**Rust —avoid recursive iterator 2**

[[TechHara](https://medium.com/@techhara?source=post_page-----5cd90a59ffe6--------------------------------)](https://medium.com/@techhara?source=post_page-----5cd90a59ffe6--------------------------------)

[TechHara](https://medium.com/@techhara?source=post_page-----5cd90a59ffe6--------------------------------)

·

Follow

6 min read

·

1 day ago

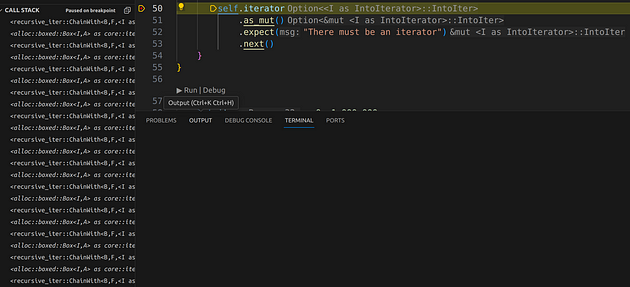
**Recap**

In the last [story](https://medium.com/@techhara/rust-avoid-recursive-iterators-50e5701e3743), we looked at how iterative call to Iterator::chain() method can crash with stack overflow just from construction. Even after replacing with lazy chain\_with() method, the program still crashes with stack overflow. Below is the recap of the recursive implementation in Rust

fn add\_two(mut iter: impl Iterator<Item = u32> + 'static) -> Box<dyn Iterator<Item = u32>> {  
 match (iter.next(), iter.next()) {  
 (Some(x), Some(y)) => Box::new(std::iter::once(x + y).chain\_with(|| add\_two(iter))),  
 \_ => Box::new(std::iter::empty()),  
 }  
}  
  
trait IteratorExt: Iterator {  
 fn chain\_with<F, I>(self, f: F) -> ChainWith<Self, F, I::IntoIter>  
 where  
 Self: Sized,  
 F: FnOnce() -> I,  
 I: IntoIterator<Item = Self::Item>,  
 {  
 ChainWith {  
 base: self,  
 factory: Some(f),  
 iterator: None,  
 }  
 }  
}  
  
impl<I: Iterator> IteratorExt for I {}  
  
struct ChainWith<B, F, I> {  
 base: B,  
 factory: Option<F>,  
 iterator: Option<I>,  
}  
  
impl<B, F, I> Iterator for ChainWith<B, F, I::IntoIter>  
where  
 B: Iterator,  
 F: FnOnce() -> I,  
 I: IntoIterator<Item = B::Item>,  
{  
 type Item = I::Item;  
 fn next(&mut self) -> Option<Self::Item> {  
 if let Some(b) = self.base.next() {  
 return Some(b);  
 }  
  
 // Exhausted the first, generate the second  
  
 if let Some(f) = self.factory.take() {  
 self.iterator = Some(f().into\_iter());  
 }  
  
 self.iterator  
 .as\_mut()  
 .expect("There must be an iterator")  
 .next()  
 }  
}  
  
fn main() {  
 let iter = 0..1\_000\_000;  
 // consume iter and generate an iterator that returns sum of pairs  
 let sum\_iter = add\_two(iter); // [0+1, 2+3, 4+5, 6+7, ...]  
 println!("{}", sum\_iter.count());  
}

**Dynamic dispatch is culprit**

This time, however, we can at least execute let some\_iter = add\_two(iter); line. The program crashes at the next line, when we are actually evaluating the iterator. Why? We can’t tail-optimize the recursion because the type is unknown at compile time. You see, we are returning Box<dyn Iterator> whose type is determined at runtime. In our case, this can go up to 1M times nested ChainWith struct, like ChainWith<ChainWith<ChainWith<...>>>. Every time we execute next() method, we have to keep unwrapping ChainWith's inner iterator with next() method. We can confirm this from the stack trace on the left



Nested call into ChainWith::next()

Eventually, when the depth exceeds program’s stack, the program crashes. Because iterator methods are being dynamically dispatched, compiler can’t even do tail-recursive optimization.

**Is this Rust problem?**

Not really. we can write a direct translation of the same implementation into C++, and the behavior is essentially the same when compiled with clang++ -O2.

