Exploring Rust

Mar 23, 2016 • nblumhardt

For the past few years I've followed the evolution of Rust with interest. Rust is a new systems programming language (like C or C++) with greater emphasis on safety. I'll spare you my reasons and opinions on its relevance since the Internet is replete with those :-)

Instead, I thought I'd write this brief post to share my perspective on learning Rust as a developer who's otherwise mostly writing C#. The post assumes you'll jump over and check out the Rust documentation where topics are necessarily summarised here.

No, this isn't one of those "I'm leaving .NET! …for Rust!" kinds of posts. Rust and C# are different enough to be complimentary, in much the same way C# and JavaScript are. Learning Rust also involves digesting concepts that have (so far) escaped C#'s all-consuming multi-paradigm expansion. Perhaps there are some ideas to bring back and apply in .NET, too?

So, you're comfortable with C# - how does the knowledge you already have map onto this new thing? Here are my notes after spending a few days writing some Rust.

Projects, builds and compilation

The first thing you'll probably write in Rust is not in the language itself, but at the command-line:

```
> cargo new --bin hello
...
> cd hello
> cargo run
...
Hello, world!
```

Cargo is Rust's package manager, and combined with the repository at crates.io, it plays the role of NuGet in the Rust ecosystem.

Unlike C# and .NET, Rust came of age in the post-NPM world. The Rust standard library is smaller than the original .NET one, and feels a lot more like the upcoming .NET Core. HTTP clients, web frameworks, async I/O and JSON serialization live out in the crates.io ecosystem where they can evolve and improve through Darwinian pressures.

You'll also probably note that `NuGet.exe` doesn't have a `run` command like the one above. This is another parallel between Rust and .NET Core: the `cargo` command-line (like `dotnet`) provides a standardised interface to the whole toolchain including dependency management, builds, a test runner and so-on.

Testing

In C#, adding unit tests to a project is a decision point. In Rust, the decision is made for you, and every (Cargo) project gets testing built in. Tests sit right alongside the code being tested, often in the same source file.

Here's an example:

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[test]
fn two_plus_two_makes_five() {
    let result = add(2, 2);
    assert_eq!(result, 5);
}
```

Side-by-side tests are not just a feel-good novelty: including tests in the *same compilation unit* as the code-under-test means there are no visibility issues to worry about. No need for `internal` and `[InternalsVisibleTo]` just to drive unit tests!

Code and its tests also always move together - no more parallel folder hierarchies in test projects to keep in sync. I like this enough to miss it in C# projects already. Conditional compilation in Rust means there's no issue with code or dependency bloat, something that might be trickier to handle cleanly in the .NET world.

The baked-in assertion macros are very basic - I'm sure there are alternatives but in my limited exploration so far I haven't felt the need for anything more.

Syntax

Rust has an expression-oriented curly-braces syntax. If you stare at C# (or C/C++, or Java, or JavaScript) all day, Rust isn't *that* surprising. It's not a language you'd immediately recognize by its syntax, like Python, Lisp or Smalltalk.

Coming at Rust from C# you might find the `name: Type` declaration order, `-> Type` return type declarations and implicit "return" unusual, but the tiny `add()` function above uses all of these and the meaning is pretty clear.

Approaching the Rust syntax is very much like approaching the syntax of any new language - it takes some time. So far nothing has left me scratching my head for too long.

Move semantics, lifetimes and borrowing

Here's where Rust begins to diverge from "nice, generic, modern language" to strike out on its own. Two of Rust's big selling points - memory safety without garbage collection, and freedom from data races - rely on statically determining the "owner" of an object as well as possible references to it (aliases).

