Rust Dublin

Zero Cost Abstractions

Michael Barber

Abstractions

- Abstractions are higher-level concepts
  - Ignore the irrelevant parts of the problem
  - Focus on the detail
- Abstractions are *helpful*
  - Clearly express what we're doing
  - Reduce cognitive load and errors
- Examples
  - Using Java or C# instead of C or Assembly
  - Using classes, interfaces, traits
  - Using iterators and functions instead of imperative loops
  - And tons more
- Abstractions are not always free
  - Let's find out… looking at **iterators** today

theoretically,

```
M00_L00:
       movsxd    rcx,esi
       mov       ecx,[r15+rcx*4+10]
       movsxd    r8,esi
       mov       r8d,[rax+r8*4+10]
       cmp       ecx,2
       jle       short M00_L01
       imul      ecx,r8d
       movsxd    r8,ecx
       add       rdi,r8
M00_L01:
       inc       esi
       cmp       edx,esi
       jg        short M00_L00
       jmp       short M00_L04
```

C#

```
public static long CalculateIterator(int[] va, int[] vb)
{
    var res = va.Zip(vb)
        .Where(pair => pair.First > 2)
        .Select(pair => (long)(pair.First * pair.Second))
        .Sum();
    return res;
}
```

```
Start with sum = 0
for every pair a, b in aligned vectors va, vb
  if a > 2, then sum += a * b
return sum
```

```
va, vb have 20k elements, flat vectors, int32 of [0;10)
Randomly selected from precalculated set of 100 vectors
Setup: WSL2 Ubuntu 20.04, AMD Ryzen 3900X, single threaded benchmarks
```

