Elementary Algorithms Solutions

Rahul Chhabra

2022-03-13

Contents

About

Solutions to the text “Elementary Algorithms”.

For fun, non-serious learning project.

The text is particularly cool because it uses both functional and imperative techniques. I’ve been wanting to get better at FP for a while - so why not actually *do something* in it?

Other options include using Elm for building an HTML UI (not particularly interesting if you don’t already have a good idea), using F# and Suave to build servers on ASP.NET (requires more knowledge of Kleisli categories and monads than I currently possess), or write a small computer algebra system (something I plan on doing *eventually*).

Studying a few standard data structures is a particularly good option then.

Solutions are available online on my Github pages.

Chapter 1

Lists

Firstly, we present a few perspectives to think about lists :

1.1 High-level Declarative Perspective

We'd pick a functional language for this, and this is where algebraic types come in :

```
type List<'a> = Empty | Node of 'a * List<'a>
```

The `|` operator represents disjoint union and `*` represents cartesian product.

1.2 High-level Imperative Perspective

Let's say we're in a garbage collected language with references and inheritance.

We can easily do this in a language like Kotlin :

```
sealed class List<T> {
    class Empty<T> : List<T>()
    class Node<T>(val data: T, val next: List<T>): List<T>()
}
```

This is not as pretty as the Declarative Perspective but at least there's pattern matching!

```
when (list) {
    is List.Node<Int> -> list.data
    is List.Empty<*> -> null
} // return type of when is `Int?`
```

1.3 Low-level Imperative Perspective

Let us start by implementing a simple list imperatively, in Rust.

```rust
struct List<T> {
    data: T,
    next: Box<List<T>>
}
```

This code is not particularly generic, making it for more generic requires the use of lifetimes :

```rust
struct List<T, 'a> {
    data: T,
    next: &'a List<T, 'a>
}
```

Also, we have currently used one single lifetime `'a` for all nodes in our list. We might want to change it later.

Do note that this `List` is immutable. This is intentional : by default, everything in Rust is immutable.

We can make it mutable as follows :

```rust
struct List<T, 'a> {
    data: T,
    next: &mut 'a List<T, 'a>
}
```

We can also implement this type algebraically :

```rust
enum List<T, 'a> {
    Empty,
    Node {
        data: T,
        next: &'a List<T, 'a>
    }
}
```

Also, we are currently constrained in using only one kind of reference (native `&` and `&mut`). In reality, we would want our `List` to be *polymorphic* over all possible kinds of references.

We can do this with Rust's `Deref` trait.

```rust
enum List<T> {
    Empty,
    Node {
        data: T,
        next: dyn Deref<Target = List<T>>
    }
}
```

The low-level implementation certainly requires a lot more work due to the lack of the garbage collector!

1.4 Exercise 1.2

> For list of type A, suppose we can test if any two elements x, $y \in A$ are equal, define an algorithm to test if two lists are identical.

First we define a recursive algorithm with HLFP :

```
let rec Equals a b elementEquals =
    match a with
    | [] -> match b with
        | [] -> true
        | head::tail -> false
    | aHead::aTail -> match b with
        | [] -> false
        | bHead::bTail -> (elementEquals aHead bHead) && (Equals aTail bTail elementEquals)
```

This algorithm is elegant and readable but suffers from consuming $O(n)$ stack space.

Let us make it *tail-recursive*!

```
let Equals a b elementEquals =
    let rec Loop a b result =
        if !result then result else
            match a with
            | [] -> match b with
                | [] -> true
                | head::tail -> false
            | aHead::aTail -> match b with
                | [] -> false
                | bHead::bTail -> Loop aTail bTail (result && (elementEquals aHead bHead))
    Loop a b true
```