```
let c1 = vec![1, 2, 3];
let c2 = c1;

// Fails to compile - the value has been moved out of c1 into c2
println!("Count: {}", c1.len());
```

You get the general idea in the example above: assignments in Rust are *moves* by default. This takes a bit of getting used to: in C#, assigments are always *copies* either of a simple value or a reference to an object, so after assignments in C#, both the source and destination variables are still usable.

For memory safety, the advantage of move semantics is that the compiler knows exactly when a value goes "out of scope" and can insert code to deallocate it at precisely the right moment, without needing a GC to scan for other references.

Now, this is obviously a pretty limited model, so Rust has a clever system of *lifetimes* and *borrowing* to make more complicated patterns expressible. A proper treatment of the idea is well beyond what I could attempt in this post (already starting to spill beyond the initial goal of "brief"), but with the risk of inadvertently bungling the analogy I'd like to try to translate the concept of a "lifetime" into C# terms, since I haven't seen this attempted elsewhere.

Ready? Read the ownership, borrowing, and lifetimes chapters of the Rust book? Ok, here it is.

Dispose-safety in C#

Let's imagine a typical use of the `using` statement in C#:

```
using (var archive = OpenZipArchive(@"C:\backup.zip"))
{
    using (var index = archive.OpenFile("index.json"))
    {
        var contents = ReadBackup(archive, index);
        // Read files from contents
    }
}
```

It's a bit contrived, but you can imagine the object returned from `ReadBackup(archive, index)` *may* have a reference to either `archive` or `index`.

That's fine in the example above, since we know at the time we `// Read files from contents` that nothing has been disposed yet, so all is well.

How about this code?

```
using (var archive = OpenZipArchive(@"C:\backup.zip"))
{
    BackupReader contents;
    using (var index = archive.OpenFile("index.json"))
    {
        contents = ReadBackup(archive, index);
    }
    // Read files from contents
}
```

Here, we might have a problem. The *lifetime* represented by the second `using` statement has finished, and `index` has been disposed. If the `BackupReader` holds a reference to `index` then we'll have trouble, probably in the form of an `ObjectDisposedException` when we try to use it.

How can we tell if this code is safe? In C#, we can't - statically - without examining the code of `ReadBackup()` and potentially any code it calls.

The analogy is not precise, since in Rust it's deallocating the memory rather than calling `Dispose()` that we're concerned with, but otherwise, this is the safety issue Rust lifetimes solve. The Rust compiler *must* know, when examining this code, whether the second usage of `contents` is valid, because allowing it otherwise could lead to reading from memory that's already been freed.

Before translating the example into Rust, let's invent a hypothetical dispose-safety checker for C#, where we can represent these kinds of things in method signatures using attributes. Here's the original:

```
static BackupReader ReadBackup(Archive archive, Index index)
{
    var manifestFilename = index.GetManifestFilename();
    return new BackupReader(archive, manifestFilename);
}
```

Ah - so the `index` argument is only used to determine a filename, and the result only depends on `index`. Here's the "dispose-safe C#" that expresses this:

```
[return: In("a")]
static BackupReader ReadBackup([In("a")] Archive archive, [In("b")] Index index)
{
    var manifestFilename = index.GetManifestFilename();
    return new BackupReader(archive, manifestFilename);
}
```

We've added some labels like `[In("a")]` to the arguments. These represent **which `using` block the value came from**. "Dispose-safe C#" requires these whenever a method accepts an `IDisposable` argument.

Annotating the return value with `[In("a")]` means that the result is valid in the same scope that the `archive` parameter is valid in - both are tagged with `a`.

The return value doesn't have any relationship at all to `index`, so the `b` tag doesn't appear anywhere else.

Back at the call-site, the checker does the mapping of labels to `using` blocks for us implicitly:

```
// 1
using (var archive = OpenZipArchive(@"C:\backup.zip"))
{
    BackupReader contents;

    // 2
    using (var index = archive.OpenIndex())
    {
        // Infers that "a" is block 1 and "b" is block 2
        contents = ReadBackup(archive, index);
    }

    // Thus, contents depends only on block 1, so this is fine
    // Read files from contents
}
```

Now, what would have happened if `ReadBackup()` had a different implementation?