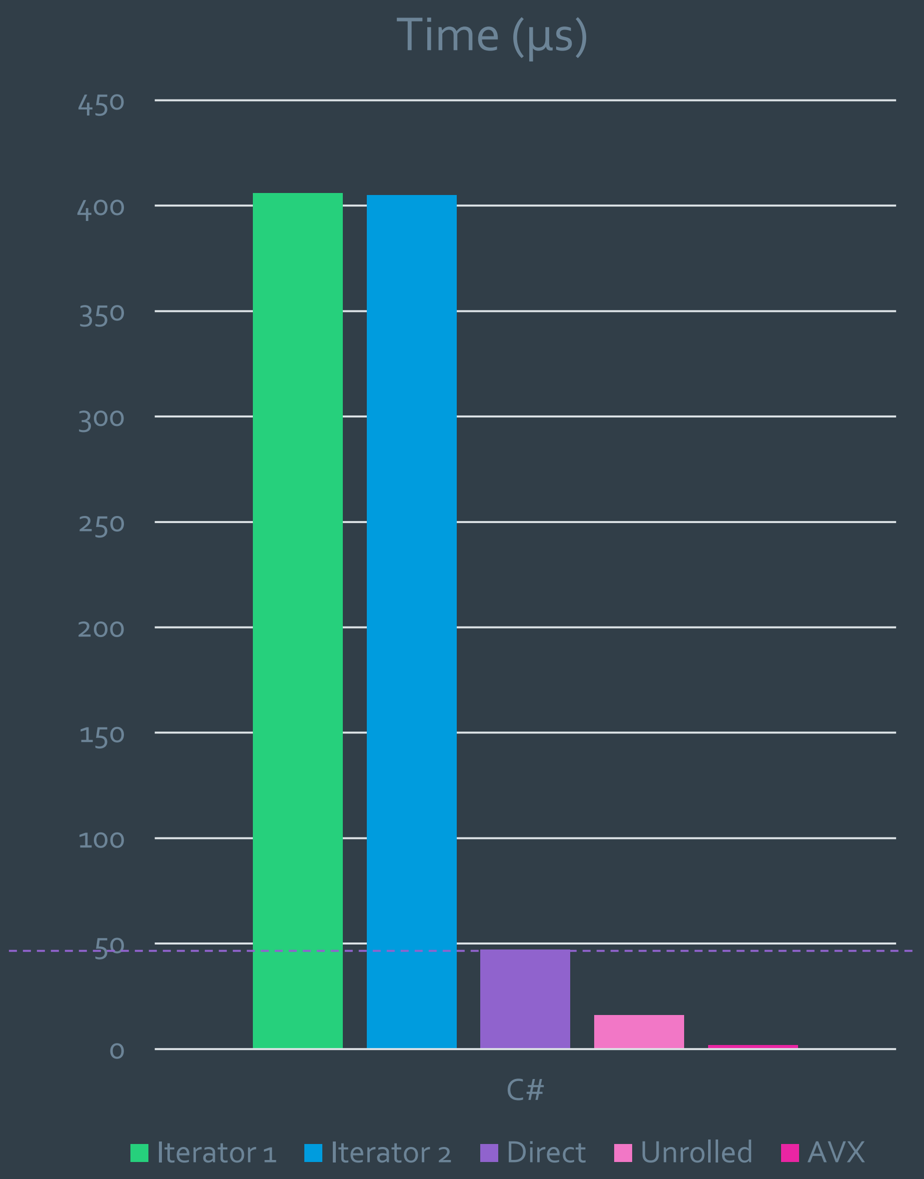
Costs – C# dotnet 5.0

- Abstractions are actually very expensive
  - Iterator 406 µs
  - Direct Loop 47 µs (typical approach)
  - Unsafe, unrolled loop 16 µs
  - Unsafe, AVX2 hand coded 2 µs (max throughput)

```
public static long CalculateIterator(int[] va, int[] vb)
{
    var res = va.Zip(vb)
        .Where(pair => pair.First > 2)
        .Select(pair => (long)(pair.First * pair.Second))
        .Sum();
    return res;
}
```

```
public static long CalculateDirect(int[] va, int[] vb)
{
    long sum = 0;
    for (var i = 0; i < va.Length; ++i)
    {
        var a = va[i];
        var b = vb[i];
        if (a > 2)
        {
            sum += a * b;
        }
    }
    return sum;
}
```

```
public static long CalculateDirectUnsafeAvx(int[] vecta, int[] vectb)
{
    if (vecta.Length != vectb.Length) throw new ArgumentException("length mismatch");

    long sum = 0;
    var len = vecta.Length;
    var chunkEndIndex = (len >> 3) << 3;
    unsafe
    {
        fixed (int* ptra = vecta, ptrb = vectb)
        {
            var chunkEnd = ptra + chunkEndIndex;
            var allEnd = ptra + len;
            var pa = ptra;
            var pb = ptrb;

            var value2 = Vector256.Create(2);
            var value0 = Vector256.Create(0);

            // two accumulators, each 4 x int64, for total 8 x int64
            var acc1 = Vector256.Create(0L);
            var acc2 = Vector256.Create(0L);

            // main loop -- 8 elements at a time
            while (pa < chunkEnd)
            {
                var a = Avx.LoadVector256(pa);
                var b = Avx.LoadVector256(pb);

                var mask = Avx2.CompareGreaterThan(a, value2);
                a = Avx2.And(a, mask);

                // odd numbered elements (i32 * i32 -> i64)
                var m1 = Avx2.Multiply(a, b);
                acc1 = Avx2.Add(acc1, m1);

                // shuffle adjacent and multiply again
                a = Avx2.Shuffle(a, 0b10_11_00_01);
                b = Avx2.Shuffle(b, 0b10_11_00_01);
                var m2 = Avx2.Multiply(a, b);
                acc2 = Avx2.Add(acc2, m2);

                pa += 8;
                pb += 8;
            }
            // tail loop -- single elements
            while (pa < allEnd)
            {
                // could do something smart like a masked load here, but
                // this will do for demonstration purposes.
                var a = Vector256.CreateScalar(*pa);
                var b = Vector256.CreateScalar(*pb);

                // as above, accumulating into acc1
                var mask = Avx2.CompareGreaterThan(a, value2);
                a = Avx2.And(a, mask);
                var m1 = Avx2.Multiply(a, b);
                acc1 = Avx2.Add(acc1, m1);

                pa++;
                pb++;
            }

            // sum all the elements
            acc1 = Avx2.Add(acc1, acc2);
            sum = acc1.GetElement(0) + acc1.GetElement(1) + acc1.GetElement(2) + acc1.GetElement(3);
        }
    }
    return sum;
}
```