#include <functional>  
#include <iostream>  
#include <memory>  
#include <optional>  
  
template <typename T> class Iterator {  
public:  
 virtual ~Iterator() = default;  
 virtual std::optional<T> next() = 0;  
 virtual std::size\_t count() {  
 std::size\_t c = 0;  
 while (next())  
 ++c;  
 return c;  
 }  
};  
  
template <typename T> class RangeIterator : public Iterator<T> {  
 T current;  
 T end;  
public:  
 RangeIterator(T start, T end) : current(start), end(end) {}  
 std::optional<T> next() override {  
 if (current < end) {  
 return current++;  
 }  
 return {};  
 }  
};  
  
template <typename T> class EmptyIterator : public Iterator<T> {  
public:  
 std::optional<T> next() override { return {}; }  
};  
  
template <typename T> class OnceIterator : public Iterator<T> {  
 std::optional<T> x;  
public:  
 explicit OnceIterator(T x) : x{std::move(x)} {}  
 std::optional<T> next() override {  
 auto ret = std::move(x);  
 if (ret)  
 x.reset();  
 return ret;  
 }  
};  
  
template <typename T> class ChainedIterator : public Iterator<T> {  
 std::unique\_ptr<Iterator<T>> iter1, iter2;  
public:  
 ChainedIterator(std::unique\_ptr<Iterator<T>> iter1,  
 std::unique\_ptr<Iterator<T>> iter2)  
 : iter1{std::move(iter1)}, iter2{std::move(iter2)} {}  
 std::optional<T> next() override {  
 if (iter1) {  
 auto x = iter1->next();  
 if (x)  
 return x;  
 iter1 = nullptr;  
 }  
 return iter2->next();  
 }  
};  
  
template <typename T, typename F> class ChainWithIterator : public Iterator<T> {  
 std::unique\_ptr<Iterator<T>> iter1, iter2;  
 F f;  
public:  
 ChainWithIterator(std::unique\_ptr<Iterator<T>> iter1, F &&f)  
 : iter1{std::move(iter1)}, f{std::forward<F>(f)} {}  
 std::optional<T> next() override {  
 if (iter1) {  
 auto x = iter1->next();  
 if (x)  
 return x;  
 iter1 = nullptr;  
 iter2 = f();  
 }  
 return iter2->next();  
 }  
};  
  
template <typename T>  
std::unique\_ptr<Iterator<T>> add\_two(std::unique\_ptr<Iterator<T>> iter) {  
 auto first = iter->next();  
 auto second = iter->next();  
 if (first && second) {  
 T sum = \*first + \*second;  
 std::unique\_ptr<Iterator<uint32\_t>> once =  
 std::make\_unique<OnceIterator<T>>(sum);  
 return std::unique\_ptr<Iterator<T>>(new ChainWithIterator{  
 std::move(once), [iter = std::move(iter)]() mutable {  
 return add\_two(std::move(iter));  
 }});  
 }  
 return std::make\_unique<EmptyIterator<T>>();  
}  
  
int main() {  
 std::unique\_ptr<Iterator<uint32\_t>> iter =  
 std::make\_unique<RangeIterator<uint32\_t>>(0, 1'000'000);  
 auto sum\_iter = add\_two(std::move(iter));  
 std::cout << sum\_iter->count() << "\n";  
 return 0;  
}

I wonder if just-in-time (JIT) compiler can perhaps optimize it while running, so let’s ask [Bing Chat](http://www.bing.com/chat) to write Java implementation for us.

#include <functional>  
#include <iostream>  
#include <memory>  
#include <optional>  
  
template <typename T> class Iterator {  
public:  
 virtual ~Iterator() = default;  
 virtual std::optional<T> next() = 0;  
 virtual std::size\_t count() {  
 std::size\_t c = 0;  
 while (next())  
 ++c;  
 return c;  
 }  
};  
  
template <typename T> class RangeIterator : public Iterator<T> {  
 T current;  
 T end;  
public:  
 RangeIterator(T start, T end) : current(start), end(end) {}  
 std::optional<T> next() override {  
 if (current < end) {  
 return current++;  
 }  
 return {};  
 }  
};  
  
template <typename T> class EmptyIterator : public Iterator<T> {  
public:  
 std::optional<T> next() override { return {}; }  
};  
  
template <typename T> class OnceIterator : public Iterator<T> {  
 std::optional<T> x;  
public:  
 explicit OnceIterator(T x) : x{std::move(x)} {}  
 std::optional<T> next() override {  
 auto ret = std::move(x);  
 if (ret)  
 x.reset();  
 return ret;  
 }  
};  
  
template <typename T> class ChainedIterator : public Iterator<T> {  
 std::unique\_ptr<Iterator<T>> iter1, iter2;  
public:  
 ChainedIterator(std::unique\_ptr<Iterator<T>> iter1,  
 std::unique\_ptr<Iterator<T>> iter2)  
 : iter1{std::move(iter1)}, iter2{std::move(iter2)} {}  
 std::optional<T> next() override {  
 if (iter1) {  
 auto x = iter1->next();  
 if (x)  
 return x;  
 iter1 = nullptr;  
 }  
 return iter2->next();  
 }  
};  
  
