

MASTER THESIS
COMPUTER SCIENCE



RADBOD UNIVERSITY

Task-Oriented Programming in Rust

Author:

Rick van der Wal
s1005618

First supervisor/assessor:

Pieter Koopman
pieter@cs.ru.nl

Second assessor:

Peter Achten
p.achten@cs.ru.nl

August 17, 2022

Abstract

Task-Oriented Programming (TOP) is a programming paradigm that allows users to specify complex workflow applications in a declarative style. This is done through the concept of *tasks*. There are primitive tasks for simple operations, as well as task combinators to construct more complex tasks. TOP also features a mechanism that allows sharing values between tasks. Currently, TOP is implemented as a shallowly embedded, domain-specific language in Clean – a lazy, purely functional programming language. In this thesis, we explore how it can be implemented in the multi-paradigm programming language Rust, and what benefits and drawbacks it would bring. We conclude that all components of a TOP library can be implemented in Rust, resulting in a useful and feature-complete library. However, it does come with limitations: sharing data between tasks proves to be quite difficult due to Rust’s ownership system. Moreover, while Rust could theoretically outperform Clean due to lower-level memory control, it currently has language limitations that prevent it from doing so. Finally, the resulting library is less flexible in the types it can support when compared to the Clean implementation.

Contents

1	Introduction	3
1.1	Research Question	4
1.2	Outline	4
2	Task-Oriented Programming	5
2.1	How TOP works conceptually	5
2.2	iTasks: TOP in Clean	6
3	The Rust Programming Language	9
3.1	Variables, Ownership, and Mutability	9
3.2	Borrow Checker	11
3.3	Data Types	12
3.4	Functions and Methods	14
3.5	Generics	16
3.6	Traits	16
3.6.1	Associated Types	18
3.6.2	Common Traits	19
3.6.3	Trait Objects	20
3.7	Macros	21
3.7.1	Declarative Macros	21
3.7.2	Procedural Macros	21
3.8	Closures	22
3.9	Asynchronous Programming	23
4	Initial Implementation	24
4.1	Public API	25
4.1.1	View Tasks	26
4.1.2	Edit Tasks	27
4.1.3	Parallel Tasks	28
4.1.4	Sequential Tasks	29
4.2	Task Abstraction	30
4.2.1	Design Choices and Alternatives	31

4.3	View Tasks	32
4.3.1	Implementation	32
4.3.2	Design Choices and Alternatives	34
4.4	Edit Tasks	35
4.4.1	Implementation	35
4.4.2	Design Choices and Alternatives	42
4.5	Parallel Tasks	43
4.5.1	Implementation	43
4.5.2	Design Choices and Alternatives	46
4.6	Sequential Tasks	47
4.6.1	Implementation	48
4.6.2	Design Choices and Alternatives	53
4.7	Executor	59
5	Analyzing Drawbacks and Improving the Library	63
5.1	Borrowing Task Values	63
5.2	Macros	66
5.2.1	Declarative Macros	66
5.2.2	Procedural Macros	69
5.3	Shared Data Sources	77
5.3.1	Sharing Data in Rust	77
5.3.2	Implementing Shared Data Sources	78
5.3.3	Supporting Shares for Advanced Tasks	85
6	Demonstration	93
6.1	Implementation	93
6.2	Output	98
7	Conclusion	99
7.1	Future Work	100
	References	102

Chapter 1

Introduction

Task-Oriented Programming (TOP) is a programming paradigm that revolves around the concept of *tasks*. There are primitive tasks, such as for entering or viewing data, as well as combinators to compose more complex programs in a simple, declarative manner. The TOP framework can then automatically generate a suitable front-end to interact with such a task, allowing programmers to focus on business logic alone.

One implementation of TOP is called iTasks. It is currently written as a shallowly embedded, domain-specific language in the functional, general-purpose programming language Clean. Clean is a good fit for task-oriented programming, considering the monadic style of the combinators. However, Clean is not widely adopted, and TOP has not gotten as much traction as it should. We believe other programming languages can benefit greatly from TOP as well. However, this requires more research on whether a system that heavily relies on functional programming concepts can be ported to other languages effectively. Imperative and object-oriented programming languages might struggle to support all of it.

In this thesis, we will implement a TOP library in the Rust programming language¹. Rust is a general-purpose, *multi-paradigm* programming language that aims to be fast and safe. It features a few concepts that are usually associated with functional languages, but also aspects that are closer to imperative languages, such as mutability. In that regard, it can be considered a bridge between the functional paradigm and the more popular imperative and object-oriented paradigms, and therefore a good candidate for a new TOP library.

This research was performed at Tweede Golf², a software company located in Nijmegen. They use Rust for many of their projects, and want to contribute to its research and open-source software.

¹<https://www.rust-lang.org/>

²<https://tweedegolf.nl/en>

1.1 Research Question

We aim to answer the following research question in this thesis:

What benefits and drawbacks does the Rust programming language bring to Task-Oriented Programming?

Our approach is to implement a TOP library in Rust, and document the benefits and drawbacks we encounter.

1.2 Outline

This thesis is structured as follows:

- In the next chapter, we will explain what Task-Oriented Programming is and what components are necessary for a feature-complete implementation.
- Chapter 3 provides an introduction to the Rust programming language, and highlights several aspects of the language that are relevant when implementing a TOP library.
- Chapter 4 describes the initial prototype of a Task-Oriented Programming library in Rust, along with all the roadblocks we encountered during its implementation.
- We improve the existing implementation with features that work differently in Rust compared to other programming languages in chapter 5. Shared data sources and drawbacks that are specific to – or more prominent in – the Rust implementation are also discussed here.
- Chapter 6 contains a nontrivial example program built with the resulting library. It was commissioned by Tweede Golf and demonstrates that the library is suitable to tackle real-world problems.
- Finally, we draw our conclusions and answer the research question in chapter 7.

Chapter 2

Task-Oriented Programming

Task-Oriented Programming (TOP) was first introduced in a paper by Plasmeijer, Achten, and Koopman (2007). They describe it as a programming paradigm that is especially useful for developing applications in which many people collaborate. These applications are quite complex to build, because they have to handle concurrent inputs from arbitrarily many devices, possibly running different operating systems.

Handling the communication between the client(s) and server, and creating intuitive user interfaces to interact with it, is time-consuming and prone to errors. TOP aims to simplify that process by allowing programmers to specify the workflow of their systems at a high level of abstraction. The TOP framework can then automatically generate a suitable user interface for interacting with the system, and handle all the user's input appropriately.

2.1 How TOP works conceptually

TOP revolves around the concept of tasks. A program following the TOP paradigm usually consists of one or more tasks that act as its workflow specification. They describe exactly how the system should behave, from what is shown to the users, to the calculations performed on the server, and the communication in between.

Tasks can be subdivided into two main categories: primitive and composed. Primitive tasks represent a basic operation, such as having the user **view** a value, or asking them to **edit** a value instead. There are also primitive tasks without user interaction, such as reading a file or retrieving a value from a sensor. Composed tasks are constructed from other tasks using *combinators*. For example, a **parallel** task can execute multiple subtasks at the same time, and a **sequential** task can execute multiple tasks after each other. Primitive tasks and combinators together allow the programmer to specify complex workflows in a declarative manner.

Types play an important role in task-oriented programming. All tasks hold an interme-

diate value of a certain type. This value can be stable, unstable, or empty. Unstable values might still change over time, while stable values are guaranteed to be static. The aforementioned sequential tasks can observe the intermediate value of a task to determine whether it should create a followup task with it. For instance, one might define a task that first shows a checkbox to the user. When the user checks the box, its task value changes to an unstable ‘true’. This can then trigger the continuation of a sequential task, which makes it transition to a new task that displays the entered value.

Tasks can share values with other tasks using Shared Data Sources (SDS, or ‘share’). Examples of shares are files, databases tables, sensors, the current system time, and simple values within the program. The value of a shared data source is automatically synchronized between all tasks that use it, allowing the programmer to define more flexible and responsive tasks. A simple use case for shares would be to have the user enter a value, and simultaneously show that value to them. Note that not all shared data sources have to be both readable and writable: read-only shares are sufficient for displaying values, for instance.

Ideally, the intermediate values of tasks can be any type that makes sense to use in the context of the application. However, this is not always straightforward. Tasks that show data to the user, or request the user to enter data, have to be turned into a user-readable format in some way. This can be difficult for complex types.

2.2 iTasks: TOP in Clean

The most complete implementation of TOP to date is the iTask system, or iTasks for short (Plasmeijer, Achten, Lijnse, & Michels, 2011). It is implemented as shallowly embedded domain-specific language in the purely functional programming language Clean (Koopman, Plasmeijer, van Eekelen, & Smetsers, 2001). There exist prototypes of TOP implementations in other languages, namely Python (Lijnse, 2022) and Lua (van Gemert, 2022). However, since those are relatively new, we will mainly be comparing with iTasks in this thesis.

Internally, iTasks features the primitive `interactR` and `interactRW` tasks to interact with the user. There are three common ways to construct these tasks:

1. `viewInformation [] "Hello, world!"` shows a value, in this case the string “Hello, world!”, to the user.
2. `updateInformation [] False` requests the user to change a value. This particular example translates the boolean to an unchecked checkbox.
3. `enterInformation []` is similar to `updateInformation`, but without an initial value. The type of the intermediate value, which determines the front-end representation, is retrieved from the context. For instance, the following definition creates an empty list of numbers:

```
task1 :: Task [Int]
task1 = enterInformation []
```

iTasks translates these interactive tasks to a web representation: `viewInformation` displays values as text, and the other two as `<input>` elements. To make this work for any type, iTasks uses generic functions that automatically derive a suitable web-representation for a type by analyzing its structure. Note that TOP in general is not limited to web based interfaces. It is possible to support other ways of interacting with the system, such as through a mobile application or command-line interface.

Another notable primitive task in iTasks is `return`, which always contains a stable value. Since interaction tasks always produce unstable values, due to the fact that the user can still change them, `return` is often used to turn unstable values into stable ones.

Moving on to the composed tasks, iTasks features the `parallel` task to execute multiple tasks concurrently. It is constructed in four main ways. Given two tasks `left :: Task a` and `right :: Task b`:

1. `left -&&- right` executes two tasks at once. The type of the intermediate value of this parallel task is `(a, b)`, combining the values of both subtasks in a tuple.
2. `left -||- right` requires that `a` and `b` are the same type. This parallel task returns the first stable value.
3. `left -|| right` executes both tasks, but only keeps the value of the left. The parallel task has type `Task a`.
4. `left ||- right` is similar, but only keeps the right value instead, giving it type `Task b`.

There are other ways to construct a parallel task. It can contain more than two subtasks, and the intermediate value can be computed from the values of the subtasks using any procedure. However, these four are most common, and we can create more advanced parallel tasks by combining multiple of them. For instance, `left ||- middle -|| right` executes three tasks while only keeping the value of the middle task.

For sequential execution, iTasks features the `step` task. `step` keeps track a current task, along with any number of continuations. When the value of the current task changes, each continuation can use that value to decide whether the sequential task should transition to their followup task. Some continuations can be triggered by pressing a button, while others only use the current value to determine when to transition.

The `step` task is usually constructed with the `>>*` combinator. A common pattern for sequential tasks is to wait until a value becomes stable before transitioning:

```
enterName :: Task String
enterName = enterInformation []
```

```
>>* [OnAction ActionOk (hasValue return)]
>>* [OnValue (ifStable (viewInformation []))]
```

This first prompts the user to enter a string value. Then, when they press an ‘Ok’ button, the `return` task turns the unstable value from the interaction task into a stable value. After that, the second sequential task detects that the value is stable, and triggers the transition to the `viewInformation` task, which then shows the user the value they entered.

Finally, `iTasks` features multiple different builtin shared data sources. The simplest is the `SimpleSDSLens`, which we can retrieve using the `withShared` function:

```
enterName = withShared "" \sds.
  updateSharedInformation [] sds -&&- viewSharedInformation [] sds
```

This example defines a shared data source containing an empty string. The `update-` and `viewSharedInformation` functions behave just like their share-less counterparts, but their representation automatically updates when the share’s value changes. As a result, when the user changes the value of the input field created by the first task, the text printed by the second changes as well.

There are other ways to construct a `SimpleSDSLens`, such as the `sharedStore` function, which will store the share in a persistent map-based structure that can be accessed from anywhere. There is also the more general `SDSLens`, as well as several other types of builtin share types that we will not cover here, as they are outside the scope of this thesis.

Chapter 3

The Rust Programming Language

Rust is a multi-paradigm, general-purpose programming language. It is designed to have similar performance to low-level programming languages, such as C and C++, but without sacrificing memory safety.

Rust achieves this performance with its memory management strategy: it does not use an automatic garbage collector, but also does not require the programmer to manually allocate and free memory. Instead, rust tracks the lifetime of heap-allocated values. When a value is not used in the remainder of the program, it is deallocated automatically.

While usable for low-level systems programming, Rust also has features that are commonly found in higher-level programming languages. Most notably, Rust borrows some concepts from functional languages like Clean and Haskell, such as sum types, closures, and iterator combinators. We believe this makes Rust a promising candidate for a TOP library. On the other hand, unlike Clean, Rust does not have a higher-kinded type system, and therefore no monad abstraction either. Translating these aspects from iTasks will require some creativity.

This chapter provides a brief overview of the concepts of Rust that we use in this thesis. Please refer to the cited sources for more elaborate explanations on these features.

3.1 Variables, Ownership, and Mutability

One of the key takeaways from Rust is the concept of *ownership*. Values in a Rust program can have at most one owner. When this owner goes out of scope, the value is dropped (Gjengset, 2021). This simple program illustrates this idea:

```
fn main() {  
    // Creates a new string value and owner variable `x`  
    let x = String::new("Hello, world!");  
}
```

```
println!("{}", x);
} // `x` goes out of scope, value is dropped
```

We can move the ownership of a value to another variable, either by assigning it directly, or by passing it as parameter of another function. Note that the previous variable no longer owns the value in that case; attempting to use it afterwards will result in a compile-time error:

```
fn main() {
    let x = String::new("Hello, world!");
    let y = x;
    println!("{}", y);
    // Value of `x` was moved, using it is not allowed
    // println!("{}", x);
}
```

If we want to be able to use the value in multiple places in our program, we can do so using *references*:

```
fn main() {
    let x: String = String::new("Hello, world!");
    let y: &String = &x;
    println!("{}", y);
    println!("{}", x);
}
```

Note that Rust can infer the types of the variables automatically, we added the type annotation here to emphasize the different types of `x` and `y`. `x` is a `String`, while `y` is a *shared reference* to a `String`. Shared references have read-only access to the value they point to. Since the `println!` macro does not need to change the value, a shared reference suffices.

In cases where we want to allow the borrower to also mutate the data, we use a *mutable reference*. Note however, that variables in Rust are immutable by default. Before we can make a mutable reference to a variable, we have to mark the variable as mutable first using the `mut` keyword:

```
fn main() {
    let x = 5;
    // Would result in a compile-time error
    // x = 6;

    let mut y = 5;
    // Allowed
    y = 6;
}
```

We can then create a mutable reference as follows:

```
fn main() {
    let mut x: i32 = 5;
    let mut y: &mut i32 = &mut x;
    increment(y);
}

fn increment(number: &mut i32) {
    *number += 1;
}
```

The unary `*` operator dereferences the reference, allowing us to mutate the value 5 directly.