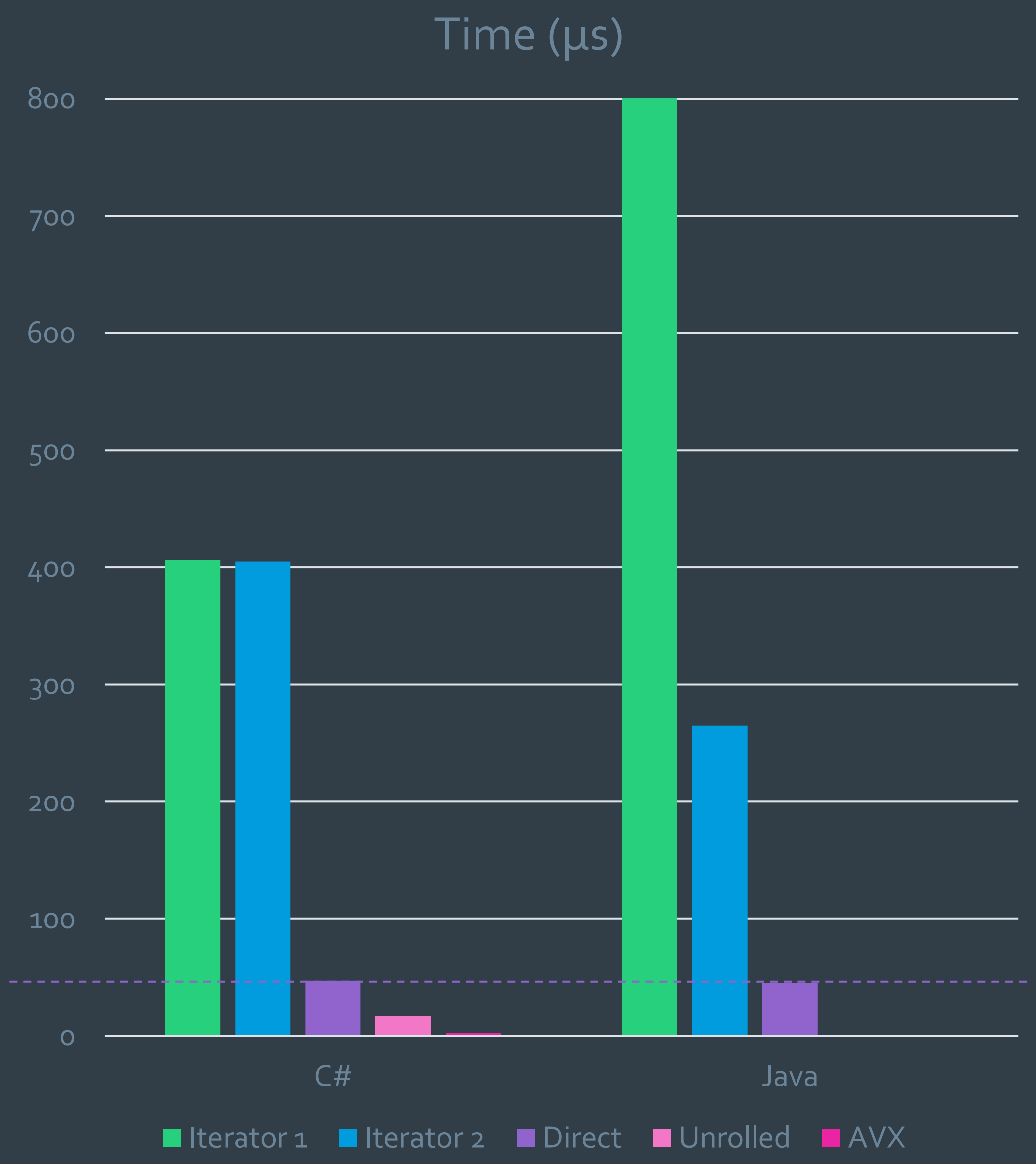
Costs – Java 11

- Java doesn't get away with it either.
  - Iterator, Guava zip 913 µs
  - Iterator, hand-coded zip 265 µs
  - Direct Loop 45 µs (typical approach)
- Perhaps Scala fairs better
  - Suspect iterator might still be relatively expensive
  - Let me know!

```
// direct loop
public long benchmarkDirect() {
    long sum = 0;
    var va = Sample(ThreadState.rng);
    var vb = Sample(ThreadState.rng);
    for (int i=0; i<va.length; ++i) {
        var a = va[i];
        var b = vb[i];
        if (a > 2) {
            sum += a * b;
        }
    }
    return sum;
}
```

```
// iterator 2
public long benchmarkIterator() {
    var va = Sample(ThreadState.rng);
    var vb = Sample(ThreadState.rng);
    // using my own zip method
    long sum = zip(Arrays.stream(va), Arrays.stream(vb))
        .filter(t -> t._1 > 2)
        .map(t -> (long)(t._1 * t._2))
        .reduce(0l, Long::sum);

    return sum;
}
```