template <typename T, typename F> class ChainWithIterator : public Iterator<T> {  
 std::unique\_ptr<Iterator<T>> iter1, iter2;  
 F f;  
public:  
 ChainWithIterator(std::unique\_ptr<Iterator<T>> iter1, F &&f)  
 : iter1{std::move(iter1)}, f{std::forward<F>(f)} {}  
 std::optional<T> next() override {  
 if (iter1) {  
 auto x = iter1->next();  
 if (x)  
 return x;  
 iter1 = nullptr;  
 iter2 = f();  
 }  
 return iter2->next();  
 }  
};  
  
template <typename T>  
std::unique\_ptr<Iterator<T>> add\_two(std::unique\_ptr<Iterator<T>> iter) {  
 auto first = iter->next();  
 auto second = iter->next();  
 if (first && second) {  
 T sum = \*first + \*second;  
 std::unique\_ptr<Iterator<uint32\_t>> once =  
 std::make\_unique<OnceIterator<T>>(sum);  
 return std::unique\_ptr<Iterator<T>>(new ChainWithIterator{  
 std::move(once), [iter = std::move(iter)]() mutable {  
 return add\_two(std::move(iter));  
 }});  
 }  
 return std::make\_unique<EmptyIterator<T>>();  
}  
  
int main() {  
 std::unique\_ptr<Iterator<uint32\_t>> iter =  
 std::make\_unique<RangeIterator<uint32\_t>>(0, 1'000'000);  
 auto sum\_iter = add\_two(std::move(iter));  
 std::cout << sum\_iter->count() << "\n";  
 return 0;  
}

Unfortunately, the Java program exhibits the same problem — it crashes with a stack overflow, implying that even JIT can’t optimize it.

I am convinced that recursive iterators through chain() variants are simply impractical. The examples found on the web 1 2 [3](https://fasterthanli.me/articles/recursive-iterators-rust) work OK **only if the recursion depth is minimal**. These implementations will not only crash with stack overflow but also run very slow and inefficient even when it works.

**The practical solution**

I find no other way but to revert back to non-recursive solution. However, I wanted to create a helper method in Iterator that hides away the implementation detail and exposes an API that looks as if it is a recursive solution. So, here is what I came up with [this](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=54196680833ccdfc04d530207ff2c8e7)

use std::marker::PhantomData;  
  
fn add\_two(iter: impl Iterator<Item = u32>) -> impl Iterator<Item = u32> {  
 iter.recursive\_chain(|iter| match (iter.next(), iter.next()) {  
 (Some(x), Some(y)) => Some(x + y),  
 \_ => None,  
 })  
}  
  
trait IteratorExt: Iterator {  
 fn recursive\_chain<F, R>(self, f: F) -> RecursiveChain<Self, F, R>  
 where  
 Self: Sized,  
 F: FnMut(&mut Self) -> Option<R>,  
 {  
 RecursiveChain {  
 base: self,  
 f,  
 r: PhantomData::default(),  
 }  
 }  
}  
  
impl<I: Iterator> IteratorExt for I {}  
  
struct RecursiveChain<B, F, R> {  
 base: B,  
 f: F,  
 r: PhantomData<R>,  
}  
  
impl<B: Iterator, F: FnMut(&mut B) -> Option<R>, R> Iterator for RecursiveChain<B, F, R> {  
 type Item = R;  
  
 fn next(&mut self) -> Option<Self::Item> {  
 (self.f)(&mut self.base)  
 }  
}  
  
fn main() {  
 let iter = 0..1\_000\_000\_000;  
 // consume iter and generate an iterator that returns sum of pairs  
 let sum\_iter = add\_two(iter); // [0+1, 2+3, 4+5, 6+7, ...]  
 println!("{}", sum\_iter.count());  
}

The recursive\_chain() method takes a function f: FnMut which outputs Option<R> given the iterator. This method transforms Iterator<Item=I> into Iterator<Item=R> by applying f iteratively to the iterator.

**How about Haskell?**

The motivation for doing all this analysis is because in Haskell, the recursive iterator works fine, not crashing with stack overflow

{-# LANGUAGE NumericUnderscores #-}  
  
twoSum :: [Int] -> [Int]  
twoSum [] = []  
twoSum [\_] = []  
twoSum (x:y:xs) = (x+y) : twoSum xs  
  
main = do  
 let iter = take 1\_000\_000\_000 [0..] :: [Int]  
 let sum\_iter = twoSum iter  
 putStrLn $ show $ length sum\_iter

Yes, it is 10+ times slower than the iterative version written in Rust, but look how concise we can write in Haskell. I am curious as to how Haskell can run without stack overflow. Quick search yields this is possible with thunks. Maybe this will be a good topic for future articles.