RAII is better than the bracket pattern

October 8, 2018

See a typo? Have a suggestion? Edit this page on Github (https://github.com/snoyberg/snoyman.com-content/edit/master/posts/raii-better-than-bracket-pattern.md)

I recently wrote an FP Complete (https://www.fpcomplete.com/) blog post entitled ResourceT: A necessary evil (https://www.fpcomplete.com/blog/2018/10/resourcet-necessary-evil). I had a line in that post I wanted to expand upon:

> I believe this is an area where the RAII (Resource Acquisition Is Initialization) approach in both C++ and Rust leads to a nicer solution than even our bracket pattern in Haskell, by (mostly) avoiding the possibility of a premature close.

(/static/io-thief.jpg)

To demonstrate what I'm talking about, let's first see an example of the problem in Haskell. First, the good code:

```
import Control.Exception (bracket)

data MyResource = MyResource

newMyResource :: IO MyResource
newMyResource = do
  putStrLn "Creating a new MyResource"
  pure MyResource

closeMyResource :: MyResource -> IO ()
closeMyResource MyResource = putStrLn "Closing MyResource"

withMyResource :: (MyResource -> IO a) -> IO a
withMyResource = bracket newMyResource closeMyResource

useMyResource :: MyResource -> IO ()
useMyResource MyResource = putStrLn "Using MyResource"

main :: IO ()
main = withMyResource useMyResource
```

This is correct usage of the bracket pattern, and results in the output:

```
Creating a new MyResource
Using MyResource
Closing MyResource
```

We're guaranteed that `MyResource` will be closed even in the presence of exceptions, and `MyResource` is closed after we're done using it. However, it's not too difficult to misuse this:

```
main :: IO ()
main = do
  myResource <- withMyResource pure
  useMyResource myResource
```

This results in the output:

```
Creating a new MyResource
Closing MyResource
Using MyResource
```

We're still guaranteed that `MyResource` will be closed, but now we're using it *after* it was closed! This may look like a contrived example, and easy to spot in code. However:

1. Many of us choose Haskell because it forces programming errors to be caught at compile time. This is an error which is definitely *not* caught at compile time.
2. In larger code examples, it's easier for this kind of misuse of the bracket pattern to slip in and evade notice.

Rust

By contrast, with the RAII approach in Rust, this kind of thing can't happen under normal circumstances. (Sure, you can use unsafe code, and there might be other cases I'm unaware of.) Check this out:

```
struct MyResource();

impl MyResource {
    fn new() -> MyResource {
        println!("Creating a new MyResource");
        MyResource()
    }

    fn use_it(&self) {
        println!("Using MyResource");
    }
}

impl Drop for MyResource {
    fn drop(&mut self) {
        println!("Closing MyResource");
    }
}

fn main() {
    let my_resource = MyResource::new();
    my_resource.use_it();
}
```

Despite looking closer to the bad Haskell code, it still generates the right output:

```
 Creating a new MyResource
 Using MyResource
 Closing MyResource
```

In fact, try as I might, I cannot figure out a way to get this to close the resource before using it. For example, this is a compile error:

```
fn main() {
    let my_resource = MyResource::new();
    std::mem::drop(my_resource);
    my_resource.use_it();
}
```

And this works just fine:

```
fn create_it() -> MyResource {
    MyResource::new()
}
fn main() {
    let my_resource = create_it();
    my_resource.use_it();
}
```

The life time of the value is determined correctly and only dropped at the end of `main`, not the end of `create_it`.

Fixing it in Haskell

UPDATE In my efforts to make this as simple as possible, I included a “solution” that’s got a massive hole in it. It still demonstrates the basic idea correctly, but if you want *actual* guarantees, you’ll need to include the phantom type variable on the monad as well. See Dominique’s comment (http://disq.us/p/1we3o1q) for more information.

We can use a similar approach as the `ST` strict state transformer via a phantom variable to apply a “lifetime” to the `MyResource`:

```
{-# LANGUAGE RankNTypes #-}
import Control.Exception (bracket)

data MyResource s = MyResource

newMyResource :: IO (MyResource s)
newMyResource = do
  putStrLn "Creating a new MyResource"
  pure MyResource

closeMyResource :: MyResource s -> IO ()
closeMyResource MyResource = putStrLn "Closing MyResource"

withMyResource :: (forall s. MyResource s -> IO a) -> IO a
withMyResource = bracket newMyResource closeMyResource

useMyResource :: MyResource s -> IO ()
useMyResource MyResource = putStrLn "Using MyResource"

main :: IO ()
main = withMyResource useMyResource
```

Now the bad version of the code won’t compile:

```
main :: IO ()
main = do
  myResource <- withMyResource pure
  useMyResource myResource
```

Results in:

```
Main.hs:22:32: error:
    • Couldn't match type ‘s0’ with ‘s’
        because type variable ‘s’ would escape its scope
      This (rigid, skolem) type variable is bound by
        a type expected by the context:
          forall s. MyResource s -> IO (MyResource s0)
        at Main.hs:22:17-35
      Expected type: MyResource s -> IO (MyResource s0)
        Actual type: MyResource s0 -> IO (MyResource s0)
    • In the first argument of ‘withMyResource’, namely ‘pure’
      In a stmt of a 'do' block: myResource <- withMyResource pure
      In the expression:
        do myResource <- withMyResource pure
           useMyResource myResource
   |
22 |   myResource <- withMyResource pure
   |                                 ^^^^
```

This approach isn't used much in the Haskell world, and definitely has overhead versus Rust:

- You need to add a type variable to all resource types, which gives more things to juggle
- It's non-trivial to deal with promoting values beyond their original scope (like we did in Rust with passing outside of the `create_it` function)

As I mentioned in the original blog post, regions (https://github.com/basvandijk/regions#readme) based on a paper by Oleg (http://okmij.org/ftp/Haskell/regions.html#light-weight) explores this kind of idea in more detail.