

type Handler = fn(&mut VM, &[Handler]) -> !;

fn dispatch(vm: &mut VM, ip: &[Handler]) -> ! {
        let (handler, rest) = ip.split_first().expect("missing halt");
        become handler(vm, rest)
}

fn op_add(vm: &mut VM, ip: &[Handler]) -> ! {
        let rhs = vm.stack.pop().unwrap();
        let lhs = vm.stack.pop().unwrap();
        vm.stack.push(lhs + rhs);
        become dispatch(vm, ip)
}
```

- `become` asks for a tail call: no growing call stack
- Design goal (RFC #3407, still unmerged): if `become` compiles, a true tail call is guaranteed; if the target can't guarantee it, compilation fails - it should never silently degrade to a stack-growing call
- This is not first-class computed `goto` because it calls a new function (thus argument passing, ABI), but targets the same control-flow shape
- Still nightly: exact portability rules (which signatures/targets qualify) are still being decided

Direct Threading with `unsafe`

```
#[cfg(target_arch = "x86_64")]
type Handler = unsafe extern "sysv64" fn(*mut VM, *const Handler) -> !;

pub struct ThreadedCode { handlers: Vec<Handler> }

impl ThreadedCode {
        pub fn new(ops: &[Op]) -> Result<Self, CompileError> {
                let mut handlers = ops.iter().map(handler_for).collect::<Vec<_>>();
                handlers.push(op_halt);
                Ok(Self { handlers })
        }

        pub fn run(&self, vm: &mut VM) -> ! {
                unsafe { jump_to(self.handlers[0], vm, self.handlers.as_ptr().add(1)) }
        }
}
```

- Public API compiles validated opcodes to handler addresses
- Raw addresses and unchecked jumps stay inside the module

Unsafe Jump Primitive

```
#[cfg(target_arch = "x86_64")]
unsafe fn jump_to(next: Handler, vm: &mut VM, ip: *const Handler) -> ! {
        core::arch::asm!(
                "jmp {target}",
                target = in(reg) next,
                in("rdi") vm as *mut VM,
                in("rsi") ip,
                options(noreturn)
        )
}

unsafe extern "sysv64" fn op_add(vm: *mut VM, ip: *const Handler) -> ! {
        unsafe {
                let rhs = (*vm).stack.pop().unwrap_unchecked();
                let lhs = (*vm).stack.pop().unwrap_unchecked();
                (*vm).stack.push(lhs + rhs);
                jump_to(*ip, &mut *vm, ip.add(1));
        }
}
```

- Code is architecture- and ABI-specific
- Raw indirect jump - this is why the safe API boundary matters

A Framework for Safety Arguments about `unsafe` code

- Before trusting `unsafe` code, make a rigorous, structured argument for its correctness
- Four steps:
  i. **Hazards** - what could go wrong? Based on unsafe capabilities: (dangling/invalid pointers, unaligned access, data races, broken aliasing, invalid bit patterns, out-of-bounds access, ...)
  ii. **Invariant** - a precise condition on the representation that rules out every hazard from step i
  iii. **Establish** - every path that creates a value (constructors, decoders, FFI) sets up the invariant before the value reaches safe code
  iv. **Maintain** - every `pub` operation, safe or `unsafe`, preserves the invariant, or restores it before the value is observable to safe code again
- Rust convention (checked by `cargo clippy`): a `// SAFETY: ...` comment on each `unsafe` block gives the invariant(s) it relies on and/or (re-)establishes
- Reference: Jung et al., *RustBelt: Securing the Foundations of the Rust Programming Language*, POPL 2018 formalizes this safe/unsafe boundary reasoning

Correctness Argument

- **Hazards**: jumping to a non-handler address; running a handler whose stack precondition isn't met; `ip` pointing past the end of `handlers`; dereferencing an invalid `VM` pointer
- **Invariant**: every entry in `handlers` is a valid handler address and the vector always ends with `op_halt`; whenever a handler runs, `ip` points within that vector and the VM's stack holds enough values for that opcode
- **Establish**: `ThreadedCode::new` builds `handlers` only from validated opcodes via `handler_for`, validates stack effects, and always appends `op_halt`
- **Maintain**: `ThreadedCode::run` starts `ip` at `handlers[1]`; every handler that continues execution calls `jump_to(*ip, vm, ip.add(1))`, so `ip` always advances one cell within the same vector and the next handler's stack precondition still holds
- Because the interface establishes and maintains the invariant, safe clients cannot manufacture an out-of-bounds `ip` or an invalid handler address

Nightly vs. Unsafe: Trade-offs

- Standardization: `become` is an unstable, evolving nightly feature; the `asm!` sketch works on stable Rust today, but only for the architecture/ABI it targets
- Safety: `become` is checked, safe Rust - the compiler enforces the function signature and guarantees no stack growth
  - the `asm!` version has no compiler-checked guarantees; every invariant is manual
- Performance: the `unsafe` version may be faster (but benchmark it!)
  - `become` goes through a Rust `fn` call: fixed signature, argument-passing convention, possible spills for arguments that don't fit in registers - could impose some overhead
  - hand-written `asm!` gives full control over which registers hold VM state across every handler, with no obligation to follow a general-purpose call signature
  - `become` is designed to fail to compile rather than silently fall back to a stack-growing call - so its risk is a build error, not a hidden stack overflow
  - ordinary calls merely placed in tail position, *without* `become`, get no such guarantee: LLVM's tail-call optimization is best-effort and has, in practice, silently regressed between compiler versions - a real interpreter reported exactly this as a production stack-overflow bug
- Suggestion: prefer `become` if it is available and fast enough; use `asm!` only if profiling shows the ABI boundary still costs real cycles

Alternative: Indirect Threading

- Instruction stream contains pointers to descriptors, not handler addresses
  - Descriptor first field = handler address
  - Other fields = immediates, metadata, profiling counters, etc.