3.2 Borrow Checker

As mentioned earlier, the borrow checker tracks the lifetimes of references at compile-time to make sure they always point to valid data. This can eliminate entire groups of errors, such as the dangling pointer (Klabnik & Nichols, 2018):

```
// Does not compile
fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

`s` owns the string value, and it goes out of scope at the end of the function. Therefore the borrow checker will reject this program, because the reference `&s` would no longer point to valid data after returning.

Lifetimes are embedded into the type system of Rust. A value's lifetime starts as soon as it is created, and ends when it is moved or its owner goes out of scope. References also keep track of their lifetime. It starts when a value is borrowed, and ends the last time that borrow is used. For a program to compile, references may never *outlive* the value they borrow. The following code will not compile, because the lifetime of the value `'b` does not outlive the lifetime of the reference `'a`.

```
fn main() {
    let r;                                // -----+-- 'a
                                        //          |
    {                                    //          |
        let x = 5;                      // -+-- 'b  |
        r = &x;                          //  |    |
    }                                    // -+    |
                                        //          |
```

```
println!("r: {}", r); // |
}
```

Note that the lifetime of a reference is not always clear, such as in this contrived example:

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    if false {
        println!("r: {}", r);
    }
}
```

This program will never use a value after it is freed, and so it is theoretically safe to compile. However, the borrow checker is conservative, and will reject any programs that *may* contain memory safety issues, including this one. In such cases, it may help to restructure the program to convince the compiler it is safe:

```
fn main() {
    let x = 5;
    let r = &x;
    println!("r: {}", r);
}
```

Rust also features the `'static` lifetime, which indicates that the data pointed to by the reference stays valid for the remainder of the running program. String literals for instance, such as in `println!("Hello, world")`, are stored as constants in the program binary. We cannot take ownership over them, we can only use them by reference. Therefore their type is `&'static str`, because the value is never dropped while the program is running.

Other than verifying references are always valid, the borrow checker also makes sure borrowed values are used in a thread-safe manner. At any given point in time, a value may either have exactly one mutable reference, or any number of shared references. This check prevents *data races*: concurrent reads and/or writes to the same memory location, causing non-deterministic results.

3.3 Data Types

To create a new type, we usually use Rust's `struct` keyword. We use two different types of structs: classic C-like structs and tuple structs. Classic structs are defined as follows:

```
struct Foo {  
    bar: i32,  
    baz: bool,  
}
```

To create an instance of this struct, and access its fields, we use the following syntax:

```
fn main() {  
    let bar = 7;  
    let foo = Foo {  
        bar,  
        baz: true,  
    };  
    println!("{}", foo.baz);  
}
```

Notice how we specify the name of the field `baz`, followed by the value we want it to have. This is not necessary for the `bar` field, because we want to assign the local variable with the same name to it.

Tuple structs are similar to classic structs, but with anonymous field names:

```
struct Foo(i32, bool);
```

These are often used for simple wrapper types with exactly one field. Constructing a tuple struct is straightforward. We can access the fields of a tuple struct using their index:

```
fn main() {  
    let foo = Foo(7, true);  
    // Prints `true`  
    println!("{}", foo.1);  
}
```

Note that Rust also supports regular (anonymous) tuples:

```
fn main() {  
    let foo: (i32, bool) = (7, true);  
    println!("{}", foo.1);  
}
```

Note that structs, as well as their fields, can be marked as public using the `pub` keyword. This allows them to be used in other modules.

```
pub struct Foo {
```

```
pub bar: i32,  
pub baz: bool,  
}
```

Structs are product types – their domain consists of the product of the domains of their fields. Rust also has **enums**, which are sum types or ‘tagged unions’. They can be created by specifying several variants in struct-like notation:

```
enum Foo {  
    Bar { quux: i32, quuz: bool },  
    Baz(i32, bool),  
    Qux,  
}
```

The syntax for constructing an enum variant is similar to the syntax of structs, but with the name of the enum as a prefix. Accessing the fields of an enum is less straightforward, because the compiler insists we handle every possible variant, for instance with a **match**:

```
fn main() {  
    let foo = Foo::Baz;  
    match foo {  
        // Rename `quux` field to `value`  
        // Use .. for all other values, as they will not be used  
        Bar { quux: value, .. } => println!("{}", value),  
        Baz(n) => println!("{}", n),  
        Qux => println!("qux"),  
        // _ is a wildcard to match all other cases, unreachable in this case  
        // _ => println!("something else")  
    }  
}
```

3.4 Functions and Methods

While we have seen functions in previous sections, we will elaborate on them for completeness. Functions can be created with the **fn** keyword. We can call functions from other Rust files if they have the **pub** keyword as well:

```
pub fn foo(bar: i32) -> Baz {  
    // return a `Baz`  
    // ...  
}
```

The type after the arrow in the signature is the function’s return type. While Rust can often infer the return type of a function automatically, the programmer must specify what this type is, for readability.

We can also associate functions with structs and enums using `impl` blocks. For example, it is recommended to give Rust types a constructor where applicable:

```
impl Foo {
    pub fn new(bar: i32) -> Self {
        let baz = true;
        Foo { bar, baz }
    }
}
```

`Self` refers to the type the function is implemented on. In this case, it is an alias for `Foo`. We can call this constructor, and any other static functions, using the double-colon syntax: `Foo::new(7)`.

The first parameter of a function can also be `self`, `&self`, or `&mut self`. In that case, it is an associated *method*:

```
impl Foo {
    pub fn borrow(&self) {
        println!("{}", self.bar)
    }

    pub fn borrow_mut(&mut self, value: i32) {
        self.bar = value;
    }

    pub fn consume(self) -> Bar {
        Bar { baz: self }
    }
}
```

To call a method, we use dot notation on an object of the type that has the method:

```
fn main() {
    let mut foo = Foo::new(7);
    foo.borrow(); // Prints 7
    foo.borrow_mut(8); // Sets `bar` field to 8
    let bar = foo.consume(); // Turns `Foo` type into `Bar`
}
```

Which kind of `self` parameter to use depends on how the object is used. `&self` indicates that a method needs to borrow the value. When we call that method, Rust automatically creates a shared reference to the object, and passes it as a parameter. Similarly, `&mut self` indicates that a method requires mutable access to the value. The rules of the borrow checker also apply here, hence we cannot call an `&mut self` method on a value that is already being borrowed. Methods that have a `self` or `mut self` parameter take ownership of the value when they are called. Since the value is moved, it can no longer

be used after the call.

As a general rule of thumb, all functions should request the least amount of ownership over their parameters that they need to work. If they only read a value, a shared reference suffices; requesting a mutable reference or ownership would make it harder to write programs that pass the borrow checker.

3.5 Generics

To create functions or data types that work with multiple different types, Rust has generic type parameters. One example that makes use of this feature is Rust's `Option` type:

```
enum Option<T> {
    None,
    Some(T),
}

impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

We can apply the same principle to structs, and loose functions:

```
pub fn unwrap<T>(option: Option<T>) -> T {
    match option {
        Some(val) => val,
        None => panic!("called `Option::unwrap()` on a `None` value"),
    }
}
```

Rust *monomorphizes* all instances of generics in a Rust program at compile time. This creates multiple specialized versions of the same type or function, that work for every concrete type that is used as a parameter. As such, it is important that the types used as generic type parameters are known at compile time.

3.6 Traits

To define shared behavior across types, we can use `traits`:

```
pub trait Foo {  
    fn bar(&self);  
}
```

This creates an interface that we can then implement for other types, using `impl _ for`:

```
impl Foo for Baz {  
    fn bar(&self) {  
        // ...  
    }  
}  
  
fn main() {  
    let baz = Baz::new();  
    baz.bar();  
}
```

We can specify supertraits for a trait, which means any type that implements it must also implement all supertraits:

```
pub trait Foo: Clone + Display {  
    // ...  
}
```

We can also use traits to put *bounds* on generic type parameters, allowing us to call the methods of that trait:

```
fn call_bar<T: Foo>(foo: &T) {  
    foo.bar();  
}
```

To keep code readable when we use multiple type parameters and many trait bounds, we can use the `where` keyword to specify trait bounds:

```
fn call_bar<T>(foo: &T)  
where  
    T: Clone + Foo,  
{  
    // ...  
}
```

It is best practice to only impose trait bounds that are necessary for the type or function itself to work, even if we know we will need the trait bound for other functionality. This is why most structs with generic type parameters do not have trait bounds, but certain methods on that struct do.

Rust also provides the `impl` keyword to simplify arguments with generic types:

```
fn call_bar(foo: &impl Foo) {  
    // ...  
}
```

It can also be used as a return type. This instructs the compiler to infer the return type from the function itself. It is useful to simplify function signatures with large, nested return types.

3.6.1 Associated Types

Traits can have both generic type parameters and *associated types*:

```
trait Foo<T> {  
    fn baz(&self) -> T;  
}  
  
trait Foo {  
    type Bar;  
  
    fn baz(&self) -> Self::Bar;  
}
```

Both of these can be used to make the trait generic over types, the difference between them is subtle. The version with type parameter is monomorphized at compile time, essentially creating multiple different traits. This allows a single type to implement the trait multiple times, as long as it uses a different type parameter each time. In contrast, the version with associated type does not create multiple instances of the trait, and therefore types can only implement it once.

Note that neither of these examples is better than the other, they just have different uses. In some cases, it is desirable to have multiple implementations of a trait, such as Rust's builtin `From<T>` trait:

```
pub trait From<T> {  
    fn from(_: T) -> Self;  
}
```

This trait, combined with its derived dual `Into`, allows us to easily convert types to other types. `From` can be seen as a constructor: it returns `Self`, and uses some other type `T` as a parameter. It is useful that this trait uses a type parameter, because it enables us to, for instance, implement both `From<i32>` and `From<i64>` for `i128`.

We can restrict the type an associated type must be in trait bounds as follows:

```
fn call_bar<T: Foo<Bar = i32>>() {  
    // ...  
}
```


However, it may also be left out to indicate that any associated type will do. This is allowed because, after `T` is monomorphized during compilation, there will only be one `Foo` implementation for it, and therefore the compiler can infer the associated type. This is not possible for generic type parameters, as there might be multiple trait implementations.

3.6.2 Common Traits

Rust's standard library contains quite a few traits. This section describes the ones we use in this thesis briefly.

- `Clone` indicates a type can be cloned. We can call the `.clone()` method to create a deep copy of the object. This is sometimes useful to turn a reference into an owned type, although doing so is considered unidiomatic if cloning can be prevented.
- `Copy` is similar to `Clone`, but it copies objects implicitly. When we *move* a `Copy` value, it is copied rather than moved, and so the old owner will still be usable. `Copy` can only be implemented on types of which all contents also implement `Copy`, which eliminates everything not stored on the stack. Moreover, `Copy` should only be implemented on types that are relatively small.
- `Debug` allows types to print their contents using `println!("{}", t)`. It is recommended to implement this trait as liberally as possible, as it makes debugging easier.
- `Default` provides a parameter-less constructor for a type. It indicates that there is a value that makes sense to use as a default, such as 0 for numbers, `false` for booleans, or `None` for `Option<T>`.
- `Eq` and `PartialEq` indicate that values of that type can be checked for equality. `PartialEq` overloads the `==` operator, while `Eq` indicates that the relation is reflexive, symmetric and transitive.
- `Ord` and `PartialOrd` are similar, but used for comparison. `PartialOrd` overloads the `<`, `>`, `<=`, and `>=` operators, while `Ord` indicates values of the type form a total order. `Eq` and `Ord` together allow a type to be used safely in binary search trees, for instance.
- `From<T>`, as described above, indicates that a type can be created from another type.
- `Into<T>` indicates that a type can be converted to another type. This trait should never be implemented manually, because implementing `From<T>` for `U` automati-

cally provides an implementation for `Into<U>` for `T`.

- `Display` allows types to print themselves in a custom way using `println!("{}", t)`. This trait is implemented by default for types that have one obvious way to represent them as text, such as numbers and strings. This is not the case for many types. `Vec<T>` for example, could be represented in many ways: comma-separated, newline-separated, with or without square brackets around the elements, etcetera. As such, it does not implement `Display`.
- `FromStr` specifies how this type can be parsed from a string. The `.parse()` method present on strings can return any type that implements the `FromStr` trait.
- `Serialize` and `Deserialize` are traits from the popular  `serde` crate. They provide a generic abstraction for types that allows specialized crates to serialize and deserialize them in a certain format, such as JSON.

3.6.3 Trait Objects

We can turn traits into objects using the `dyn` keyword:

```
struct Qux {  
    foo: Box<dyn Foo>,  
}
```

This indicates that Rust should use *dynamic dispatch*: it will keep track of the relevant function pointers of the trait at runtime. Compared to generic type parameters, this should shorten compile time, but imposes some runtime overhead.

Note that we wrapped `dyn Foo` in a `Box<T>`. The reason this is necessary is because Rust needs to know the size of the `Qux` type at compile time. `dyn Foo` has an unknown size, so we wrap it in a `Box`, which has a known size regardless of what is stored inside. The `Box` type indicates that we store the data inside on the heap. Its size is known because it is basically a fat pointer that keeps track of the location on the heap and the size of the content.

Trait objects can only be created for traits that are *object-safe*. To be object-safe, a trait must satisfy all of these requirements:

- The trait must not require that `Self: Sized`
- None of the methods have type parameters (the trait itself may have type parameters)
- `Self` is never used within the trait definition

This implies that, among others, the `Clone` trait may not be turned into a trait object, as its `clone` method returns `Self`.

3.7 Macros

Rust allows users to create macros to reduce boilerplate. Macros can be seen as functions that take a piece of Rust code – a *tokenstream* – as input, and produce more Rust code. All macros are expanded at compile time. There are two different types of macros: *declarative* macros and *procedural* macros.

3.7.1 Declarative Macros

Declarative macros are the simplest and most common variant. They are defined using a domain-specific language similar to a `match` expression. When called, declarative macros look similar to normal functions, but have an extra exclamation mark at the end. We have already seen the `println!` macro. This is a macro because it has to support a variable number of parameters, which is not possible to express in normal Rust code. More accurately, the `println!` macro actually has exactly one argument, which is a tokenstream consisting of a comma-separated list. The macro expands into Rust code that prints each of the parts of the format string separately. Similarly, the `vec!` macro creates a `Vec<T>` and pushes all of the supplied elements into it.

We can use macros to reduce boilerplate code in various ways, such as trivially implementing a trait for many different types:

```
trait Foo {}

macro_rules! impl_foo {
    ($($t:ty),*) => {
        $(impl Foo for $t {})*
    }
}

impl_foo!(u8, u16, u32, u64, u128, usize, i8, i16, i32, i64, i128, isize, f32, f64);
```

`$($t:ty),*` indicates that we want a comma-separated list of types. We loop over all of those types, generating the code `impl Foo for $t {}` for each type, where `$t` is replaced with the current type.

3.7.2 Procedural Macros

The example above uses `macro_rules!` to specify a macro in a declarative manner. It is also possible to write macros as a normal Rust function that takes a `TokenStream` as input and produces another `TokenStream`. This gives more flexibility than `macro_rules!` can provide, at the cost of added complexity. We call these *function-like procedural macros*.

Another flavor of procedural macros is the *custom derive*. By default, Rust allows us to derive certain traits for types:

```
#[derive(Copy, Clone, Debug, Default)]
struct Foo(i32);
```

We can create custom derive macros for our own traits. How this is done exactly is explained in chapter 5.

3.8 Closures

A closure is a function that can capture variables in its environment. We can create a closure in Rust using double pipes, with any number of parameters between them. We can then call the closure like any other function:

```
fn main() {
    let n = 5;
    let closure = |x: i32| x + n;
    let output = closure(6);
    // Prints `11`
    println!("{}", output);
}
```

Of course, this has implications for the borrow checker. A closure can either (mutably) borrow the variables it captures (`n` in the example above), or take ownership over them. To differentiate between these, closures automatically implement the traits `FnOnce`, `FnMut`, and `Fn` where applicable.

- All closures implement `FnOnce`, which describes what the closure would do if it gets ownership over the variables it captures. If neither other trait is implemented for the closure, it can only be called once, hence the name.
- If a closure would still work with mutable references to the variables it captures, rather than ownership, it will implement `FnMut`. These closures can be called multiple times, as long as they can borrow the environment mutably.
- Finally, if a closure can also do with shared references, it will implement `Fn`. These, too, can be called as often as needed, as long the closure does not outlive the captured variables.

It is best practice to use the least restrictive trait necessary as a bound on type parameters, such as `Option<T>`'s `.map()` method:

```
pub fn map<U, F>(self, f: F) -> Option<U>
where
    F: FnOnce(T) -> U,
{
    match self {
        Some(x) => Some(f(x)),
    }
}
```

```
        None => None,
    }
}
```

3.9 Asynchronous Programming

Many code fragments throughout this thesis contain the keywords `async` and `await`, as well as the trait bounds `Send` and `Sync`. `async` indicates that a function does not return the indicated return type `T`, but rather an `impl Future<Output = T>`, a promise that a value of type `T` will be returned eventually. Callers of asynchronous functions can choose to `await` the future, which will yield the value of type `T`. Alternatively, they can use an asynchronous runtime to, for instance, await multiple futures in parallel across different threads.

`async` functions might need to move intermediate values to other threads if those values were created before an `await`. This is where Rust's thread safety comes into play. The `Send` trait is automatically implemented for any type that is safe to send to another thread. Similarly, `Sync` is implemented for any type `T` that is safe to *share* with other threads, i.e. `&T` implements `Send`. Generic type parameters in asynchronous functions therefore need `Send` and/or `Sync` trait bounds whenever a value of that type is held across an `await`.

Asynchronous programming can create fast, concurrent code that looks like synchronous code. Although the `Send` and `Sync` trait bounds can be quite verbose at times, a synchronous implementation of the TOP library described in this thesis would look very similar. We chose to use `async` functions in order to make it easier to communicate with asynchronous external libraries. Other than that, `async`, `await`, `Send`, and `Sync` can be safely ignored.

Chapter 4

Initial Implementation

In this chapter, we explain how we implemented a naive initial version of a task-oriented programming library in Rust. This includes the core of the library that allows it to function, as well as any parts that are reasonably language-agnostic.

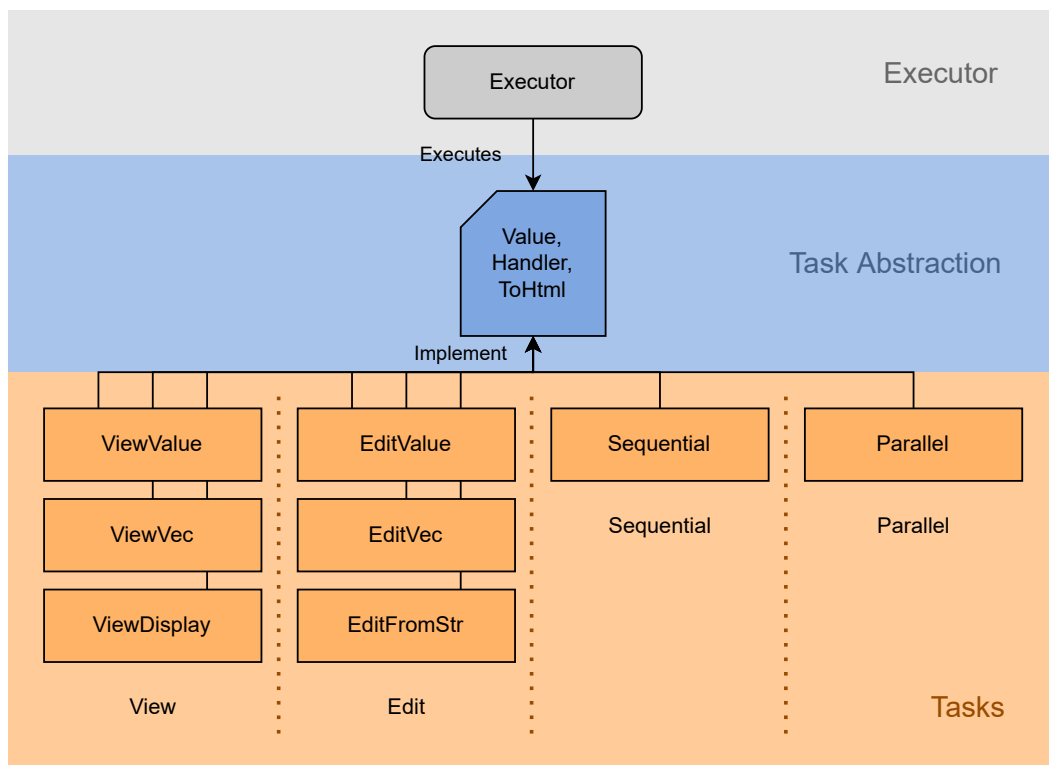


Figure 4.1: Overview of the structure of our initial TOP implementation.

Figure 4.1 gives an overview of the essential components of this library. The implementation can be split into three parts.

The task abstraction in the middle of the figure is arguably the most important part of the library, as all other components depend on it one way or another. The abstraction consists of three traits that describe what to display to the user, how the task should respond to inputs, and how we can retrieve its intermediate value.


At the bottom are the concrete tasks. In contrast with tasks in `iTasks`, tasks in this library are objects rather than functions. However, the library does provide functions to construct these tasks in a way that looks similar to `iTasks`. Tasks are subdivided into four main categories: view, edit, sequential, and parallel tasks. Unlike `iTasks`, which has `interactR`, `interactRW`, `parallel` and `step`, our implementation does not have one generic task for each of these categories. Instead, some categories contain multiple tasks, such as `EditValue` to edit simple types, and `EditVec` to edit vectors.

On the other side of the figure is the task executor. Its job is to take any structure that implements the task abstraction traits, and execute it. This includes providing users with a suitable interface, parsing their inputs, and reacting appropriately according to the task's specification.

We will explain each part of the implementation in the following sections. First, to give an idea what we are working towards, we will briefly cover the public API of the library when it is finished. After that, we will explain how the task abstraction works and what choices we made during its implementation. Then we explain in detail how we implemented tasks from each of the four categories in order. Finally, we will shortly cover how to go about executing tasks.

We will tackle most drawbacks of this implementation in the next chapter by making use of features that set Rust apart from other languages. This also includes the implementation of shared data sources.

4.1 Public API

We start with an overview of what users of this library would see, before going into the implementation in the following sections. As part of our executor implementation, we use the popular  `axum` web framework. A barebones `axum` program looks as follows:

```
async fn index() -> impl IntoResponse {  
    "Hello, world!"  
}  
  
#[tokio::main]  
async fn main() {  
    let router = Router::new()  
        .route("/", get(index));  
}
```

```

    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(router.into_make_service())
        .await
        .unwrap();
}

```

It uses a `Router` to specify which pages lead to which `tower` services. The `get` service forwards GET requests to its handler parameter. It is designed to work with any function that returns a type that implements `IntoResponse`, and for which all parameters implement `FromRequest`. This holds for the `index` function, as it does not have any parameters, and `IntoResponse` is implemented by default for strings. Running this code and visiting `http://0.0.0.0:3000` results in a webpage with content type `text/plain`, displaying “Hello, world!”.

When we finish implementing the library, a barebones TOP program will look similar, making it simple for `axum` users to adopt:

```

async fn index() -> impl Task {
    unimplemented!()
}

#[tokio::main]
async fn main() {
    let router = Router::new()
        .nest("/static", TopService::new())
        .route("/", task(index));

    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(router.into_make_service())
        .await
        .unwrap();
}

```

The `task` function works similar to `get`, but it requires any function that returns a type that implements the task abstraction to work instead. We attach it to our router in the same way. Tasks require some JavaScript code to function as well, so we serve those in a separate `TopService`. Running this code and visiting `http://0.0.0.0:3000` will yield an interface for interacting with the task created by `index`.

4.1.1 View Tasks

To construct a view task, we simply pick the type of task we need and call its constructor. The simplest view task is `ViewValue`, which displays any type that implements `Display`:

```

fn greet() -> impl Task {
    ViewValue::new("Hello, world!")
}

```

When the executor calls this function, it should display “Hello, world!” to the user in some way.

`ViewValue` works with many simple types, such as numbers, booleans, characters, and strings. Theoretically, we could also make it work for more complex types, such as `Vec<T>`, as long as we implement `ToHtml` for `ViewValue<Vec<T>>`. However, since complex types generally have more parameters to tweak, we will implement specialized tasks for them:

```
fn greet() -> impl Task {
    ViewVec::new(vec!["hello", "world"])
}
```

Later on, this will allow us to change the way it displays its elements, in a row or in a column for instance.

It can quickly become cumbersome to remember which types of tasks are meant for which types of values: `ViewValue::new(vec!["hello", "world"])` would be a reasonable mistake. Manually constructing each type of task is also quite verbose. To solve both of these problems, we will define a `view` function that automatically picks an appropriate task to display a certain type of value:

```
fn greet() -> impl Task {
    view("Hello, world!")
}
```

This call constructs a `ViewValue<&str>` task, whereas `view(vec!["hello", "world"])` constructs a `ViewVec<&str>` task. The `view` function is the recommended way to construct tasks where possible, but it only defines the default task for a certain type of value. If we want to use a different type of task than the default, such as a `ViewSmiley` task that displays booleans as a smiley face, we would have to call its constructor directly.

4.1.2 Edit Tasks

The API of edit tasks is similar to the API of view tasks. We can create edit tasks by using their constructors directly. Note that edit tasks do not necessarily start with a value. Therefore, the constructor usually requires an optional parameter:

```
fn greet() -> impl Task {
    EditValue::new(Some(false))
}
```

Additionally, there are `edit` and `enter` functions that behave similar to `view`. `edit` has one parameter, and passes it as `Some` parameter to the underlying task:

```
fn greet() -> impl Task {
    edit(false)
}
```

`enter`, on the other hand, has no parameter. It either passes `None` to the underlying task, or some default value:

```
fn greet() -> impl Task {
    enter::<bool>()
}
```

We used the *turbofish* (`::<>`) here to specify a concrete type parameter for the `enter` function. It can be elided if the type can be retrieved from the context.

4.1.3 Parallel Tasks

Similar to `iTasks`' `-&&-`, `-||-`, `-||`, and `||-`, we can construct parallel tasks using combinator methods: `and`, `or`, `left`, and `right`. These methods are present on all types that implement the task abstraction:

```
async fn greet() -> impl Task {
    view("Hello")
    .and(view("Please enter a name:"))
    .right(enter().or(return_value("world").to_owned()))
    .left(view("!"))
}
```

This is roughly equivalent to the following definition in `iTasks`:

```
greet :: Task String
greet = viewInformation [] "Hello"
    -&&- viewInformation [] "Please enter a name:"
    ||- (enterInformation [] -||- return "world")
    -|| viewInformation [] "!"
```

`return_value` is equivalent to `iTasks`' `return` method. We renamed it because `return` is a keyword; we cannot use it as an identifier. In this example, we use it to specify a default value for when the user does not enter a name. Note that `enter` does not need a *turbofish* here, because its output type is inferred to be the same as the output type of `return_value`. The intermediate value of the entire parallel task is a `String` representing a name:

1. The `and` combinator creates a task with output type `(&str, &str)`
2. `right` then ignores that output, and uses the intermediate value of its parameter instead

3. The `or` combinator in the parameter returns the first non-empty value of its tasks, which is a `String`
4. Finally, `left` keeps that string value, ignoring the output of its own parameter

4.1.4 Sequential Tasks

To construct a sequential task, we use the `step` method on an existing task. We can then add one or more continuations to it using the `on` method. `on` takes three parameters: a trigger, a condition, and a transformation function:

```
async fn greet() -> impl Task {
    view("Please enter your name:")
        .right(enter::<String>())
        .step()
        .on(
            Trigger::Button(Button::new("Ok")),
            |value| if_value(|name| !name.is_empty()),
            |value| view(format!("Hello, {}!", value.unwrap())),
        )
        .on(
            Trigger::Button(Button::new("Cancel")),
            always,
            |_| view("Goodbye!".to_owned()),
        )
}
```

The trigger determines when the task should attempt to transition to the next task. It can either be `Trigger::Update`, which triggers every time the value of the task updates, or `Trigger::Button`, which creates a button and triggers when that button is pressed. These are similar to `iTasks`' `OnValue` and `OnAction` continuations.

The condition dictates whether the task is allowed to transition. If the condition is false, any buttons that would trigger this continuation will be disabled. The transformation function actually constructs the followup task from the value of the current task. To make task value handling less verbose in these functions, the library provides helper methods such as `if_stable`, `has_value`, `if_value`, and `always`. In `iTasks`, the condition and transformation function are one and the same. Other than that, the approaches are quite similar.

The example above will first prompt the user to enter their name. Then, if they press the 'Ok' button, the task will be replaced with a new view task that greets the user. If they press 'Cancel', the system will say goodbye instead.

4.2 Task Abstraction

For an executor to be able to work with a task, the task abstraction must be able to determine three things for any task:

1. What to display to the user
2. How to react to user inputs
3. How to retrieve the intermediate value

There is no systematic way of finding the most intuitive API that satisfies these requirements, but we can try to translate them to code as directly as possible. We found that the best way to do this in Rust is using traits, one for each requirement. The first trait handles how a task should be displayed to the user. We focus on web-based interfaces in this thesis, so we define a trait to translate tasks to HTML:

```
pub trait ToHtml {  
    async fn to_html(&self) -> Html;  
}
```

The second trait describes how a task should respond to user input:

```
pub trait Handler {  
    async fn on_event(&mut self, event: Event) -> Feedback;  
}
```

It receives an `Event` parameter, which indicates that the user pressed a button or updated an HTML input with a certain identifier. The task is then expected to update its state (and optionally the state of its subtasks) and return appropriate `Feedback`. The `Feedback` then instructs the client to change their user interface. For instance, if the user updates the value of an input, the feedback could be that the new value is valid, instructing the front-end to display a green checkmark next to the input. Other examples of feedback include replacing an element with a certain identifier with arbitrary HTML, appending HTML to it, or removing it altogether.

Finally, the third trait allows retrieving intermediate values from a task:

```
pub trait Value {  
    type Output;  
  
    async fn value(&self) -> TaskValue<Self::Output>;  
}
```

The associated type `Output` keeps track of the type of the intermediate values this task produces. It is equivalent to the type parameter `t` in `Task t` from `iTasks`. The `value`

method yields the task's current value, which can be `Stable`, `Unstable`, or `Empty` due to the `TaskValue` enum.

To prevent users from having to write `impl Value + Handler + ToHtml` as return type for `axum` handlers, we will also define an auxiliary `Task` trait that combines these three:

```
pub trait Task: Value + Handler + ToHtml {}

impl<T> Task for T where T: Value + Handler + ToHtml {}
```

4.2.1 Design Choices and Alternatives

While we tried our best to write code that is as intuitive as possible, there are many other ways in which one might implement useful TOP library. In this recurring subsection, we elaborate on confusing details, other prototypes, and important decisions we faced during the implementation. Note that not all earlier prototypes are discussed here, only those that could still have potential benefits over the chosen implementation.

Associated type or type parameter

Instead of using the associated type `Output`, we could have given `Value` a generic type parameter:

```
pub trait Value<T> {
    async fn value(&self) -> TaskValue<T>;
}
```

This would make it more similar to the `Task t` type from `iTasks`. However, as explained in chapter 3, this definition can monomorphize to multiple different types of traits, one for each unique type parameter that occurs in the program. Multiple traits would allow us to have multiple implementations for the same type of task. This is unnecessary, because the four categories of tasks we cover in this chapter only ever produce one type of value; there is no need for more than one implementation.

We could still choose to use a generic type parameter, and simply never implement it more than once for any type. However, that could potentially confuse users by implying a task might be able to produce different types of values. Moreover, the compiler can use the knowledge of there being only one implementation to improve type inference. If a task could theoretically produce multiple types of values, users would often have to specify which value they want.

Split traits

We could have combined `ToHtml`, `Handler`, and `Value` into a single `Task` trait. That way, the task abstraction would be simpler. However, splitting the `ToHtml` trait from

the others makes sense, because not all tasks need a user interface. `return_value`, for instance, does not need a web representation. The same goes for `Handler`: not all tasks need to be able to respond to user input. Tasks that do not appear in the user interface cannot be interacted with, and view tasks do not respond to user input either.

Another reason to split the traits is because it makes it easier to generalize the `ToHtml` and `Handler` traits for other types of front-end in the future. We could turn `ToHtml` into something similar to the following:

```
pub trait ToRepr<T> {  
    async fn to_repr(&self) -> T;  
}
```

Current `ToHtml` representations would be equivalent to `ToRepr<Html>` in this case. However, the generic type parameter allows us to implement other types of representations for the same task as well. For instance, a `ToRepr<Cli>` implementation for command line interfaces, and `ToRepr<Android>` for a mobile application. This would allow TOP to derive suitable interfaces for multiple different platforms from a single task definition, extending the domain in which the library can be used.

4.3 View Tasks

This section elaborates on our implementation of view tasks to display read-only information to the user. It includes a simple task to view primitive types. While the complete library also contains tasks for more complicated types, such as optionals and vectors, those will be explained in the next section on edit tasks.

4.3.1 Implementation

To implement a view task for basic values, we need to create a new type and implement the task abstraction traits for it. The `ViewValue<T>` task is quite straightforward, as it only holds the value it should display to the user:

```
struct ViewValue<T> {  
    value: T,  
}  
  
impl<T> ViewValue<T> {  
    pub fn new(value: T) -> Self {  
        ViewValue { value }  
    }  
}
```

To translate this task to an HTML representation, we simply convert the value to text using the builtin `Display` trait, and put that in an HTML `<div>` element. This will

allow `ViewValue` to work with any type, as long as that type implements `Display`:

```
impl<T> ToHtml for ViewValue<T>
where
    T: Display
{
    async fn to_html(&self) -> Html {
        Html(format!("<div>{}</div>", self.value))
    }
}
```

The `Handler` implementation is trivial: view tasks merely display information, users cannot interact with them. Any events that the user sends cannot be targeted at a view task, therefore we return empty feedback regardless of the event:

```
impl<T> Handler for ViewValue<T>
where
    T: Send,
{
    async fn on_event(&mut self, _event: Event) -> Feedback {
        Feedback::new()
    }
}
```

Finally, we can retrieve the intermediate value of this task by cloning the contained value:

```
impl<T> Value for ViewValue<T>
where
    T: Clone + Send + Sync,
{
    type Output = T;

    async fn value(&self) -> TaskValue<Self::Output> {
        TaskValue::Stable(self.value.clone())
    }
}
```

Note that this might be inefficient. In some cases, a reference to the value of the task might be enough, and cloning would be a waste of time. Why we chose to do this regardless, and how to tackle this problem efficiently, will be covered in chapter 5. Also note that the value returned from this view task is always stable. This is because the user cannot change it, and without shared data sources, the value will remain static.

The `ViewValue<T>` task is now complete. The implementation of `ViewVec<T>` and `ViewOption<T>` are quite similar, but because `Vec<T>` and `Option<T>` do not implement `Display`, they need some adjustments to how are displayed. We will not cover

them here, because the next section will explain the implementation of `EditVec<T>`, which is more interesting.

To make the `view` function which automatically determines which task to use for a type of value, we first define a new trait:

```
pub trait View: Sized {
    type Task: Value<Output = Self>;

    fn view(self) -> Self::Task;
}
```

This trait uses an associated type `Task` to specify the default task of the implementing type. We can now implement this trait for every type that implements `Display`, to indicate that those types should use the `ViewValue<T>` task:

```
impl<T> View for T
where
    T: Display + Sized
{
    type Task = ViewValue<T>;

    fn view(self) -> Self::Task {
        ViewValue::new(self)
    }
}
```

Now we can use this trait to define the `view` function, which takes any type that implements the `View` trait, and constructs the the associated task for it:

```
pub fn view<T: View>(value: T) -> T::Task {
    value.view()
}
```

4.3.2 Design Choices and Alternatives

Handler implementation

Note that we implemented the `Handler` trait for `ViewValue<T>`, even though it does not actually handle user input. Ideally, we would elide this implementation. The same goes for `ToHtml` implementations for tasks that have nothing to display. However, our current implementation requires that all tasks implement all three task abstraction traits, because it saves time and makes the implementation much shorter.

To illustrate the problem with an example: a parallel task should only implement `Handler` if any of its subtasks implements `Handler`. We can achieve this with multiple different `Handler` implementations, one for each combination of subtasks that do

or do not implement `Handler`. However, this is extremely verbose, and the compiler would need to use *specialization* to pick the most suitable implementation of `Handler` for a specific type. Specialization is a difficult problem, and the Rust compiler does not currently support it¹. There are workarounds², but those are far from idiomatic and would clutter the code to the point where it becomes unmaintainable.

We would need implementations for every possible combination of `ToHtml` and `Handler`, for both parallel and sequential tasks, and the executor. Therefore, it is less work to require that all tasks implement all task abstraction traits. They do not indicate that the task *should* handle events or *should* be displayed, but rather that they *can*, even if their implementation trivially responds without content.

4.4 Edit Tasks

In this section, we will explain how we implemented two specific edit tasks that allow the user to interactively change values: `EditValue` for primitive types, and `EditVec` for vectors. These tasks are slightly more complicated than view tasks, because they involve handling user input. Moreover, the HTML-representation requires more work, as types are converted to inputs, rather than simple text.

4.4.1 Implementation

The definition of the `EditValue<T>` task is similar to that of `ViewValue<T>`. The main difference is that it might not always have a value, and therefore stores it as an optional. It also needs an identifier for its `<input>` field, so that it can identify which incoming Events apply to this task:

```
pub struct EditValue<T> {
    id: Uuid,
    value: Option<T>,
}

impl<T> EditValue<T> {
    pub fn new(value: Option<T>) -> Self {
        EditValue {
            id: Uuid::new_v4(),
            value,
        }
    }
}
```

Remember that we used the `Display` trait to convert values to HTML `<div>` for view tasks. Because there is no builtin trait that describes how those same types can be

¹<https://rust-lang.github.io/rfcs/1210-impl-specialization.html>

²<https://lukaskalbertodt.github.io/2019/12/05/generalized-autoref-based-specialization.html>

displayed as HTML `<input>` elements, we will create a custom `Input` trait to do just that. We implement this trait for any type that has an input-type equivalent:

```
pub trait Input {
    fn input(value: &Option<Self>, id: &Uuid) -> Html;
}

impl Input for String {
    fn input(value: &Option<Self>, id: &Uuid) -> Html {
        Html(format!(
            r#<input id="{id}" value="{}" oninput="update(this)"/>"#,
            value.as_ref().unwrap_or_default(),
        ))
    }
}

impl Input for i32 {
    fn input(value: &Option<Self>, id: &Uuid) -> Html {
        Html(format!(
            r#<input id="{id}" type="number" value="{}" oninput="update(this)"/>"#,
            value.as_ref().unwrap_or_default(),
        ))
    }
}

impl Input for char {
    fn input(value: &Option<Self>, id: &Uuid) -> Html {
        Html(format!(
            r#<input id="{id}" maxlength="1" value="{}" oninput="update(this)"/>"#,
            value.as_ref().unwrap_or_default(),
        ))
    }
}

impl Input for bool {
    fn input(value: &Option<Self>, id: &Uuid) -> Html {
        Html(format!(
            r#<input id="{id}" type="checkbox" value="{}"\
            onclick="update(this, this.checked.toString())"/>"#,
            value.as_ref().unwrap_or_default(),
        ))
    }
}

// ...
```

Now we can implement `ToHtml` for any `EditValue<T>`, as long as `T` can be turned into an `<input>`:

```
impl<T> ToHtml for EditValue<T>
where
```

```

T: Input + Send + Sync
{
  async fn to_html(&self) -> Html {
    Html(format!("{}", T::input(&self.value, &self.id)))
  }
}

```

Notice how each of the `Input` implementations calls a JavaScript `update` function when the input's value changes. `update` creates an `Event` containing the input's identifier and new value. It then transmits it to the server, which will pass it to the executed tasks' `Handler` implementation, where we can handle it:

```

impl<T> Handler for EditValue<T>
where
  T: FromStr + Send + Sync,
  T::Err: Send,
{
  async fn on_event(&mut self, event: Event) -> Feedback {
    match event {
      Event::Update { id, value } if id == self.id => {
        match value.parse::<S::Value>() {
          Ok(value) => {
            self.value = Some(value);
            Feedback::from(Change::Valid { id })
          }
          Err(_) => {
            self.value = None;
            Feedback::from(Change::Invalid { id })
          }
        }
      }
      _ => Feedback::new(),
    }
  }
}

```

We first check if the identifier of the event matches the identifier of this task, because it is possible that the user has multiple different edit tasks on-screen. If it matches, we try to parse the new value of the input using the builtin `FromStr` trait. If that succeeds, we update the value of this task and send feedback that the value was synchronized successfully. Otherwise, we remove the value and indicate that the supplied value is invalid. The executor will send this feedback back to the front-end, which gives visual feedback accordingly.

Finally, the `Value` implementation of edit tasks is simple. In contrast to view tasks, edit tasks can only produce unstable values. This is because the user can still change the value through the user interface as long as the task exists:

```

impl<T> Value for EditValue<T>
where
    T: Clone + Send + Sync,
{
    type Output = T;

    async fn value(&self) -> TaskValue<Self::Output> {
        match self.value.clone() {
            None => TaskValue::Empty,
            Some(value) => TaskValue::Unstable(value),
        }
    }
}

```

The special `edit` and `enter` functions are implemented similarly to the `view` function, by using an `Edit` trait that defines a default edit task for a type:

```

pub trait Edit: Sized {
    type Task: Value<Output = Self>;

    fn edit(value: Option<Self>) -> Self::Task;
}

pub fn enter<T: Edit>() -> T::Task {
    T::edit(None)
}

pub fn edit<T: Edit>(value: T) -> T::Task {
    T::edit(Some(value))
}

```

The `edit` function initializes the task with the supplied value, while `enter` uses either `None` or the default value of that type as initial value. The latter depends on whether the default input of a type represents a valid value. For example, `enter::<i32>()` creates an `EditValue<i32>` with `Some(0)` as initial value, because a number `<input>` starts with 0 as value, which is a valid `i32`. In contrast, `enter::<char>()` is initialized with `None`, because its input starts out empty, which is not a valid character:

```

impl<T> Edit for i32 {
    type Task = EditValue<T>;

    fn edit(value: Option<Self>) -> Self::Task {
        EditValue::new(Some(self.unwrap_or_default()))
    }
}

impl<T> Edit for char {
    type Task = EditValue<T>;
}

```

```

    fn edit(value: Option<Self>) -> Self::Task {
        EditValue::new(value)
    }
}

// ...

```

Now on to a slightly more complicated task: `EditVec<T>`. Unlike in `EditValue<T>`, the generic type parameter `T` here indicates the type of task used for single elements within the vector, rather than the type of intermediate value. In its most basic form, `EditVec<T>` consists of a vector of subtasks, and allows the user to edit those subtasks, as well as adding new tasks or removing existing ones. Other than that, its definition mainly keeps track of identifiers for containers and buttons, which allow it to manipulate its front-end representation in response to button presses:

```

pub struct EditVec<T> {
    container_id: Uuid,
    elements_id: Uuid,
    add_id: Uuid,
    rows: Vec<Row>,
    tasks: Vec<T>,
}

impl<T> EditVec<T> {
    pub fn new(tasks: Vec<T>) -> Self {
        EditVec {
            container_id: Uuid::new_v4(),
            elements_id: Uuid::new_v4(),
            add_id: Uuid::new_v4(),
            rows: tasks.iter().map(|_| Row::new()).collect(),
            tasks,
        }
    }
}

struct Row {
    container_id: Uuid,
    remove_id: Uuid,
}

impl Row {
    fn new() -> Self {
        Row {
            container_id: Uuid::new_v4(),
            remove_id: Uuid::new_v4(),
        }
    }
}

```

The `Row` struct keeps two identifiers for each subtask: one for a container `<div>` element,

and one for the remove button that allows the user to delete an element. The `ToHtml` implementation of `EditVec<T>` also provides the user with a plus button to insert a new task of type `Ts`:

```
impl<T> ToHtml for EditVec<T>
where
  T: ToHtml + Send + Sync,
{
  async fn to_html(&self) -> Html {
    let mut tasks = String::new();
    for (task, row) in self.tasks.iter().zip(&self.rows) {
      tasks += &format(
        r#"<div id="{}">
          {}
          <button id="{}" type="button" onclick="press(this)">-</button>
        </div>"#,
        row.container_id,
        task.to_html().await.to_string(),
        row.remove_id,
      );
    }
    Html(format!(
      r#"<div id="{}">
        <div id="{}">{}</div>
        <button id="{}" type="button" onclick="press(this)">+</button>
      </div>"#,
      self.container_id,
      self.elements_id,
      tasks,
      self.add_id,
    ))
  }
}
```

Similar to the JavaScript `update` function, `press` creates a button press `Event` and sends it to the server. We can react to this event in the `Handler` implementation:

```
impl<T> Handler for EditVec<T>
where
  T: ToHtml + Handler + Default + Send,
{
  async fn on_event(&mut self, event: Event) -> Feedback {
    match event {
      Event::Press { id } if id == self.add_id => {
        // Plus button pressed, add a new row.
        let row = Row::new();
        let task = T::default();
        let html = Html(format!(r#"
          <div id="{}">
            {}
            <button id="{}" type="button" onclick="press(this)">-</button>
          </div>"#
```