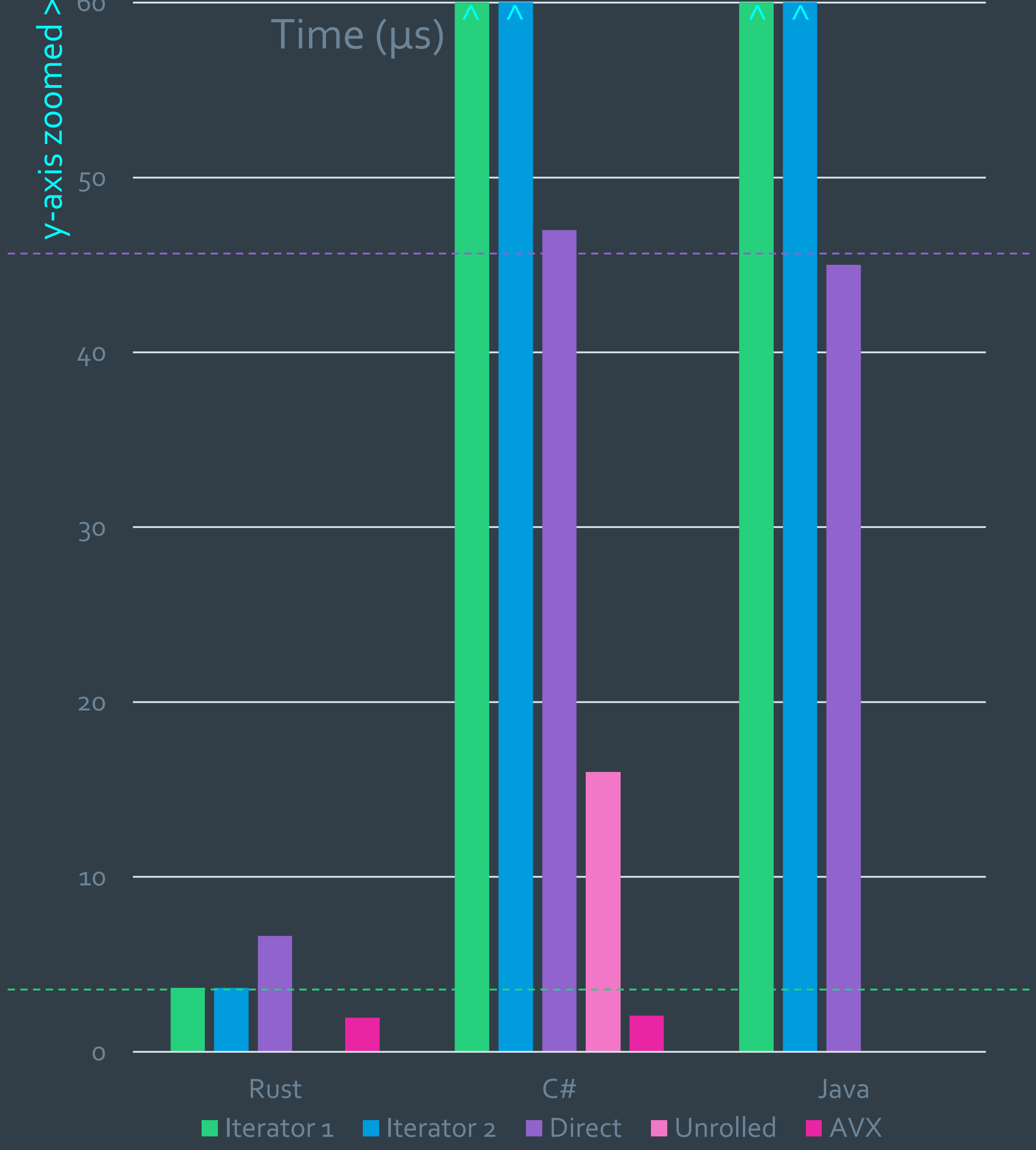
- Let's look at Rust next…



Costs – Rust 1.50

- Rust abstractions are often *faster* than a loop
  - Iterator filter map 3.66 µs
  - Iterator fold 3.66 µs
  - Direct Loop 6.60 µs
  - AVX2 1.90 us
- "Zero Cost Abstractions"
  - Compiler knows all the tricks: branchless, AVX, unrolled, inline, etc.
  - And we get to keep our nice abstractions

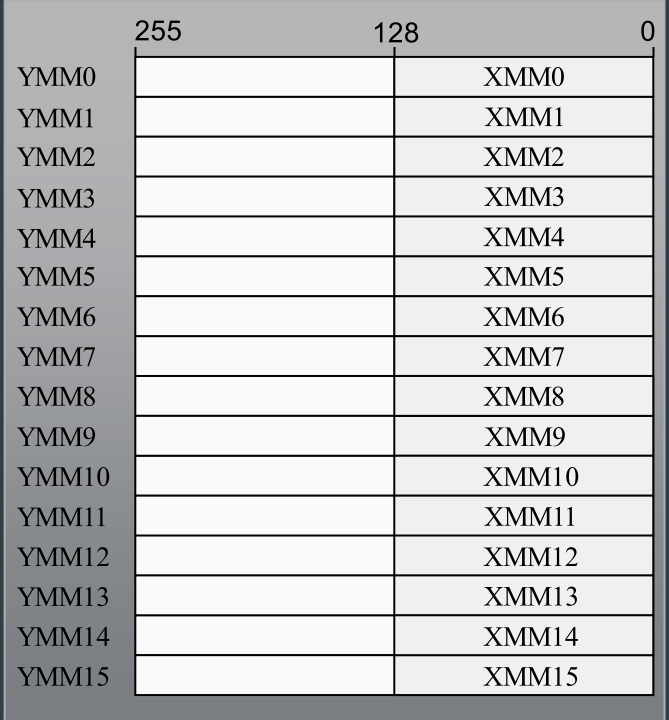
```
pub fn calculate_iter(slice_a: &[i32], slice_b: &[i32]) -> i64 {
    slice_a
        .iter()
        .zip(slice_b.iter())
        .filter_map(|(a, b)| match *a > 2 {
            true => Some(*a as i64 * *b as i64),
            false => None,
        })
        .sum()
}

pub fn calculate_direct(slice_a: &[i32], slice_b: &[i32]) -> i64 {
    let mut res = 0;
    for (a, b) in slice_a.iter().zip(slice_b.iter()) {
        if *a > 2 {
            res += *a as i64 * *b as i64;
        }
    }
    res
}
```



SIMD Refresher

Single Instruction Multiple Data

- Higher throughput per instruction
- SSE2 – XMM registers
  - 128 bit wide registers
  - On all AMD/Intel 64-bit processors
  - 4x 32-bit integers/floats 
  - 2x 64-bit integers/floats
- AVX – YMM registers
  - 256 bit wide registers
  - AVX is floating point mostly
  - AVX2 is integer mostly
  - On all AMD/Intel processors since about 2013
  - 8x 32-bit integers/floats 
  - 4x 64-bit integers/floats
  - YMM aliases XMM
- AVX may not be faster on older Intel CPUs
- AVX is easier to use – non-destructive
- Tons of instructions, e.g. add, multiply, shift, and, etc.