```
[return: In("b")]
static BackupReader ReadBackup([In("a")] Archive archive, [In("b")] Index index)
{
    var files = archive.LoadAllFiles();
    return new BackupReader(files, index);
}
```

In this version of the method, the returned `BackupReader` is declared to depend on `b`, the lifetime of the `index` argument. The mapping of the `using` statements is the same, but the code is no longer valid and the checker will reject it:

```
// 1
using (var archive = OpenZipArchive(@"C:\backup.zip"))
{
    BackupReader contents;

    // 2
    using (var index = archive.OpenIndex())
    {
        // Infers that a is block 1 and b is block 2
        contents = ReadBackup(archive, index);
    }

    // Error: contents depends on block 2, which has ended
    // Read files from contents
}
```

By giving the checker a bit more information about the intentions of the code, it's helpfully saved us from an `ObjectDisposedException` at runtime — awesome!

Lifetime variables in Rust

I hope this little "dispose-safe C#" thought experiment sheds some light now on the corresponding (memory-safe) Rust:

```
fn read_backup<'a,'b>(archive: &'a Archive, index: &'b Index) -> BackupReader<'a> {
    let manifest_filename = index.manifest_filename();
    BackupReader::new(archive, manifest_filename)
}
```

Don't let the reuse of the angle-brackets "generic" syntax throw you. The little `'a` and `'b` annotations in there are lifetime variables, exactly the same as the `[In("a")]` and `[In("b")]` examples in C#.

At the call-site, there's no special `using`-like statement to delineate lifetimes in Rust, which are implied by the natural boundaries of blocks:

```
// 1
let archive = open_zip_archive("C:\\backup.zip");
let contents: BackupReader;

// 2
{
    let index = archive.open_index();
    contents = read_backup(&archive, &index);
}

// Read files from contents
```

When the compatible implementation of `read_backup()` is used, this compiles fine; when the second implementation of `read_backup()` is used here, the Rust compiler rejects the code with `error: index does not live long enough`.

I hope my analogy hasn't stretched too far and led you astray, as my own understanding of Rust is far from perfect. My hope is that it makes the whole *lifetime* concept a bit more approachable. Ownership, borrowing and lifetimes are interesting concepts that it are worth spending some time on.

The stack and the heap

There's one more place I've spotted where C# and Rust differ fundamentally, and that's how memory is allocated between the *stack* and the *heap*.

C# has two families of data structures that dermine allocation behaviour: `struct`s and `class`es. If a type is a struct then values of that type will be allocated on the stack*:

```
// Here, now is the actual DateTime value
var now = new DateTime(2016, 03, 23);
```

(**It's been called out that this is an oversimplification - `struct` values in C# can end up in registers or embedded within heap-allocated values, too.*)

If a type is a class, then values of that type are allocated on the heap, and we work with them through references:

```
// Here, user is a pointer to some memory holding a User
var user = new User("nblumhardt");
```

C# also has an "escape hatch" for moving structs to the heap as well, in the form of 'boxing':

```
// Here, now is a pointer to memory holding the DateTime
var now = (object)new DateTime(2016, 03, 23);
```

Boxed structs in C# however are a bit of a second-class citizen - there's no static type to represent them, so they end up as `object`s.

After the last section, you might feel some relief to hear that Rust is simpler on this point. In Rust, all values are essentially C# structs:

```
// Here, user is the actual User value
let user = User::new("nblumhardt");
```

To move a value onto the heap, as in C#, the value is boxed:

```
// Here, user is a Box, pointing to memory on the heap
let user = Box::new(User::new("nblumhardt"));
```

Boxed values are first-class in Rust, so in practice they're more like C#'s classes in usage, enjoying the benefits of static typing for example.

The declaration and construction syntax for structs in Rust is quite a bit different from the C# equivalent, but I've found that getting clear about the differences in the allocation model has made the biggest difference to my understanding of Rust code.

Language machinery

Rust is a modern language. Its designers have drawn widely from existing languages and experience to exclude the worst footguns (`null`able references, mutability-by-default) and include the best available tools for abstraction (closures, generics, traits, pattern matching, algebraic data types and more).

One of the refreshing aspects of Rust is that like C#, it's a multi-paradigm language with a functional features. Many of the new languages I've encountered in recent years are presented as functional languages, which is nice enough, but there's a place in the future for variety.

I'd love to continue poking at different corners of Rust with this post, but writing it's already pushing at the limits of my attention, so before I lose your interest too I'd better wrap up.

In closing

I'm having a great time exploring Rust. If you're working in one of the currently popular industrial languages like C#, and want to learn something that's different in more than just syntax, Rust could be what you're looking for. Have fun!

Discuss this post on Hacker News

Nicholas Blumhardt

Nicholas Blumhardt
nblumhardt@nblumhardt.com
nblumhardt
nblumhardt