```

        </div>
        "#, row.container_id, task.to_html().await, row.remove_id));

        self.rows.push(row);
        self.tasks.push();

        Feedback::from(Change::AppendContent {
            id: self.elements_id,
            html,
        })
    }
    Event::Press { id } if self.rows.iter().any(|row| row.remove_id == id) => {
        // Minus button pressed, remove an existing row.
        let index = self.rows.iter().position(|row| row.sub_id == id).unwrap();
        let removed = self.rows.remove(index);
        self.tasks.remove(index);

        Feedback::from(Change::Remove { id: removed.container_id })
    }
    // Different event, forward to all subtasks
    _ => future::join_all(
        self
            .tasks
            .iter_mut()
            .map(|task| task.on_event(event.clone()))
    ).await,
    }
}
}

```

So far, the `Feedback` we have seen only indicated whether or not values were valid. With `EditVec<T>`, however, the feedback is used to change the structure of the layout entirely. When the user presses the plus button, we create an empty task, put it inside a `<div>` element along with a new minus button, and append it to the end of the list. When that minus button is pressed, we remove it from the front-end again.

The `Value` implementation of this task simply takes the values of each of the inserted tasks, and joins them in a vector:

```

impl<T> Value for EditVec<T>
where
    T: Value + Send + Sync,
{
    type Output = Vec<T::Output>;

    async fn value(&self) -> TaskValue<Self::Output> {
        let values = future::join_all(
            self
                .tasks
                .iter()
                .map(|task| task.value())
        )
    }
}

```

```

        ).await;
    values.into_iter().collect()
}
}

```

The call to `into_iter` immediately followed by `collect` might seem strange. It implicitly converts a vector of task values to a task value of a vector using a custom `FromIterator` implementation for `TaskValue`. It will yield `TaskValue::Empty` if any of the task values is empty. Otherwise, it will return an unstable value if any of the values is unstable, as the resulting vector could still change in that case. If all values in the vector are stable, the result is stable as well.

We have shown how to define edit tasks for types that only require one input field to create using `EditValue<T>`, as well as types composed of multiple fields with `EditVec<T>`. Using the same techniques, we can define custom edit tasks for most types imaginable. This also includes more creative tasks, such as a *choose* task that displays several options in a dropdown menu or checklist.

4.4.2 Design Choices and Alternatives

Feedback

The flexibility of tasks is limited by the **Feedback** primitives we choose for the front-end. **AppendContent**, **Remove**, **Valid**, and **Invalid** are sufficient for many simple tasks, but there are useful examples of tasks where these fall short. For example, updating a list in `iTasks` also allows users to reorder elements of the list, using buttons that swap elements with the element above or below them. This requires better primitives, because we only allow inserting new HTML at the end of an element through **AppendContent**.

Similarly, more **Event** variants would also allow for more useful (and more complex) tasks. An event currently only indicates a new value for an input field, or a button press. A more modern and user-friendly approach to reorder items in a list would be to drag elements to their desired location. This would require a more elaborate **Event** variant to keep track of the starting- and destination index, for instance. For now, we chose simple primitives that are powerful enough to show TOP is viable in Rust; user experience design is outside the scope of this thesis.

Input trait

The **Input** trait forms a minor disparity between view- and edit tasks. **Display** turns types into text, which we then turn into HTML by surrounding it with `<div>` tags, whereas the **Input** trait turns types into HTML directly. This is necessary because some types require different attributes other than just the `type` attribute. `bools`, for instance, require an adjusted call to `update`, because the `value` field of a checkbox does not indicate whether it is checked or not; the `checked` field does.

To fix this disparity, we would replace `Display` with a new trait that describes how types should be turned into HTML-representations for view tasks directly. This also allows more fine-grained control over the front-end: `bools` are currently displayed as ‘true’ or ‘false’, but a new trait would allow a check- or crossmark representation for example.

4.5 Parallel Tasks

Parallel tasks execute multiple subtasks at the same time, and compute their own value by combining the values of those tasks in some way. Conceptually, this combining procedure can be anything, so there are many different variations of parallel tasks. In this section, we will implement the `and`, `or`, `left`, and `right` combinators that are also included in `iTasks`. We will also show how to support other combining procedures.

4.5.1 Implementation

Our implementation of the parallel task consists of exactly two subtasks, along with a function that combines them:

```
pub struct Parallel<L, R, F> {
    left: L,
    right: R,
    combine: F,
}
```

The `ToHtml` and `Handler` implementations of this task are quite straightforward. `ToHtml` simply concatenates the HTML representations of both subtasks:

```
impl<L, R, T> ToHtml for Parallel<L, R, T>
where
    L: ToHtml + Send + Sync,
    R: ToHtml + Send + Sync,
    T: Send + Sync,
{
    async fn to_html(&self) -> Html {
        let left = self.left.to_html().await;
        let right = self.right.to_html().await;
        Html(format!("{left}<br/>{right}"))
    }
}
```

`Handler` forwards the event to both subtasks, and combines the feedback they return:

```
impl<L, R, T> Handler for Parallel<L, R, T>
where
    L: Handler + Send + Sync,
    R: Handler + Send + Sync,
```

```

T: Send + Sync,
{
    async fn on_event(&mut self, event: Event) -> Feedback {
        let left = self.left.on_event(event.clone()).await;
        let right = self.right.on_event(event).await;
        left.merged_with(right)
    }
}

```

For the `Value` implementation, the `Output` type of a parallel task depends on the combination function. For example, the `and` combinator turns two task values into one task value of a tuple, while the `left` combinator simply returns the first task value. We retrieve the values of both subtasks, and combine them using the `combine` field of the task:

```

impl<L, R, F, T> Value for Parallel<L, R, F>
where
    L: Value + Send + Sync,
    L::Output: Send,
    R: Value + Send + Sync,
    F: Fn(TaskValue<L::Output>, TaskValue<R::Output>) -> TaskValue<T> + Send + Sync,
    T: Send + Sync,
{
    type Output = T;

    async fn value(&self) -> TaskValue<Self::Output> {
        let left = self.left.value().await;
        let right = self.right.value().await;
        (self.combine)(left, right)
    }
}

```

This is essentially everything we need to get a working parallel task. Now, to add combinator methods for common combining procedures to any task, we use the *extension trait convention*.³ This involves defining a new trait with methods with default implementations, and implementing that trait for any type that implements the task abstraction:

```

type And<L, R> = fn(TaskValue<L>, TaskValue<R>) -> TaskValue<(L, R)>;
type Or<T> = fn(TaskValue<T>, TaskValue<T>) -> TaskValue<T>;
type Left<L, R> = fn(TaskValue<L>, TaskValue<R>) -> TaskValue<L>;
type Right<L, R> = fn(TaskValue<L>, TaskValue<R>) -> TaskValue<R>;

pub trait TaskParallelExt: Task + Sized {
    fn and<R>(self, right: R) -> Parallel<Self, R, And<Self::Output, R::Output>>
    where
        R: Value,

```

³<https://rust-lang.github.io/rfcs/0445-extension-trait-conventions.html>

```

{
  Parallel {
    left: self,
    right,
    combine: |left, right| match left {
      TaskValue::Stable(a) => match right {
        TaskValue::Stable(b) => TaskValue::Stable((a, b)),
        TaskValue::Unstable(b) => TaskValue::Unstable((a, b)),
        TaskValue::Empty => TaskValue::Empty,
      },
      TaskValue::Unstable(a) => match right {
        TaskValue::Stable(b) | TaskValue::Unstable(b) =>
↳ TaskValue::Unstable((a, b)),
        TaskValue::Empty => TaskValue::Empty,
      },
      TaskValue::Empty => TaskValue::Empty,
    },
  }
}

fn or<R>(self, right: R) -> Parallel<Self, R, Or<Self::Output>>
where
  R: Value<Output = Self::Output>,
{ /* ... */ }

fn left<R>(self, right: R) -> Parallel<Self, R, Left<Self::Output, R::Output>>
where
  R: Value,
{ /* ... */ }

fn right<R>(self, right: R) -> Parallel<Self, R, Right<Self::Output, R::Output>>
where
  R: Value,
{ /* ... */ }
}

impl<T> TaskParallelExt for T where T: Value {}

```

We can easily add support for other combining procedures by adding more methods, although there are only so many options with just two subtasks. We could also have users supply their own combining procedure for more fine-grained control:

```

pub trait TaskParallelExt: Task + Sized {
  // ...

  fn parallel<R, T>(
    self,
    right: R,
    combine: fn(TaskValue<L>, TaskValue<R>) -> TaskValue<T>
  ) -> Parallel<Self, R, fn(TaskValue<L>, TaskValue<R>) -> TaskValue<T>> {
    Parallel {
      left: self,

```

```

        right,
        combine,
    }
}

// ...
}

```

4.5.2 Design Choices and Alternatives

Infix operators

In `iTasks`, parallel tasks are created using custom infix operators, which Rust does not support. Rust does allow us to overload some existing operators, including `&`, `|`, `<<`, and `>>`. We could use these to replace `and`, `or`, `left`, and `right`, though we have no influence on the precedence of the operators, meaning `<<` and `>>` would bind stronger. Other than that, this is a viable alternative. The reason we went with methods instead, is because there are no overloadable operators that interact nicely with the sequential tasks from the next section. Using both methods and infix operators in the same task definition makes it hard to read, so we went with methods only for now.

One downside of this approach is that we cannot specify that `and` has precedence over `or`, for example; the right-hand side of the operator is always enclosed with parentheses. Instead, we can manipulate the order of operations by putting the combinator either inside the argument or after the call:

```

// `or` inside argument, has precedence over `and`
enter::


---



```

As an alternative, we could define `and`, `or`, `left`, and `right` *functions* with two task parameters, to act as prefix operators:

```

// Eager evaluation of arguments: `or` has precedence
and(enter::


---



```

This forces the programmer to think about the order of operations, rather than implicitly evaluating them from left to right. Both arguments of the function would also get the same level of indentation in large task definitions. When using methods, the right hand side would be indented one layer deeper due to the parentheses, which can hurt readability. On the other hand, the method approach allows IDEs to list all applicable

operators for a task, making it easier for the programmer to navigate and read documentation. Since there are only four of these functions, prefix operators might be better suited in this case, but it is mostly a matter of preference.

Two subtasks

Initially, we tried to implement a generic parallel task type that works for any number of subtasks, just like `iTasks`' parallel task. Such an implementation would look roughly as follows:

```
pub struct Parallel<F> {  
    tasks: Vec<Box<dyn Task>>,  
    combine: F,  
}
```

However, this relies heavily on dynamic dispatch. More is hidden in the `Value` implementation:

```
impl<F, T> Value for Parallel<F>  
where  
    F: Fn(Vec<TaskValue<Box<dyn Any>>>) -> TaskValue<T> + Send + Sync,  
    T: Send + Sync,  
{  
    type Output = T;  
  
    async fn value(&self) -> TaskValue<Self::Output> {  
        let values = future::join_all(self.tasks.iter().map(Value::value).map(Box::new))  
            .await;  
        (self.combine)(values)  
    }  
}
```

Dynamic dispatch imposes a small runtime overhead, but more importantly it makes the API harder to use. Rather than multiple statically typed arguments, we now have one vector argument containing boxed dynamic types. The combination function will have to create the task value by unboxing each of those values, which is unnecessarily verbose. Moreover, to do anything nontrivial, it must assume that the dynamic types are the same type as the `Output` of the task at the same index. The compiler cannot guarantee this at compile time. Trading runtime performance, ease-of-use, and type-safety for the ability to include more than two subtasks in a parallel task is hardly worth it. In the next chapter, we will explore an alternative that uses macros instead.

4.6 Sequential Tasks

Parallel tasks are useful for showing multiple tasks at the same time; sequential tasks are useful for showing multiple tasks *one after another*. They use the intermediate

value of their current task to create followup tasks for the user. Sequential tasks are essential to the library, because they allow the programmer to specify what to do with the intermediate values tasks hold. Unsurprisingly, they are also the most difficult type of task to implement.

4.6.1 Implementation

The definition of the sequential task looks as follows:

```

type Condition<A> = Box<dyn Fn(TaskValue<&A>) -> bool + Send + Sync>;
type Transform<A, B> = Box<dyn FnOnce(TaskValue<A>) -> DynTask<B> + Send + Sync>;

type DynTask<T> = Box<dyn Task<Output = T> + Send + Sync>;

pub struct Sequential<T, U>
where
    T: Value,
{
    container_id: Uuid,
    current: Either<T, DynTask<U>>,
    continuations: BTreeMap<Trigger, Continuation<T::Output, U>>,
}

struct Continuation<A, B> {
    condition: Condition<A>,
    transform: Transform<A, B>,
}

impl<T, U> Sequential<T, U>
where
    T: Value + Send + Sync,
    T::Output: Send + Sync,
    U: Send + Sync,
{
    async fn transform(&mut self, trigger: Trigger) -> Result<Feedback, TransformError>
    ↪ {
        match &self.current {
            Either::Left(task) => {
                let value = task.value().await;
                for continuation in self
                    .continuations
                    .iter_mut()
                    .filter(|(t, _)| **t == trigger)
                    .map(|(_, c)| c)
                {
                    if (continuation.condition)(value.as_ref()) {
                        let next = (continuation.transform.take().unwrap())(value);
                        let html = next.to_html().await;
                        self.current = Either::Right(next);
                        return Ok(Feedback::from(Change::ReplaceContent {
                            id: self.container_id,
                            html,
                        }));
                    }
                }
            }
        }
    }
}

```

```

        }));
    }
    }
    Err(TransformError::FalseConditions)
}
}
Either::Right(_) => Err(TransformError::InvalidState),
}
}
}
}

```

Note that `Sequential<T, U>` keeps track of the task it currently represents using the `Either` type. `Either` comes from the `either` crate, and is essentially an enum with two variants that hold two different types. A sequential task can either be in its pre-transform state and hold task `T`, or its post-transform state with a task that has an intermediate value of type `U`. The `transform` method triggers this state transition, by consuming the value of task `T` and creating the followup task using the transformation function. After transforming, a sequential task should become the followup task. However, it cannot simply replace itself with that task, as that would change its type. Therefore, it holds the followup task, and mimics it by forwarding events to it and returning its feedback.

This definition heavily relies on dynamic dispatch. The condition and transformation function require dynamic dispatch because closures are always different types, even if they are identical. If we were to use static dispatch through type parameters instead, we would only be able to specify one continuation, unless we use the same closures for all of them, which is rarely useful. The `DynTask<T>` type also uses dynamic dispatch, because we want each continuation to create a task with the same *output* type. The types of the tasks themselves do not have to be equal. For example, if we relied on static dispatch instead, the following example would not work:

```

async fn example() -> impl Task {
    enter::<String>()
        .step()
        .on(
            Trigger::Button(Button::new("View")),
            has_value,
            |value| view(value.unwrap()),
        )
        .on(
            Trigger::Button(Button::new("Judge")),
            has_value,
            |value| view("Great!").right(view(value.unwrap()))
        )
}

```

In this prototype, the `step` method wraps any task in a sequential task with no continuations. The `on` method can then add new continuations to it, boxing and casting the

closures to the right dynamic type. This would be a reasonable task definition, because all continuations produce a task with an intermediate `String` value. However, both transformation functions return different types of tasks, therefore we cannot store them in the same structure without dynamic dispatch.

The task trait implementations for sequential tasks depend on whether it is in its pre- or post-transform state. For `ToHtml`, the pre-transform task first needs to be put in a separate container, so that the task may remove and replace itself upon transform. Moreover, if the continuation triggers on the press of a button, that button should also appear in the HTML. On the other hand, if the task already transformed, we can simply return the HTML translation of the new task:

```
impl<T, U> ToHtml for Sequential<T, U>
where
  T: Value + ToHtml + Send + Sync,
  U: Send + Sync,
{
  async fn to_html(&self) -> Html {
    match &self.current {
      Either::Left(task) => {
        let buttons: Html = self
          .continuations
          .keys()
          .filter_map(|trigger| if let Trigger::Button(button) = trigger {
            Some(button)
          } else {
            None
          })
          .map(|button| Html(format!(
            r#"<button id="{button.id}" type="button" onclick="press(this)">
              {}
            </button>"#,
            button.id,
            button.text,
          )))
          .collect();

        Html(format!(
          r#"<div id="{self.container_id}">{}</div></div>"#,
          self.container_id,
          task.to_html().await,
          buttons
        ))
      }
      Either::Right(task) => task.to_html().await,
    }
  }
}
```

This pattern also occurs in the `Handler` and `Value` trait implementations. As soon as

a sequential task transforms, it should behave exactly as if it were the continued task. We can simply delegate the method call to that task's trait implementation:

```
impl<T, U> Handler for Sequential<T, U>
where
  T: Value + Handler + Send + Sync,
  T::Output: Send + Sync + 'static,
  U: Send + Sync + 'static,
{
  async fn on_event(&mut self, event: Event) -> Feedback {
    match &mut self.current {
      Either::Left(task) => match event {
        Event::Press { id: button_id } => match self
          .continuations
          .keys()
          .find(|trigger| {
            if let Trigger::Button(Button { id, .. }) = trigger {
              button_id == *id
            } else {
              false
            }
          })
          .cloned()
        {
          None => Feedback::new(),
          Some(trigger) => self.transform(trigger).await.unwrap_or_default(),
        },
      - => {
        let feedback = task.on_event(event.clone()).await;
        self.transform(Trigger::Update).await.unwrap_or(feedback)
      }
    },
    Either::Right(task) => task.on_event(event).await,
  }
}
```

This method will attempt to transform the task when one of its buttons is pressed, or when its value updates.

The `Value` implementation is straightforward: the intermediate value of a sequential task is simply the intermediate value of the continuation tasks. If the task is still in its pre-transform state, we cannot reliably create a value of the right type, therefore it returns an empty value:

```
impl<T, U> Value for Sequential<T, U>
where
  T: Value + Send + Sync,
  U: Send + Sync,
{
  type Output = U;
```

```

    async fn value(&self) -> TaskValue<Self::Output> {
        match &self.current {
            Either::Left(_) => TaskValue::Empty,
            Either::Right(task) => task.value().await,
        }
    }
}

```

Finally, to create the combinator, we first repeat the pattern we used for parallel tasks:

```

pub trait TaskSequentialExt: Value + Sized {
    fn step<U>(self) -> Sequential<Self, U> {
        Sequential {
            container_id: Uuid::new_v4(),
            current: Either::Left(self),
            continuations: BTreeMap::new(),
        }
    }
}

impl<T> TaskSequentialExt for T where T: Value {}

```

Then we add the `on` method to the `Sequential<T, U>` task directly:

```

impl<T, U> Sequential<T, U>
where
    T: Value,
{
    pub fn on<C, F, K>(mut self, trigger: Trigger, condition: C, transform: F) -> Self
    where
        C: Fn(TaskValue<&T::Output>) -> bool + Send + Sync + 'static,
        F: FnOnce(TaskValue<T::Output>) -> K + Send + Sync + 'static,
        K: Task<Output = U> + Sync + Send + 'static,
    {
        let continuation = Continuation {
            condition: Box::new(condition),
            transform: Box::new(move |value| Box::new(transform(value))),
        };
        self.continuations.insert(trigger, continuation);
        self
    }
}

```

To further simplify creating conditions, we add a few common variants to the public API:

```

pub fn if_stable<T>(value: TaskValue<&T>) -> bool {
    value.is_stable()
}

```

```

pub fn if_unstable<T>(value: TaskValue<&T>) -> bool {
    value.is_unstable()
}

pub fn if_empty<T>(value: TaskValue<&T>) -> bool {
    value.is_empty()
}

pub fn has_value<T>(value: TaskValue<&T>) -> bool {
    !value.is_empty()
}

pub fn if_value<F, T>(f: F) -> fn(TaskValue<&T>) -> bool
where
    F: FnOnce(&T) -> bool
{
    |value| value.map(f).unwrap_or_default()
}

pub fn always<T>(_: TaskValue<&T>) -> bool {
    true
}

```

4.6.2 Design Choices and Alternatives

Static lifetime requirement

By default, trait objects have a static lifetime: `dyn Fn(TaskValue<A>) -> DynTask` is implicitly `dyn Fn(TaskValue<A>) -> DynTask + 'static`. A closure only has a static lifetime if the captured variables from its environment also have a static lifetime. This implies that the following example would not compile for now:

```

fn foo(slice: &[u32]) -> impl Task {
    view(6)
        .step()
        .on(
            Trigger::Update,
            always,
            |_| view(slice.len())
        )
}

async fn bar() -> impl Task {
    foo(&[1, 2, 3])
}

```

It is entirely reasonable to create tasks that depend on non-static references; only being able to use ones with static lifetimes is a severe limitation. To remedy this, we have to add explicit lifetimes to our trait objects:

```

type Condition<'a, A> = Box<dyn Fn(TaskValue<&A>) -> bool + Send + Sync + 'a>;
type Transform<'a, A, B> = Box<dyn FnOnce(TaskValue<A>) -> DynTask<B> + Send + Sync +
↳ 'a>;

type DynTask<T> = Box<dyn Task<Output = T> + Send + Sync>;

pub struct Sequential<'a, T, U>
where
    T: Value,
{
    container_id: Id,
    current: Either<T, DynTask<U>>,
    continuations: BTreeMap<Trigger, Continuation<'a, T::Output, U>>,
}

struct Continuation<'a, A, B> {
    condition: Condition<'a, A>,
    transform: Transform<'a, A, B>,
}

```

The drawback of this approach is that we have to specify this lifetime in the return type of every sequential task, even when that task does not rely on non-static lifetimes:

```

async fn greet() -> impl Task + '_ {
    view("Hello, world!")
        .step()
}

```

Infinite recursion

Sequential tasks can be used to create recursive tasks:

```

fn infinity() -> Sequential<Return<i32>, i32> {
    return_value(42)
        .step()
        .on(Trigger::Button(Button::new("Next")), always, |_| infinity())
}

```

Pressing the button will yield an identical sequential task in its pre-transform state. As such, this task will never have an intermediate value. This gives the desired behavior when used with the `or` combinator:

```

fn infinity_or_seven() -> impl Task {
    infinity().or(return_value(7))
}

```

This task returns the value 7, the only terminating option.

However, one must exercise caution when defining recursive tasks. `Trigger::Update` will trigger every time the value updates, but also once upon creation. The latter is a conscious design decision. It allows sequential tasks to immediately transition when their current task starts with a valid value. However, if we replace the trigger in `infinity` with `Trigger::Update`, it will eagerly trigger continuations until a stack overflow occurs.

An alternative would be to have `Trigger::Update` only trigger when the value changes, and not on creation. The example above would still work in that case. However, tasks of which the values do not change will then never trigger the continuation. Therefore, the executor will have to manually check at least once whether new sequential tasks can trigger with their initial value or not, upon execution rather than creation. This is more work to implement appropriately, as new sequential tasks can be created at any moment. For that reason, and because the example above has little practical value, we chose the simpler implementation.

One continuation

Much like how we could remove dynamic dispatch from parallel tasks when we gave them a fixed number of subtasks, the sequential task can also be simplified by giving it exactly one continuation. However, in contrast with parallel tasks, this simpler variant is not equally powerful. It would look as follows:

```
pub struct Sequential<T, U, C, F> {
    container_id: Uuid,
    current: Either<T, U>,
    trigger: Trigger,
    condition: C,
    transform: F,
}

impl<T, U, C, F> Sequential<T, U, C, F>
where
    T: Value + Send + Sync,
    U: ToHtml + Send + Sync,
    C: Fn(&TaskValue<<T::Share as ShareConsume>::Value>) -> bool + Send + Sync,
    F: Fn(TaskValue<<T::Share as ShareConsume>::Value>) -> U + Send + Sync,
{
    async fn transform(&mut self) -> Result<Feedback, TransformError> {
        match &self.current {
            Either::Left(task) => {
                let value = task.value().await;
                if (self.condition)(value.as_ref()) {
                    let next = (self.transform)(value);
                    let html = next.to_html().await;
                    self.current = Either::Right(next);
                    Ok(Feedback::from(Change::ReplaceContent { id: self.id, html }))
                } else {
                    Err(TransformError::FalseCondition)
                }
            }
        }
    }
}
```

```

    }
    Either::Right(_) => Err(TransformError::InvalidState),
  }
}
}

```

The type parameters `C` and `F` keep track of the types of the condition and transformation function. Since there is only one of each, dynamic dispatch is no longer necessary. The followup task `U` can also be stored without dynamic dispatch, because there are no other followup tasks that might be a different type. Moreover, this implementation would allow us to remove the extra call to `step` from the public API:

```

async fn greet() -> impl Task {
  view("Please enter your name:")
    .right(enter::<String>())
    .on(
      Trigger::Update,
      |value| if_value(|name| name == "Bob"),
      |value| view("Welcome, Bob!".to_owned()),
    )
    .on(
      Trigger::Button(Button::new("Ok")),
      |value| if_value(|name| !name.is_empty()),
      |value| view(format!("Hello, {}!", value.unwrap())),
    )
}

```

However, this implementation is less flexible: it does not allow us to split the workflow into two separate sequences. Suppose the desired semantic of a task is that, when the user enters a name and presses ‘Ok’, they will see a message greeting that name. Moreover, when they enter the name “Bob”, the task will immediately welcome Bob, also removing the button. The example above would do just that if it stored multiple continuations in one task, and included a call to `step`.

Unfortunately, in the single-continuation variant, the first sequential task is stored inside the second. The second sequential task will therefore still display the ‘Ok’ button after the first task transitions. Pressing that button will then retrieve the value “Welcome, Bob!” as name, rather than the name the user entered. Worse still, when the user enters a different name than Bob, the second sequential task will retrieve the value from the first, which is empty because it has not transitioned yet. The button will be disabled until the user enters “Bob”.

This does not mean that the single-continuation variant of the sequential task is not useful. It is still slightly more performant, and does not require an extra lifetime in the return type. We can define an auxiliary `then` method, which adds a single continuation to any task using this more efficient approach:

```

async fn greet() -> impl Task {
  view("Please enter your name:")
    .right(enter::<String>())
    .then(
      Trigger::Button(Button::new("Ok")),
      |value| if_value(|name| !name.is_empty()),
      |value| view(format!("Hello, {}!", value.unwrap())),
    )
}

```

Split API methods

One might wonder why the public API consists of one method that turns a task into a sequential task without continuations, and another method to add a single continuation to it. One method which turns any task into a sequential task with multiple continuations would be more intuitive:

```

async fn greet() -> impl Task {
  view("Please enter your name:")
    .right(enter::<String>())
    .step(vec![
      Trigger::Update,
      |value| if_value(|name| name == "Bob"),
      |value| view("Welcome, Bob!".to_owned()),
    ], (
      Trigger::Button(Button::new("Ok")),
      |value| if_value(|name| !name.is_empty()),
      |value| view(format!("Hello, {}!", value.unwrap())),
    ))
}

```

Unfortunately, this will not compile, because we attempt to store different types of closures in the same vector (remember that different closures are always different types). It would work if the user boxes them first. The return type of the transformation function must also be boxed manually:

```

async fn greet() -> impl Task {
  view("Please enter your name:")
    .right(enter::<String>())
    .step(vec![
      Trigger::Update,
      Box::new(|value| if_value(|name| name == "Bob")),
      Box::new(|value| view("Welcome, Bob!".to_owned())),
    ], (
      Trigger::Button(Button::new("Ok")),
      Box::new(|value| if_value(|name| !name.is_empty())),
      Box::new(|value| Box::new(view(format!("Hello, {}!", value.unwrap())))),
    ))
}

```

```
}
```

It is a matter of personal preference which of these is more useful. We went with the less verbose version in an attempt to keep task definitions more readable.

Split condition and transform

In `iTasks`, the condition and transformation function are combined into one function. It takes the value of the current task, and optionally returns a followup task. If it does return a task, it will automatically transition to it, just like when our condition is true. This is often more ergonomic than our solution. For instance, if a condition checks whether the value is not empty, and the transformation function needs that value, it has to handle the case where it is empty again:

```
async fn greet() -> impl Task {
    enter::<char>()
        .then(
            Trigger::Update,
            has_value,
            |value| view(value.unwrap()),
        )
}
```

The `unwrap` method is similar to `Option::unwrap` and `Result::unwrap`: it assumes the enum contains a value and returns it. The program will panic if there is no value, therefore it is usually a better idea to handle each case.

`unwrap` is not necessarily unidiomatic in this case, because we assert that the value exists in the condition. However, if we merge the condition and transformation function, we could make `has_value` return a task, and there would be no need for `unwrap` at all:

```
async fn greet() -> impl Task {
    enter::<char>()
        .then(
            Trigger::Button(Button::new("Ok")),
            |value| has_value(view),
        )
}
```

The reason we split the condition and transformation function regardless is due to Rust's ownership system. The condition receives a `TaskValue<&T::Output>` as parameter. It only needs a shared reference of the value to determine whether or not the task should transition, there is no reason to mutate the data when doing so. In contrast, the transformation function gets a `TaskValue<T::Output>`, making it the new owner of the value of the current task. This allows it to create a new task – which requires ownership over the value – without cloning the value.

If we merge the condition and transformation function, we would have to choose between using `TaskValue<&T::Output>` and `TaskValue<T::Output>` as parameter. The former requires us to clone the value if we want to create a followup task with it. The latter forces the sequential task to clone the borrowed value of the current task every time it triggers, because it cannot determine whether the task can be consumed before the function call. Splitting the condition allows the sequential task to first borrow the value of the current task. Then, if the condition passes, it can safely consume the current task, giving it ownership over the value without cloning it. It can then pass this value to the transformation function. This is more efficient, at the cost of being less ergonomic.

4.7 Executor

We conclude this chapter with a short overview of how we execute tasks in this library. Note that this is highly dependent on which type of front-end we choose to support, and which supporting libraries we use. As such, it should be considered as merely an example.

As explained in the API section, we defined a `task` function to serve tasks. This function essentially combines two services: one to fetch a template web page, and one to connect a websocket to the client:

```
pub fn task<H, T>(handler: H) -> TaskRouter
where
  H: FnOnce() -> T + Clone + Send + 'static,
  T: Task + Send + Sync + 'static,
{
  let wrapper = get(wrapper);
  let connect = get(|ws| connect(ws, handler()));