Source: Wikipedia

SIMD in Rust

- Currently, on Stable:
  - LLVM auto-vectorisation works on most platforms
  - Intrinsics
    - x86-64 is supported: same intrinsics as in C++; **std::arch::x86_64**
    - Unsafe and very low level
    - Refer to the Intel Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/
    - Crates like **simdeez** for slightly higher-level code
  - Target the right processor
    - **.cargo/config** or **RUSTFLAGS**
    - **target-cpu=native** or **target-cpu=skylake**
    - https://docs.rs/rustc-std-workspace-std/1.0.1/std/arch/index.html
- Currently, on Unstable:
  - Intrinsics for ARM, WebAssembly, etc.
  - Inline assembly for most platforms
  - Crates like **faster**, **packed_simd** for higher-level code
  - Very active area; expect changes
- Future
  - Rust Portable SIMD Project Group
  - std::simd
  - Very complex problem to solve – will take some time
- Should you use it? Maybe. Relative benefit is lower.

```rust
use std::arch::x86_64::*;
use std::mem::transmute;

const SIZE: usize = 8;
let value2 = unsafe { _mm256_set1_epi32(2) }; // broadcast 2
let mut acc1 = unsafe { _mm256_setzero_si256() };
let mut acc2 = unsafe { _mm256_setzero_si256() };
for (chunk_a, chunk_b) in slice_a
    .chunks_exact(SIZE)
    .zip(slice_b.chunks_exact(SIZE)) {
    unsafe {
        // load slices into AVX registers, unaligned load
        let mut va = _mm256_loadu_si256(chunk_a.as_ptr() as *const __m256i);
        let vb = _mm256_loadu_si256(chunk_b.as_ptr() as *const __m256i);

        // zero out `va` elements where they're less than 2
        let mask = _mm256_cmpgt_epi32(va, value2);
        va = _mm256_and_si256(va, mask);

        // odd numbered elements (i32 * i32 -> i64)
        let m1 = _mm256_mul_epi32(va, vb);
        acc1 = _mm256_add_epi64(acc1, m1); // note: 64-bit integer addition

        // shuffle adjacent to switch even/odd and multiply odd elements again
        let shuf_va = _mm256_shuffle_epi32(va, 0b10_11_00_01);
        let shuf_vb = _mm256_shuffle_epi32(vb, 0b10_11_00_01);
        let m2 = _mm256_mul_epi32(shuf_va, shuf_vb);
        acc2 = _mm256_add_epi64(acc2, m2); // note: 64-bit integer addition
    }
}

// call existing function for the remaining items
let remainder = calculate_iter(
    slice_a.chunks_exact(SIZE).remainder(),
    slice_b.chunks_exact(SIZE).remainder(),
);

// final total
let sum: i64 = unsafe {
    let acc = _mm256_add_epi64(acc1, acc2);
    let acc: &[i64; 4] = transmute(&acc);
    acc.iter().sum::<i64>() + remainder
};
```

Conclusions

- Abstractions are expensive in many languages…
  - Know how to identify hot code paths
  - Use classic loops for these; abstractions elsewhere
  - Know how to do micro-benchmarking and profiling
  - If it’s a **really** hot loop, consider more aggressive optimisation
  - Keep an eye on SIMD support in your language
- C#, Java
  - Use the classic optimisation techniques (e.g. unrolling)
  - JIT compiler is in a hurry and won’t do these for you
- Rust
  - Use the abstractions – they’re fast!
  - Target the right CPU
  - Let the compiler do most of the hard work
  - Check your assembly: **--emit asm** or www.godbolt.org
  - Use **Criterion** for benchmarking

- Learn some Rust!

  “A language empowering everyone
  to build reliable and efficient software.”

  - Extremely interesting language – more than just speed
  - Stack Overflow’s [Most Loved Language]() for last 5 years
  - Little bit of a learning curve, and useful outside of Rust too
  - Functional devs will enjoy the type system
  - Really good tooling
  - They have a cool mascot:

<Ferris

Code for this presentation:
https://github.com/mike-barber/rust-zero-cost-abstractions