Trust the Compiler

The cool things the compiler does behind your back
(and how you can make use of them)

The compiler does a LOT

Rust and LLVM each have complex optimization passes

Things like data flow and control flow analysis optimizes so many simple things

Conditional flow

Consider this simple function:

```rust
#[no_mangle]
pub fn run_conditionally(val: u32) {
    let reduced = val % 2;

    match reduced {
        0 => foo(),
        1 => bar(),
        _ => unreachable_fn(),
    }
}
```

Compiled LLVM IR

The unreachable code gets optimized away

```
define void @run_conditionally(i32 noundef %val) unnamed_addr #1 {
  %0 = and i32 %val, 1
  %trunc.not = icmp eq i32 %0, 0
  br i1 %trunc.not, label %bb2, label %bb3

bb2:                                              ; preds = %start
  tail call void @foo()
  br label %bb4

bb3:                                              ; preds = %start
  tail call void @bar()
  br label %bb4

bb4:                                              ; preds = %bb3, %bb2
  ret void
}
```

This can be taken further...

Assume the below function. What if `index` is always <10?

```rust
pub fn index_under_10(arr: [i32; 10], index: usize) -> i32 {
    arr[index]
}
```

Rust still generates the check, and a panic call

But you still want to force it away

Instead of resorting to raw pointer operations, simply add an assertion

```
pub fn index_under_10_no_panic(arr: [i32; 10], index: usize) -> i32 {
    unsafe {
        if index >= 10 {
            std::hint::unreachable_unchecked();
        }
    }

    arr[index]
}
```

And it's gone!

```
example::index_under_10:
        mov     rax, rsi
        cmp     rsi, 9
        ja      .LBB0_2
        mov     eax, dword ptr [rdi + 4*rax]
        ret
.LBB0_2:
        push    rax
        lea     rdx, [rip + .L__unnamed_1]
        mov     esi, 10
        mov     rdi, rax
        call    qword ptr [rip + core::panicking::panic_bounds_check@GOTPCREL]
        ud2

example::index_under_10_no_panic:
        mov     eax, dword ptr [rdi + 4*rsi]
        ret
```

Static vs Dynamic arrays

Let's say you have an array that's always only 4 elements long.

You need to perform an action on it. Say, sum:

```rust
#[inline(never)]
pub fn sum_vec(arr: Vec<f32>) -> f32 {
    arr.iter().sum()
}
```

What actually gets generated?

Let's see the assembly for this simple function

That's a lot of instructions...

For just 4 elements

```
example::sum_vec:
        push    rax
        mov     rax, rdi
        mov     rdi, qword ptr [rdi]
        mov     rsi, qword ptr [rax + 16]
        test    rsi, rsi
        je      .LBB0_1
        mov     ecx, esi
        and     ecx, 7
        cmp     rsi, 8
        jae     .LBB0_4
        xorps   xmm0, xmm0
        xor     edx, edx
        jmp     .LBB0_6
.LBB0_1:
        xorps   xmm0, xmm0
        jmp     .LBB0_9
.LBB0_4:
        and     rsi, -8
        xorps   xmm0, xmm0
        xor     edx, edx
.LBB0_5:
        addss   xmm0, dword ptr [rdi + 4*rdx]
        addss   xmm0, dword ptr [rdi + 4*rdx + 4]
        addss   xmm0, dword ptr [rdi + 4*rdx + 8]
        addss   xmm0, dword ptr [rdi + 4*rdx + 12]
        addss   xmm0, dword ptr [rdi + 4*rdx + 16]
        addss   xmm0, dword ptr [rdi + 4*rdx + 20]
        addss   xmm0, dword ptr [rdi + 4*rdx + 24]
        addss   xmm0, dword ptr [rdi + 4*rdx + 28]
        add     rdx, 8
        cmp     rsi, rdx
        jne     .LBB0_5
.LBB0_6:
        test    rcx, rcx
        je      .LBB0_9
        lea     rdx, [rdi + 4*rdx]
        xor     esi, esi
.LBB0_8:
        addss   xmm0, dword ptr [rdx + 4*rsi]
        inc     rsi
        cmp     rcx, rsi
        jne     .LBB0_8
.LBB0_9:
        mov     rsi, qword ptr [rax + 8]
        test    rsi, rsi
        je      .LBB0_11
        shl     rsi, 2
        mov     edx, 4
        movss   dword ptr [rsp + 4], xmm0
        call    qword ptr [rip + __rust_dealloc@GOTPCREL]
        movss   xmm0, dword ptr [rsp + 4]
.LBB0_11:
        pop     rax
        ret
```