  TaskRouter { wrapper, connect }
}

pub struct TaskRouter {
  wrapper: MethodRouter<Body, Infallible>,
  connect: MethodRouter<Body, Infallible>,
}
```

`axum` requires that all services implement the `Service` trait. Since we rely on two existing services, this implementation is quite standard. It only inspects the request to find out whether the client requests the web page or the websocket, and forwards it to the appropriate service:

```
impl Service<Request<Body>> for TaskRouter {
  type Response = <MethodRouter as Service<Request<Body>>>::Response;
  type Error = <MethodRouter as Service<Request<Body>>>::Error;
  type Future = <MethodRouter as Service<Request<Body>>>::Future;
```

```

fn poll_ready(&mut self, _cx: &mut std::task::Context<'_>) -> Poll<Result<(),
↳ Self::Error>> {
    Poll::Ready(Ok(()))
}

fn call(&mut self, req: Request<Body>) -> Self::Future {
    match req.headers().get("upgrade") {
        Some(header) if header == "websocket" => self.connect.call(req),
        _ => self.wrapper.call(req),
    }
}
}

```

When the user visits the page normally, the server will respond with a template web page:

```

async fn wrapper() -> impl IntoResponse {
    Html(format!(
        r#"
        <!DOCTYPE html>
        <html lang="en">
            <head>
                <meta charset="utf-8">
                <title>TOP Axum</title>
                <script src="top/top.js"></script>
            </head>
            <body>
                <section class="section">
                    <div class="container">
                        <div id="00000000-0000-0000-0000-000000000000"></div>
                    </div>
                </section>
            </body>
        </html>
        "#
    ))
}

```

We can structure this page however we want, as long as it contains the following components:

- Code to connect to the websocket service
- The JavaScript functions **update** and **press**, as introduced earlier in this chapter, to send **Events** to the server
- A procedure to update the page asynchronously, according to the **Feedback** returned by the websocket
- An element to contain the executed task, with the ‘nil UUID’ as identifier

When the client connects to the server through a websocket, the service attached to the websocket simply sends the HTML representation of the executed task. The client will then insert this HTML in the nil UUID container, allowing the user to interact with it. When they do so, the server will pass the transmitted `Event` to the task, and return its `Feedback`:

```

async fn connect<T>(ws: WebSocketUpgrade, task: T) -> impl IntoResponse
where
    T: Task + Send + Sync + 'static,
{
    ws.on_upgrade(|socket| async move {
        let (mut sender, mut receiver) = socket.split();

        // Transmit initial page
        let html = task.to_html().await;
        let feedback = Feedback::from(Change::AppendContent {
            id: Uuid::nil(),
            html,
        });
        send_feedback(&mut sender, feedback).await;

        // Respond to input
        while let Some(Ok(message)) = receiver.next().await {
            if let Ok(text) = message.into_text() {
                match serde_json::from_str(&text) {
                    Ok(event) => {
                        // Received event
                        let feedback = task.on_event(event).await;
                        if !feedback.is_empty() {
                            // Transmit feedback
                            send_feedback(&mut sender, feedback).await;
                        }
                    }
                    Err(_) => warn!("not an event: {}", text),
                }
            }
        }
    })
}

async fn send_feedback(sender: &mut SplitSink<WebSocket, Message>, feedback: Feedback) {
    match serde_json::to_string(&feedback.changes()) {
        Ok(text) => sender
            .send(Message::Text(text))
            .await
            .unwrap_or_else(|error| error!("failed to send feedback: {error}")),
        Err(error) => error!("failed to serialize: {error}"),
    }
}

```

This concludes our initial implementation of TOP in Rust. The library currently allows

programmers to construct tasks using a relatively simple API, and have users interact with them through a web interface. While it supports the interaction tasks `view` and `edit`, they currently only work with primitive types by default. Parallel tasks allow constructing more complicated types, but adding full support for new types would require manually implementing new types of tasks. Sequential tasks are also functional, but inefficient because tasks clone their intermediate value whenever it is retrieved. Additionally, it is not yet possible to share values between tasks, which makes this library significantly less powerful than `iTasks`. We will address these issues using Rust-specific features in the next chapter.

Chapter 5

Analyzing Drawbacks and Improving the Library

The implementation from the previous chapter has a few major limitations when compared to a mature framework like `iTasks`, such as not being able to use every conceivable type in interactive tasks, or share values between tasks. Additionally, it does not take full advantage of Rust's low-level memory control that could allow it to be more performant. This chapter attempts to improve these aspects of the naive implementation by using Rust language features that are substantially different from other programming languages. While not all limitations are lifted completely at the end of this chapter, we aim to provide more insight in the advantages and limitations of Rust for task-oriented programming.

5.1 Borrowing Task Values

As described in the previous chapter, `view` and `edit` tasks clone their value whenever the `value` method is called. This happens every time a sequential task's trigger goes off, even though an owned value is only necessary for the actual transformation. It would be more efficient if sequential tasks could borrow the value of the current task to check if the condition holds. If that is the case, the sequential task could then gain ownership over the value by consuming its current task, allowing it to use the value without cloning to create the followup task. However, implementing this behavior for all tasks is harder than it seems.

To illustrate, consider this naive adaptation of the `Value` trait:

```
pub trait Value {  
    type Output;  
  
    async fn borrow(&self) -> TaskValue<&Self::Output>;  
}
```

```

    async fn consume(self) -> TaskValue<Self::Output>;
}

```

It would easily accomplish our goal for view and edit tasks:

```

impl<T> Value for ViewValue<T>
where
    T: Send + Sync,
{
    type Output = T;

    async fn borrow(&self) -> TaskValue<&Self::Output> {
        TaskValue::Stable(&self.value)
    }

    async fn consume(self) -> TaskValue<Self::Output> {
        TaskValue::Stable(self.value)
    }
}

```

However, it creates a problem for composed tasks, such as a view task for tuples:

```

impl<L, R> Value for ViewTuple<L, R>
where
    L: Value + Send + Sync,
    L::Output: Clone + Send,
    R: Value + Send + Sync,
    R::Output: Clone,
{
    type Output = (L::Output, R::Output);

    async fn borrow(&self) -> TaskValue<&Self::Output> {
        let left = self.0.borrow().await;
        let right = self.1.borrow().await;
        // left.clone().and(right.clone()).as_ref()
        panic!("dangling reference")
    }

    async fn consume(self) -> TaskValue<Self::Output> {
        let left = self.0.consume().await;
        let right = self.1.consume().await;
        left.and(right)
    }
}

```

The method should return a reference to a tuple of owned values. However, doing so is a challenge. We first need ownership over the values of both subtasks, which we can achieve by cloning them. This is not efficient, and unnecessary because we do not necessarily need ownership to check the condition, but it works for argument's sake.

We can then store these cloned values in a tuple, and finally, return a reference to that tuple. This results in a dangling reference, because the tuple is dropped at the end of the function, after which the reference would point to invalid memory. As a result, the borrow checker will reject this implementation.

It is possible to work around this problem. Note that the method signature contains two references. We did not specify the lifetime of either reference, because the compiler determines through lifetime elision that both references should have the same lifetime:

```
async fn borrow<'a>(&'a self) -> TaskValue<&'a Self::Output> { /* ... */ }
```

This gives us a hint on how to solve the problem. The implementation above is invalid because the created reference does not outlive the borrow of `self`. To fix that, we could store the tuple in `self`, and return a reference to that instead. However, this requires adding an extra field to every composed task, which adds to complexity. Moreover, those tasks would need to clone their subtasks' values every time they are used, which is barely an improvement over the implementation in the previous chapter.

Instead of returning a reference to a tuple, it would be convenient if the borrow method returned a tuple of references. This is a completely different type than a tuple of values, so to accommodate for this, the `Value` trait would need a second associated type:

```
impl<L, R> Value for ViewTuple<L, R>
where
    L: Value + Send + Sync,
    L::Output: Send,
    R: Value + Send + Sync,
{
    type Borrow<'a> = (&'a L::Output, &'a R::Output);
    type Consume = (L::Output, R::Output);

    async fn borrow(&self) -> TaskValue<&Self::Output> {
        let left = self.0.borrow().await;
        let right = self.1.borrow().await;
        left.and(right)
    }

    async fn consume(self) -> TaskValue<Self::Consume> {
        let left = self.0.consume().await;
        let right = self.1.consume().await;
        left.and(right)
    }
}
```

This solution removes the need for clones and extra fields, making it ideal for our use case. Unfortunately, it does not compile yet. The implementation depends on *Generic Associated Types* (GATs), a feature that is still unstable. Without it, we cannot add

lifetimes, or any other parameters, to the associated type `Borrow<'a>`.¹

GATs will probably be stabilized in the near future, and we can already use them with the nightly compiler. However, they come with a limitation that kills our use case: traits with GATs are not object-safe. Since our implementation of sequential tasks relies on trait objects, they would no longer work. While this limitation will likely be removed in the future, GATs have been under development since 2016, and have proven to be quite difficult to implement. It might take a while before they become object-safe.

5.2 Macros

In chapter 4, we demonstrated how one might implement simple view and edit tasks for primitive types, as well as more advanced tasks for vectors for instance. The principles described there are sufficient for every conceivable type, but implementing them manually for every type is inconceivable. `iTasks` solves this problem with generic functions that work for any type. For instance, by looking at the structure of a type, it can determine how to generate an HTML representation for it. Rust cannot do that exactly, but we can get close by using macros.

5.2.1 Declarative Macros

Declarative macros are especially useful to reduce boilerplate code within the library itself. For example, it would be useful to have a view task for tuples consisting of types that also have view tasks. However, tuples come in many different sizes, each of which is a different type. Rust implements the `Debug` trait for tuples with up to twelve items. If we want to create a view task for each of those, including unit `()` and the singleton `(T,)` for completeness, that requires thirteen slightly different implementations. This takes well over a thousand lines of code, with a lot of duplication and room for mistakes. The problem gets worse if we choose to repeat this pattern for edit tasks, and potentially others.

A declarative macro allows us to generalize the implementation for tuples of any size:

```
macro_rules! view_tuple {
    ($name:ident<$(($param:ident,)*>)* => {
        #[derive(Debug)]
        pub struct $name<$(($param,)*> {
            value: ($(($param),*),
        }

        impl<$(($param,)*> $name<$(($param,)*> {
            pub fn new(value: ($(($param),*)) -> Self {
                $name { value }
            }
        }
    }
```

¹<https://blog.rust-lang.org/2021/08/03/GATs-stabilization-push.html>

```

impl<$(($param),)*> ToHtml for $name<$(($param),)*>
where
  $(($param: ToHtml + Send + Sync),*
{
  async fn to_html(&self) -> Html {
    Html(format!(
      "{}",
      Html::default()$(
        .add(self.value.${index()}.to_html().await) // $param
      )*
    ))
  }
}

impl<$(($param),)*> Handler for $name<$(($param),)*>
where
  $(($param: Send),*
{
  async fn on_event(&mut self, _event: Event) -> Feedback {
    Feedback::new()
  }
}

impl<$(($param),)*> Value for $name<$(($param),)*>
where
  $(($param: Clone + Send + Sync),*
{
  type Output = $(($param),*);

  async fn value(&self) -> TaskValue<Self::Output> {
    TaskValue::Stable(self.value.clone())
  }
}

impl<$(($param),)*> View for (($param),*)
where
  $(($param: Clone + Send + Sync),*
{
  type Task = $name<$(($param),)*>;

  fn view(self) -> Self::Task {
    $name::new(self)
  }
}
}
}

```

This may look daunting at first, but declarative macros are essentially just more powerful match statements. The `($name:ident<$(($param:ident),)*>)` at the top is the first, and only, case that the macro will check its input against. It consists of an identifier that specifies the name of the task, followed by any number of comma-separated type

parameters between angled brackets. `ViewCouple<A, B>`, for example, will bind `name` to `ViewCouple`, and `param` to a sequence of `A` and `B`.

Looping over sequences within a macro is done in a similar way: `$($param,)*` will expand to `A, B` with the input above. We can also add more tokens between the parentheses, and indicate that we want a trailing comma by putting the comma right before the asterisk. For example, `where $($param: Clone),*` will expand to:

```
where
  A: Clone,
  B: Clone,
```

That is all there is to declarative macros. To illustrate, the macro expansion of the type definition looks as follows:

```
#[derive(Debug)]
pub struct ViewCouple<A, B> {
    value: (A, B,),
}
```

The trailing comma in the tuple is to make sure that the singleton `(A,)` is also recognized as tuple. Now we can generate a new view task for a tuple of any specific size in a single line:

```
view_tuple!(ViewUnit<>);
view_tuple!(ViewMonuple<A>);
view_tuple!(ViewCouple<A, B>);
view_tuple!(ViewTriple<A, B, C>);
view_tuple!(ViewQuadruple<A, B, C, D>);
view_tuple!(ViewQuintuple<A, B, C, D, E>);
view_tuple!(ViewSextuple<A, B, C, D, E, F>);
view_tuple!(ViewSeptuple<A, B, C, D, E, F, G>);
view_tuple!(ViewOctuple<A, B, C, D, E, F, G, H>);
view_tuple!(ViewNonuple<A, B, C, D, E, F, G, H, I>);
view_tuple!(ViewDecuple<A, B, C, D, E, F, G, H, I, J>);
view_tuple!(ViewUndecuple<A, B, C, D, E, F, G, H, I, J, K>);
view_tuple!(ViewDuodecuple<A, B, C, D, E, F, G, H, I, J, K, L>);
```

We can use the same strategy to improve the parallel tasks from the previous chapter. It currently only supports exactly two subtasks, and the dynamic dispatch alternative that supported more than that, sacrificed type-safety, simplicity, and runtime performance. Declarative macros can create parallel tasks of multiple different sizes, without relying on dynamic dispatch. Alternatively, they could rewrite higher-level task definitions to use the simple two-subtask variant. For instance, an `all!` macro that expands to multiple `.and()` calls, or an `any!` macro for `.or()`. This can make some task definitions easier to read as well:

```
fn greet() -> impl Task {
    all!(
        view("Hello"),
        view("Welcome"),
        view("Goodbye"),
    )
}
```

5.2.2 Procedural Macros

Now we can create view and edit tasks for most types in the standard library. If the user wants to use their own types in TOP, however, they would have to create new task types, task trait implementations, and `View` and `Edit` implementations for each of them as well. We can simplify this process by creating a custom derive macro that automatically generates a suitable default view and edit task for a type.

There is no domain specific language to define procedural macros. Instead, we create a separate library, `top_derive`, and define our derive macro there:

```
#[proc_macro_derive(View)]
pub fn view_derive(input: TokenStream) -> TokenStream {
    // ...
}
```

The input `TokenStream` contains the definition of the struct or enum that the derive macro is used on. We need to parse it, and use this definition to generate another `TokenStream` containing code that allows us to use the type in view tasks. For any struct `Foo`, deriving `View` requires doing the following:

1. Generate a `ViewFoo` struct, containing the default view tasks for each of the fields
2. Implement `ToHtml` for `ViewFoo`, which combines the HTML of the subtasks
3. Implement `Handler`, which does nothing for `View` but would forward events to subtasks for the `Edit` derive
4. Implement `Value`, which constructs `Foo` from the values of each of the subtasks
5. Implement the `View` trait for `Foo`, to specify that `ViewFoo` is its default view task

We use the `syn` crate to parse the input stream as an abstract syntax tree. This allows us to reason about the data type in question using simple match expressions. To generate a `TokenStream` from template Rust code, we use the `quote` crate. These crates are the de facto standard for writing procedural macros in Rust. As an example, let us start by supporting classic structs with named fields:

