Advanced Programming
5th Exercise: Affine Types and Dependent Types

Programming Group
guido.salvaneschi@unisg.ch

University of St.Gallen

1 Goals

In this exercise, you'll practice affine types in Rust for safe memory management without garbage collection, exploring the concepts of ownership and borrowing. Additionally, we'll briefly have a look at dependent types in Idris2, which increase the expressiveness of types system even further, allowing you to enforce stronger program invariants at compile time.

2 Affine Types in Rust

Rust is a systems programming language that guarantees memory safety without a garbage collector, through a sophisticated affine type system that enforces strict ownership and borrowing rules at compile time.

2.1 Ownership and Borrowing

Ownership is a central concept in Rust that ensures memory safety by enforcing a single owner for each value. Ownership can be transferred through **move semantics**: by using variables by value in functions or assignments the ownership of the value is transferred to the function or the new variable. When the owner goes out of scope, the value is automatically deallocated.
Multiple parts of the code can access the same value through **borrowing**, which is done via references. References can be either immutable or mutable: multiple immutable references are allowed, but only one mutable reference is allowed at a time (to prevent data races in concurrent programs).
We are going to explore these concepts by implementing a trie (Figure 1). Open the folder `affine-types-in-rust` with VSCode.

Info: A **Trie** is a tree data-structure to store strings, optimized to search for all strings that match a given prefix. The edges in a trie are labeled by characters, so that a word $c_1 \cdots c_n$ is represented as the path from the trie's root along the edge c_1 until the edge c_n. If such an edge exists, then we can enumerate all the words that have that prefix. To know that a word $c_1 \cdots c_n$ exists in the trie, we attach to each node in the trie a boolean flag representing the end of a word.

Task 1: Structs

In Rust, you can define custom data types using `struct`:

```
struct Point {
  x: f64,
  y: f64,
}
```

Implement the necessary structures for a trie. Start by defining a `Node` structure, and then the `Trie` structure itself (the root `Node`).

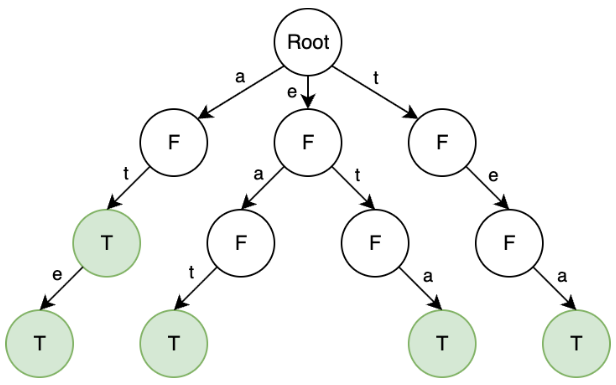

Figure 1: A trie containing the words "at", "ate", "eat", "eta", "tea". Words are nodes colored in green, meaning that the end-of-word flag is true.

Task 2: Associated Functions

In Rust, you can define functions associated with struct types, in a way similar to methods:

```
impl Point {
  fn new(x: f64, y: f64) -> Point { Point { x, y } }   // static method
  fn to_pair(self) -> (f64, f64) { (self.x, self.y) }  // instance method
}
```

Note how associated functions act like “static methods” if no `self` argument is present, and like “instance methods” if `self` is present.
Implement two static factory methods: `Node::new` creates an empty `Node`; `Trie::new` creates an empty `Trie`.

Task 3: Ownership and Borrowing

Depending on the type of parameters of your methods, Rust will enforce ownership, immutability, or mutability:

```
impl Point {
  fn get_x(&self) -> f64 { self.x }        // immutable borrow: read-only
  fn set_x(&mut self) { self.x = 0; }      // mutable borrow: read-write
  fn free(self) { std::mem::drop(self); } // immutable ownership: read-free
  fn dealloc(mut self) { self.x = 0; }    // mutable ownership: read-write-free
}
```

The `&` and `mut` modifiers can be applied to any type. Ownership gives full control over the value, so you can always switch between mutable and immutable ownership, unlike borrowing. Owned values are also deallocated automatically when they go out of scope.
Implement a method `add_word` that adds a word to a `Trie`.

Task 4: Has Prefix

Implement a method `has_prefix` that checks if there are strings that match a given prefix in the `Trie`.

Task 5: Get All Words

Implement a method `get_all_words` that returns all the words that match a given prefix in the `Trie`.

3 Dependent Types in Idris2

Idris2 is a functional programming language with full dependent types.

Info: In a simple type system, *terms depends on other terms* and there are only concrete types. These type systems have evolved in three dimensions and their combinations (see **Lambda Cube**):

- **Polymorphic Types** allow user-defined *terms to depend on types*;
- **Higher-Kinded Types** allow user-defined *types to depend on types*;
- **Dependent Types** allow user-defined *types to depend on terms*.

Dependent types enable more expressive type systems that can capture complex invariants and properties of programs at the type level. Idris2 has all of these features.

3.1 Type-Safe Formatter

In the previous exercise, we've seen how to implement a type-safe format function in Typescript. Recall that we had to define a mini-language through type-level functions to be executed at compile time in order to do this. This was not very pleasant and the irony is that for a type-safe language, programming at the type level is not safe at all!

With dependent types, we don't need to implement a mini-language because, after all, we're using one! In fact, dependent types allow us to mix expressions and types together in the same language allowing definitions from both worlds to interact seemlessly.

Task 6: Type-Safe Formatter

Open the file `01.idr`. Let's look at the implementation of a type-safe `format` function in Idris2.
If you don't have Idris2 installed, you can use this unofficial online playground and REPL to test your code: `https://dunhamsteve.github.io/idris2-playground/`