Now let's change just 1 thing

The type from Vec to array

```
#[inline(never)]
pub fn sum_arr(arr: [f32; 4]) -> f32 {
    arr.iter().sum()
}
```

What actually gets generated now?

That's it

```
example::sum_arr:
        xorps   xmm0, xmm0
        addss   xmm0, dword ptr [rdi]
        addss   xmm0, dword ptr [rdi + 4]
        addss   xmm0, dword ptr [rdi + 8]
        addss   xmm0, dword ptr [rdi + 12]
        ret
```

Static vs Dynamic dispatch

Consider the following simple rust trait:

```rust
trait Execute {
    fn execute(&self, x: i32) -> i32;
}

struct AddX {
    x: i32,
}

impl Execute for AddX {
    fn execute(&self, x: i32) -> i32 {
        self.x + x
    }
}
```

Static vs Dynamic dispatch

Here are the two main ways to execute them:

```rust
fn execute_dyn(ex: &dyn Execute, x: i32) -> i32 {
    ex.execute(x)
}

fn execute_generic<T: Execute>(ex: &T, x: i32) -> i32 {
    ex.execute(x)
}

pub fn run() {
    let add = AddX { x: 5 };

    dbg!(execute_dyn(&add, 7));
    dbg!(execute_generic(&add, 7));
}
```

Underlying mechanics

LLVM may look intimidating at first

but it's fairly simple

```
define i32 @execute(ptr %self, i32 %x) {
  // Read the field from the object
  %_3 = load i32, ptr %self, align 4
  // Add the field to the argument
  %_0 = add i32 %_3, %x
  // Return the result
  ret i32 %_0
}

define i32 @execute_dyn(ptr %ex.0, ptr %ex.1, i32 %x) {
  // Fetch the vtable pointer from the object
  %0 = getelementptr inbounds ptr, ptr %ex.1, i64 3
  // Read the function pointer from the vtable
  %1 = load ptr, ptr %0, align 8
  // Call the function pointer with the object and the argument
  %_0 = tail call i32 %1(ptr %ex.0, i32 %x)
  // Return the result
  ret i32 %_0
}

define internal fastcc i32 @execute_generic(ptr %ex) {
  // Directly call the execute function with the object and the argument
  %_0 = tail call i32 @execute(ptr %ex, i32 7)
  // Return the result
  ret i32 %_0
}
```

What's that argument?

It's vtables!

```
@vtable.0 = private unnamed_addr constant <{ ptr, [16 x i8], ptr }>
  <{
    ptr @"_ZN4core3ptr34drop_in_place$LT$example..AddX$GT$17h2205ae1cb6e0e05dE",
    [16 x i8] c"\04\00\00\00\00\00\00\00\04\00\00\00\00\00\00\00",
    ptr @execute
  }>, align 8

define void @run() {
  %add = alloca i32, align 4
  store i32 5, ptr %add, align 4
  %_2 = call i32 @execute_dyn(ptr %add, ptr @vtable.0, i32 7)
  %_21 = call i32 @execute_generic(ptr %add)
  ret void
}
```

So, static or dynamic?

Static:

- Compiler can be smarter about it
- No runtime overhead

Dynamic:

- Easier to do
- Reduces the amount of code generated

Summary

- Tell the compiler your intentions
- The compiler uses those intentions to optimize code
- The compiler can optimize:
  - Data flow
  - Control flow
  - Function calls
  - Algorithms
  - And more

Thanks