Advanced Programming
4th Assignment: Affine Types
Deadline: 19 December 2025, 23:59

Programming Group
guido.salvaneschi@unisg.ch

University of St.Gallen

1 Assignment

In this assignment we will implement our own library for streams in Rust. The task is to complete the implementation of the `src/assignment.rs`. After you have completed the assignment, please submit your solution using the course website. You should modify and submit ONLY the `assignment.rs` file.

1.1 Background

A stream is a potentially infinite sequence of values. In languages like Java, streams are the implementation of lazy lists, where elements are computed on demand (i.e., lazy evaluation). In `src/lib.rs`, we designed the contract of our `Stream` library to provide a single method `next`, which evaluates and returns a `Some` of the next element of the stream, or `None` if the stream is exhausted.

By this definition, we can easily implement a `skip` function, that skips the first n elements of a stream, and a `vec`, which collects the elements of a stream into a `Vec` (vectors from the standard library). Next, you will be tasked to extend the library with more combinators. Note that one non-functional requirement of the library is to be very efficient, i.e., it should avoid unnecessary allocations and copies. To this end, you should **avoid at all costs cloning values via the `clone` method from the `Clone` trait**. Also, remember to **make your definitions public using the `pub` keyword**, otherwise they will not be accessible from the tests.

1.2 Natural Numbers

All starts with the natural numbers. So, let's start by implementing the stream of natural numbers to get some familiarity with the library interface.

Info: Note that `Stream` is a trait, i.e., the equivalent of a type-class in Haskell. Contrary to interfaces, type-classes cannot be used as types. Thus, we write the type `impl Stream<A>` to denote some existential type for which the `Stream<A>` trait is implemented, in particular a type that is uniquely determined at compile time (making it really efficient).

Task 1: Natural Numbers

Implement the `nats` function, which returns the stream of natural numbers, i.e. $[0, 1, 2, 3, ...]$.
Hint: You can define a `struct` and provide an implementation of the `Stream` trait for it, using `impl`.

1.3 Stream Combinators

Our library already provides some combinators to manipulate streams (`skip` and `vec`), however it is still very limited. First, let's implement the complement of `skip`, i.e., a `take` function.

Task 2: Take

Implement the `take` function, which takes the first n elements of a stream and then exhausts it. Use it to implement the stream of the first three natural numbers, i.e., $[0, 1, 2]$.
Hint: With generics, you can parametrize a new struct by the type of the inner stream, e.g., `struct Take<A, S: Stream<A>>`. The type checker requires all generics to be used in a field of the struct, so you might need to add some artificial fields of type `PhantomData<A>`. These are only used at compile time and optimized away before runtime.

Next, let's simplify the creation of streams. Note that a `Stream` has a single method, which means that any `Stream` can be implemented as a *closure*, i.e., a function plus some internal state (same as a lambda in Java or Scala). For many streams, the state is as simple as the last element produced. Thus, we define an helper function `generate` to create these simpler streams.

Task 3: Generate

Implement the `generate` function, which returns a stream whose elements are generated by repeatedly applying a generator function to the last element produced, starting from an initial element. Use it to re-implement the stream of natural numbers.
Hint: In Rust, closures are represented as type-classes of the form `Fn(A, B, C) -> D`, where `A, B, C` are the types of the inputs and `D` is the type of the output. As before, you can parametrize a new struct by the type of the closure.

Finally, let's implement a few more combinators to manipulate streams: one to transform the elements of a stream, one to discard them based on a predicate, and to collect them into a single result.

Task 4: Map

Implement the `map` function, which returns a stream whose elements are generated by applying a transformation function to the elements of another stream. Use it to implement the stream of squares, i.e., $[0, 1, 4, 9, ...]$.

Task 5: Filter

Implement the `filter` function, which returns a stream whose elements are generated by discarding the elements of another stream that do no satisfy a given predicate. Use it to implement the stream of even numbers, i.e., $[0, 2, 4, 6, ...]$.