```
desc = load ip        ; load instruction descriptor pointer
ip   = ip + PTR_SIZE ; advance threaded-code pointer
dst  = load desc[0]  ; load handler address from descriptor
jmp  dst             ; jump to handler
```

- Roughly 4 RISC-like dispatch instructions
- More compact code, since opcode arguments are in tnhe descriptor
- Overall: More design flexibility than direct threading, but one extra load

Dispatch Summary

Strategy	Representation	Dispatch cost
`match` jump table	fixed `Instr` enum	~5 + loop jump
function table	opcode bytes + `fn` table	~5, but one is a call
true bytecode	compact opcodes + operands	smaller code footprint
direct threading	handler addresses	~3
indirect threading	descriptor pointers	~4

- Safe Rust can be pretty fast w/ bytecode, slots, checked decoding, explicit function tables
- Fully optimizing dispatch step needs nightly or carefully encapsulated unsafe

Limitations of Safe Rust

- Stable safe Rust does not expose:
  - computed `goto`
  - labels-as-values
  - guaranteed tail-call dispatch
- Options:
  - stay safe and accept match/function-call dispatch
  - use nightly features when they match the design
  - use unsafe internally and expose a small safe API
- Course theme: **unsafe implementation, safe interface, argue correctness at the boundary**

Contrasting Example: Compact Tagged Values

- A naive `Value` enum pays a discriminant tag + padding per value
- Tagged pointers pack the tag into unused pointer bits
  - Heap objects are aligned, so their low bit is `0`
  - Store small integers as `(n << 1) | 1`
- Encapsulated `Value` type:
  - Internal: raw machine word
  - Public: safe constructors and checked accessors
- Core pattern: **unsafe implementation, safe interface, argued correctness at the boundary**

Tagged Value Code

```
use std::marker::PhantomData;

#[repr(align(2))]
pub struct Obj { /* fields omitted */ }

#[derive(Clone, Copy)]
pub struct Value<'heap> {
        bits: usize,
        _heap: PhantomData<&'heap Obj>,
}

impl<'heap> Value<'heap> {
        pub fn int(n: isize) -> Self {
                Self { bits: ((n as usize) << 1) | 1, _heap: PhantomData }
        }

        pub fn obj(obj: &'heap Obj) -> Self {
                let bits = obj as *const Obj as usize;
                debug_assert_eq!(bits & 1, 0);
                Self { bits, _heap: PhantomData }
        }
}
```

Checked Accessors

```
impl<'heap> Value<'heap> {
        pub fn as_int(self) -> Option<isize> {
                (self.bits & 1 == 1).then(|| (self.bits as isize) >> 1)
        }

        pub fn as_obj(self) -> Option<&'heap Obj> {
                if self.bits & 1 == 0 {
                        unsafe { Some(&*(self.bits as *const Obj)) }
                } else {
                        None
                }
        }
}
```

- Only `as_obj` needs `unsafe`
- The tag check must happen before the raw pointer dereference

Exercise: Argue that Tagged Value Code is Safe

- What are the hazards?
- What are the safety invariant(s)?
- How are the safety invariants established?
- How are the safety invariants maintained?

Safety Argument for the Value API

- **Hazards**: dereferencing `bits` as an `Obj` pointer when the tag says otherwise, or when it's dangling; producing a `Value` whose bits match no valid encoding; a recovered `&'heap Obj` outliving the `Obj` it points to
- **Invariant**: low bit `1` means the remaining bits are a shifted `isize`; low bit `0` means the bits are a non-null, aligned pointer to an `Obj` that is live for `'heap`
- **Establish**: `Value::int` builds the tagged-integer bit pattern directly; `Value::obj` copies the address of an actual `&'heap Obj`, so `PhantomData<&'heap Obj>` ties the encoded pointer to a lifetime that is still valid
- **Maintain**: `Value` is `Copy` with no operation that mutates `bits` after construction, so the invariant holds unchanged for the value's whole lifetime; `as_int` / `as_obj` only read `bits` and check the tag before trusting it
- What would break it: a public `from_bits` bypassing the constructors, or an `Obj` deallocated while some `Value<'heap>` still encodes its address

Extensions and Resources

- Register-based vs stack-based bytecode (Lua/Dalvik vs JVM/CPython, vs Wasm's stack model)
- Superinstructions / instruction fusion - fuse common sequences to cut dispatch overhead
- Inline caching - remember the result of a dynamic lookup at the call site
- JIT compilation - deferred to Lecture 4
- Further reading: Ben Titzer, *A Fast In-Place Interpreter for WebAssembly*, OOPSLA 2022 - combines many of today's techniques in a production-grade interpreter

Summary: Fast Interpreters in Rust

- We covered bytecode, threaded interpreters, and tagged values
- Many optimizations are achievable in safe Rust
- For others: can a Nightly feature be used? If not, what unsafe invariant would need to be documented and maintained?
- Next lecture: Just-In-Time Compilation in Rust