Universität Bielefeld
Technische Fakultät

AG Genominformatik
Dr. Lucas Robidou

The Rust programming language
Summer 2024 / 2025

Exercises

1 The typestate builder pattern

Imagine having a struct like this:

```
struct Sender {
    url: String,
    key: u64,
}
```

This could be used to store a URL to a web service and an API key for authentication.

URLs have to be encoded[1]. This is not the focus of this exercise, but let's imagine that setting fields require expensive runtime computation. Hence, you want to indicate to your library's users that some fields should be set only once.

If you don't want to pollute the API of Sender, one way to do it is to use the builder pattern:

```
impl Sender {
    /// Creates a new SenderBuilder.
    /// By setting the fields of the builder one by one,
    /// we can have fine-grain control of the logic of the setters.
    pub fn builder() -> SenderBuilder {
        SenderBuilder::new()
    }
}

struct SenderBuilder {
    url: Option<String>,
    key: Option<u64>,
}

impl SenderBuilder {
    pub fn new() -> SenderBuilder {
        SenderBuilder {
            url: None,
            key: None,
        }
    }

    /// Build a new Sender with the info from the builder
    pub fn build(self) -> Sender {
        Sender {
            url: self.url.expect("should have set url"),
            key: self.key.expect("sohould have set key"),
        }
```

[1] https://en.wikipedia.org/wiki/Percent-encoding

```
    }

    /// Set the URL in the builder
    pub fn set_url(self, url: String) -> SenderBuilder {
        SenderBuilder {
            url: Some(url),
            key: self.key,
        }
    }

    /// Set the key in the builder
    pub fn set_key(self, key: u64) -> SenderBuilder {
        SenderBuilder {
            url: self.url,
            key: Some(key),
        }
    }
}
```

Note that in set_url, you could now check if the URL is already set and panic (or return an Option of the builder) to gracefully handle the error. But it would be better to detect at compile time such incorrect programs, so let's encode that detection in the type system.

The trick is, as usual, to use generics:

```
struct SenderBuilder<U, K> {
    url: U,
    key: K,
}
```

We now need the types to put in U and K:

```
// URL states
struct NoUrl;
struct Url {
    data: String,
}

// key states
struct NoKey;
struct Key {
    data: u64,
}
```

Finally, we can pull something like this:

```
impl<K> SenderBuilder<NoUrl, K> {
    pub fn set_url(self, url: String) -> SenderBuilder<Url, K> {
        SenderBuilder::<Url, K> {
            url: Url { data: url },
            key: self.key,
        }
    }
}
```

Let's take a moment to analyse what happens here.

- `impl<K>`: this impl block will refer to any given type K

- `SenderBuilder<NoUrl, K>` the impl block is only for SenderBuilder for which the first actual type is NoUrl, the second one being any type K
- `set_url(self, url: String) -> SenderBuilder<Url, K>` the return type is a SenderBuilder with Url as the first type, the second type being K, the same as the input parameter

Basically, when setting the URL field, we switch from the NoUrl struct to the Url struct. Since this method is not implemented for `SenderBuilder<Url, K>`, you cannot call it twice.

1. Rewrite the builder using generics, so that the user can only set both fields once.
2. Write a `new` method to create a SenderBuilder with no field set.
3. Make the build method only available when every field has been set.

2 A genetic algorithm

Last week, you wrote a travelling salesman algorithm that generates random solutions. Let's implement a genetic algorithm to find a good approximation of the best path in the graph.

Genetic algorithms are based on the ideas of natural selection and genetics. A population of candidate solutions is evolving with a selection pressure that favors the better solution.

- The fittest individuals reproduce by merging their genomes
- Individuals can receive random mutations to their genome
- Random individuals are regularly introduced to prevent genetic problems (i.e., getting stuck on a local minimum)

1. Implement a `fn merge(&self, other: &Path) -> Path` method for Path, that creates a new path such that
    - The first half of the path is copied from `self`
    - The second half of the path is copied from `other`
    - Duplicated cities are removed, and absent cities are added

Now, you may have something similar to:

```
impl Path {
    pub fn generate(map: &Map) -> Path;
    pub fn merge(&self, other: &Path) -> Path;
    pub fn mutate(&mut self);
    pub fn score(&self, map: &Map) -> f64;
}
```

- Write a Solver struct that contains a vector of Path and a Map.
- Implement a compute method for Solver that:
    - conceptually splits that vector into 4 consecutive parts of the same size (let's name them A, B, C, and D)
    - keeps A untouched
    - fills D with new random paths
    - merges solutions from A and B, two by two, to overwrite part C
    - mutates every path in B

  - sorts the vector of paths according to their score
  - returns a copy of the best path in the vector, and its score

- Write a main function that creates a random Map, creates a Solver, and calls compute 10,000 times. At each iteration, print the best score in a file.

Hint: to sort a vector of Path, you can use

```
fn sort_vec_of_path(v: &mut Vec<Path>, map: &Map) {
    v.sort_by(|x, y| {
        x.score(&map)
            .partial_cmp(&y.score(&map))
            .expect("one score is NaN")
    })
}
```

2.1 Bonus

- Write a Python (or R) program to print and display the score with regard to the number of iterations.
- Write a Tentative trait, with methods generate, merge, mutate, and score.
- Implement Tentative for Path.
- Rewrite the Solver so that it works for any Tentative. At this point, your solver can solve a wide range of problems.