Task 6: Aggregate

Implement the `aggregate` function, which returns a stream of aggregate results. Each result is obtained by composing the next element of another stream with the previous result, starting from an initial result. Use it to implement the stream of factorials, i.e., $[1, 1, 2, 6, 24, ...]$.

1.4 Fluent Style

At this point, our library is already quite useful, but it is still not very ergonomic to use and composition of combinators becomes difficult to read. For example, consider the following transformation:

```
filter(map(take(nats(), 5), |x| "a".repeat(x)), |s| s < "abc")
```

It would be easier to read if we adopted a *fluent style*, such that the operations that we apply are read left-to-right, instead of inside-out:

```
nats().take(5).map(|x| "a".repeat(x)).filter(|s| s < "abc")
```

With this style, our streams start to look much more like other standard stream libraries, e.g., Java Streams.

Task 7: Fluent Style

Define a trait `StreamOps` that **extends Stream** with methods for `skip`, `vec`, `map`, `filter`, and `aggregate`, so that they can be chained in fluent style. Provide an implementation of this trait for **ALL** Streams.
Hint: For user-defined traits, you can provide a *blanket implementation*, i.e., an implementation for all generic types that satisfy some constraints, e.g., `impl<A: Trait> OtherTrait for A`.

1.5 Hooking to the Standard Library

Finally, let's make our library interoperable with Rust's standard library. We already have a `vec` function to collect the elements of a stream into a `Vec`. Still, we are missing a way to create streams from vectors.

Task 8: Streamable... once.

Define the trait `StreamableOnce`. Implementors of this trait must provide a method `stream_once`, which consumes the implementor to return a stream of its elements. Provide an implementation of this trait for `Vec`. Use it to create the stream of magical numbers $[22, 13, 42]$.

By supporting conversions from vectors to streams and vice-versa, we also support, for example, generating vectors from other vectors through the stream combinators.

Task 9: Ownership and Borrowing

What is a limitation with `StreamableOnce`? Show an example of useful application where compilation fails because of ownership. Show another example of useful application where compilation fails because of borrowing.

1.6 Streamable... not only once! (Advanced)

To solve the limitation of `StreamableOnce`, we need to avoid consuming the implementor when calling `stream_once`. Let's approach a solution incrementally, we can try to define a new trait:

```
trait Streamable<A> { fn stream(&self) -> impl Stream<A> }
```

A call to this method would borrow the implementor without consuming it. This works fine, until we try to implement it for `Vec`. In fact, we would need to define a new struct that holds a *reference* to the vector to read its contents. Unfortunately, we cannot do that without introducing *lifetimes*.

Lifetimes are the way Rust tracks how long references are valid, or, in other words, when they will be deallocated. By this mechanism, the borrow checker can ensure that you will never dereference a dangling pointer. Every reference has a lifetime, but the compiler can usually infer most lifetimes automatically, so you rarely need to specify them. However, one particular case is when a struct contains a reference. A lifetime can be declared as a generic parameter prefixed by a tick, so the stream of a `Vec` would look like:

```
struct RefVecStream<'a, A> { source: &'a Vec<A>, /* other fields */ }
```

Now, the problem is that `RefVecStream` cannot be returned by `stream` in `Streamable`, because `'a` is never provided and the compiler cannot infer it. So, we need to adapt the trait `Streamable` to take a lifetime as a generic parameter:

```
trait Streamable<'a, A> { fn stream(&'a self) -> impl Stream<A> + 'a; }
```

This way, we are tying `'a` to the lifetime of the implementor, i.e., the `Vec`. In summary, `Vec` can now implement `Streamable` by returning a `RefVecStream`, whose lifetime `'a` is tied to the lifetime of the `Vec`.

Task 10: Streamable

Define the trait `Streamable` as described above. Provide an implementation of this trait for `Vec`. Show how this implementation overcomes the limitations in the examples of the previous task.