```
#[proc_macro_derive(View)]
```

```

pub fn view_derive(input: TokenStream) -> TokenStream {
    let ast: DeriveInput = parse_macro_input!(input as DeriveInput);
    impl_view(ast).into()
}

fn impl_view(ast: DeriveInput) -> TokenStream2 {
    match ast.data {
        Data::Struct(data_struct) => impl_view_struct(ast.ident, data_struct,
↳ ast.generics),
        Data::Enum(_) => todo!("enums are not yet supported"),
        Data::Union(_) => panic!("unions are not supported"),
    }
}

fn impl_view_struct(ident: Ident, data_struct: DataStruct, generics: Generics) ->
↳ TokenStream2 {
    match data_struct.fields {
        Fields::Named(fields) => impl_view_classic_struct(ident, fields, generics),
        Fields::Unnamed(fields) => todo!("tuple structs are not yet supported"),
        Fields::Unit => todo!("unit structs are not yet supported"),
    }
}

fn impl_view_classic_struct(ident: Ident, fields: FieldsNamed, generics: Generics) ->
↳ TokenStream2 {
    let view_ident = format_ident!("View{ident}");

    let task = gen_task(&view_ident, &fields, &generics);
    let to_html = impl_to_html(&view_ident, &fields, &generics);
    let handler = impl_handler(&view_ident, &generics);
    let value = impl_value(&ident, &view_ident, &fields, &generics);
    let view = impl_view(&ident, &view_ident, &fields, &generics);

    quote! {
        #task
        #to_html
        #handler
        #value
        #view
    }
}

```

Note that `quote` returns a different type of token stream (`TokenStream2`) than the `derive` macro expects. This is because it keeps track of extra parameters that make it easier to work with. In the end, we convert it to the right `TokenStream` type using `.into()`. The code above forwards the necessary information to five functions. Each of those functions performs one of the five requirements of deriving `View`, as listed above. We will now elaborate on each of them, in order:

```

fn gen_task(task_ident: &Ident, fields: &FieldsNamed, generics: &Generics) ->
↳ TokenStream2 {

```



```

let (field_ids, field_types): (Vec<_>, Vec<_>) = fields
    .named
    .iter()
    .map(|field| (&field.ident, &field.ty))
    .unzip();

quote! {
    #[derive(Debug)]
    pub struct #task_ident #generics {
        #(#field_ids: <#field_types as View>::Task),*
    }
}

```

This generates the `ViewFoo` definition, which has the same field names, but contains the default view task of each type rather than the type itself. Notice how the syntax of including variables in the `quote!` macro is similar to that of declarative macros. It only replaces the `$` with `#` to allow generating declarative macros inside these procedural macros as well.

Implementing `ToHtml` for this structs is straightforward, we simply concatenate the HTML of all subtasks:

```

fn impl_to_html(task_ident: &Ident, fields: &FieldsNamed, generics: &Generics) ->
    ↳ TokenStream2 {
    let field_ids = fields
        .named
        .iter()
        .map(|field| &field.ident);
    let (impl_generics, ty_generics, where_clause) = generics.split_for_impl();

    quote! {
        impl #impl_generics ToHtml for #task_ident #ty_generics #where_clause {
            async fn to_html(&self) -> Html {
                Html(format!(
                    "<div>{}</div>",
                    Html::default()
                        #(.add(self.#field_ids.to_html().await))*
                ))
            }
        }
    }
}

```

The `split_for_impl` method splits the declaration of generic type parameters from their usage as parameters. For instance, `struct Foo<A: Clone, B> where B: Clone` will be split into:

- `<A: Clone, B>` as declaration

- `<A, B>` for their usage in `Foo<A, B>`
- In this case `where B: Clone`, because a `where` clause is present

This is a common pattern in macros. It allows us to support parameterized types with minimal effort.

The `Handler` implementation is trivial for view tasks:

```
fn impl_handler(task_ident: &Ident, generics: &Generics) -> TokenStream2 {
    let (impl_generics, ty_generics, where_clause) = generics.split_for_impl();

    quote! {
        impl #impl_generics Handler for #task_ident #ty_generics #where_clause {
            async fn on_event(&mut self, _event: Event) -> Feedback {
                Feedback::new()
            }
        }
    }
}
```

The `Value` implementation retrieves the value of each subtask, and constructs the type from those values:

```
fn impl_value(ident: &Ident, task_ident: &Ident, fields: &FieldsNamed, generics:
↳ &Generics) -> TokenStream2 {
    let field_idsents = fields
        .named
        .iter()
        .map(|field| &field.ident);
    let (impl_generics, ty_generics, where_clause) = generics.split_for_impl();

    quote! {
        impl #impl_generics Value for #task_ident #ty_generics #where_clause {
            type Output = #ident #ty_generics;

            async fn value(&self) -> TaskValue<Self::Output> {
                #(&let #field_idsents = self.#field_idsents.value().await;)*

                let value = #ident {
                    #(&field_idsents,)*
                };

                TaskValue::Stable(value)
            }
        }
    }
}
```

Note that we include `ty_generics` in the associated type as well. We retrieved those type parameters from the type deriving `View`, and so we need to use them everywhere

that type occurs in our code.

Finally, the `View` implementation constructs the task type we generated earlier:

```
fn impl_view(ident: &Ident, task_ident: &Ident, fields: &FieldsNamed, generics:
↳ &Generics) -> TokenStream2 {
    let field_idsents = fields
        .named
        .iter()
        .map(|field| &field.ident);
    let (impl_generics, ty_generics, where_clause) = generics.split_for_impl();

    quote! {
        impl #impl_generics View for #ident #ty_generics #where_clause {
            type Task = #task_ident #ty_generics;

            fn view(self) -> Self::Task {
                #task_ident {
                    #(#field_idsents: self.#field_idsents.view(),)*
                }
            }
        }
    }
}
```

We can repeat these steps to implement an `Edit` derive macro as well. Supporting other types of structs is straightforward. Tuple structs use indices rather than field names, the remainder of the code is equivalent. Unit structs do not have fields at all, making their implementation trivial, although it does raise the question how useful view and edit tasks are for those types.

Adding support for enums requires a bit more work. Viewing an enum is essentially no different from viewing a struct, perhaps with an extra label to indicate the name of the variant. Editing an enum, however, requires a task that allows the user to select a variant, and then provide them with a task to edit that variant. Both of these subtasks are simple, the former is a *choose* task that was shortly introduced in the previous chapter, while the latter is equivalent to a normal edit task for structs. Combining these is not a problem: changing the value of the choose task should produce `Feedback` that replaces the current variant form with a new one. However, derive macros tend to get large quickly, and any changes in the library result in having to rewrite large chunks of this code. Due to this and timing constraints, we did not implement the `Edit` derive macro for enums.

Nevertheless, using custom structs in interaction tasks is now short and elegant:

```
#[derive(Edit)]
struct Person {
    name: String,
    age: u8,
```

```

}

async fn greet() -> impl Task {
  enter::<Person>()
    .then()
    .on(
      Trigger::Button(Button::new("Ok")),
      has_value,
      |value| view(format!("Hello, {}!", value.unwrap().name)),
    )
}

```

Note that `View` and `Edit` can only be derived if all of the fields of the struct also implement `View` and `Edit` respectively. If that is not the case, or if we simply want to use a different task, we could theoretically improve the `derive` macro to support annotations:

```

#[derive(Edit)]
struct Person {
  #[top(edit: Choose<String> = Choose::new(value, vec!["Alice".to_string(),
↪ "Bob".to_string()]))]
  name: String,
  #[top(edit: EditRange<u8> = EditRange::new(value, 0, 100))]
  age: u8,
}

```

If a field has such an annotation, the `View` implementation should replace the call to `self.#field_idents.view()` with the provided task, and similar for `Edit`. It should also replace the `<#field_types as View>::Task` type with the annotated type in the task definition. This is an excellent way to generate customized tasks for new types, with few lines of code.

One drawback of this approach is that it is not possible to generate more than one view and one edit tasks for a type. The macro requires a `TokenStream` of the type definition to work, therefore only annotations present on the type itself can generate such tasks. While we could add support for secondary and tertiary task derives, this would clutter the type definition with annotations, and make the `derive` macro more convoluted.

Another major drawback of the `derive` macro is that we cannot use it on foreign types. If a program depends on types from another library, and that library does not implement `View` or `Edit` for its types, we have no choice but to implement them manually. Because this will likely be a common problem, we devised a workaround. Suppose another library features the `Foo` type:

```

mod other_library {
  // Foreign type, does not implement `Edit`
  pub struct Foo {
    bar: i32,

```

```

    }

    impl Foo {
        pub fn new(bar: i32) -> Self {
            Foo { bar }
        }
    }
}

```

While we could construct a value of type `Foo` using an edit task for `i32`, and display its `bar` field using a view task, we cannot use the type in tasks directly. This becomes problematic if we want to include `Foo` in other types that we want to use with tasks:

```

use other_library::Foo;

#[derive(Edit)]
pub struct Baz {
    // `Foo` does not implement `Edit`
    // foo: Foo,
}

```

It is still possible to construct `Baz` using an `i32` task, but it becomes more and more verbose. Instead, we can create a *builder* for the `Foo` type, which contains enough information to construct a value of type `Foo` using the builtin `From` trait. Because the builder definition is in our own crate, we can derive `Edit` for it. The idea is to use the builder type in our tasks, using a special `EditFrom` task that automatically converts the input and output values of the task from and into `Foo`:

```

#[derive(Edit)]
pub struct FooBuilder {
    bar: i32
}

impl From<FooBuilder> for Foo {
    fn from(builder: FooBuilder) -> Foo {
        Foo::new(builder.bar)
    }
}

impl From<Foo> for FooBuilder {
    fn from(foo: Foo) -> FooBuilder {
        FooBuilder { bar: foo.bar }
    }
}

async fn qux() -> impl Task<Output = Foo> {
    EditFrom::<FooBuilder>::new(Some(Foo::new(5)))
}

```

While still a lot of boilerplate, it is less work than manually creating an `EditFoo` task, and implementing `Edit` for `Foo` to assign it as default edit task for the type. `EditFrom` requires both `From` implementations, but the analogous `ViewInto` task only requires the `From<Foo>` implementation for `FooBuilder`, which automatically implements `Into<FooBuilder>` for `Foo`. This reduces the amount of boilerplate if we only need a view task.

Now we can use the `Foo` type in other types while still allowing them to derive `Edit`:

```
#[derive(Edit)]
pub struct Baz {
    #[top(edit: EditFrom<FooBuilder, Foo> = EditFrom::<FooBuilder>::new(value))]
    foo: Foo,
}
```

A small addition to the annotation could simplify this common pattern:

```
#[derive(Edit)]
pub struct Baz {
    #[top(edit_from: FooBuilder)]
    foo: Foo,
}
```

Similar to `From` and `Into`, Rust's standard library contains the `FromStr` and `Display` traits to parse strings into other types, and turn them back into strings. We can use the same strategy to create `EditFromStr` and `ViewDisplay` tasks. In this case, there is no need for a separate builder type, because that type is always `String`. Since many types in other libraries already implement `FromStr` and `Display`, using those in tasks is now quite simple:

```
#[derive(View, Edit)]
pub struct Config {
    #[top(view_display, edit_from_str)]
    addr: SocketAddr,
}
```

The default edit task for `Config` now shows a single text field to the user, in which they should enter a socket address such as `127.0.0.1:8000`. If they do not enter a valid socket address, `FromStr`'s associated type `Err` allows us to display a fitting error message, to help the user figure out why their input is invalid.

Another pair of potentially useful conversion tasks would be `EditDeserialize` and `ViewSerialize`. `serde`'s `Serialize` and `Deserialize` traits contain detailed information about the structure of a type, and are already derived for many types across the entire Rust ecosystem. If we can automatically derive fitting tasks for types that implement these traits, there would be few cases where manual task implementations

are still necessary. However, we did not dive into this further during this research.

The `derive` macro is not quite as powerful as `iTask`'s generic solution that works for all types. However, its addition makes this library flexible enough to be useful for many applications. With the library itself providing tasks for standard types, the `derive` macro taking care of new types, and the conversion tasks tackling foreign types, we essentially cover every type as well, albeit more verbose.

5.3 Shared Data Sources

Finally, we made an attempt to add shared data sources to our library. This proved to be quite a challenge, with many discarded prototypes along the way. While the end result is functional, it is not without drawbacks, and many types of shares are complex to implement this way. This section is split into three subsections. In the first subsection, we explain how data is generally shared between different structures in Rust. Then we go over the successful aspects of our shared data source implementation. Finally, we elaborate on a less successful example of a share for a task with a variable number of subtasks.

5.3.1 Sharing Data in Rust

Sharing values between different tasks works a little differently in Rust than in other languages due to its ownership model. If any task wants to mutate the value at some point, it would need to store a mutable reference to that data, which blocks all other tasks from storing even a shared reference to the same value. To work around this, without having to worry about potential data races, we can wrap the value in a `Mutex<T>`. The mutex allows us to get a mutable reference to its inner value, even if we only have a shared reference to the mutex:

```
fn main() {
    let mutex = Mutex::new(5);
    let immutable_borrow = &mutex;
    let mut lock: MutexGuard<'_, i32> = immutable_borrow.lock();
    *lock += 1;
    // Prints 6
    println!("{}", *lock);
}
```

This is called the *interior mutability pattern*. The mutex guarantees that the data is not accessed concurrently, and therefore it is completely safe to use, despite breaking the borrow checker's rules on the inside. One can think of it as guaranteeing the rules at runtime rather than at compile time. Note that we never actually access task values concurrently in our implementation; we only use the mutex to convince the compiler that it is safe to store multiple mutable references.

Mutexes are often used in combination with `Arc<T>`, a reference-counting pointer. We can clone an `Arc` to gain immutable access to the value within. The `Arc` then increments an internal counter that signifies how many such instances exist currently. Whenever one of them goes out of scope, it decrements the counter, and once the counter reaches zero, the inner data is freed:

```
fn main() {
    let arc = Arc::new(Some(5));    // 1 reference
    let other = arc.clone();        // 2 references
    std::mem::drop(arc.unwrap());   // 1 reference
    println!("{}", other);
}                                   // 0 references -> drop value
```

Note that `other` remained accessible, even though the apparent owner of the value (`arc`) went out of scope. `Arc` enables multiple variables to be the owner of the same value.

An `Arc` is immutable, but combined with mutexes, an `Arc<Mutex<T>>` allows us to store values in multiple places, and also mutate their inner values in a safe way. This is useful for our library, as it allows us to store shared values directly in the tasks that use them, rather than store references to a separate data structure that holds all shared values.

5.3.2 Implementing Shared Data Sources

In our first attempt, let us start by defining a trait that describes shared data source behavior:

```
pub trait Share {
    type Value;

    fn value(&self) -> MutexGuard<TaskValue<Self::Value>>;
}
```

Essentially, a shared data source is just a value of some type. Note that the `value` method returns a `MutexGuard`, which is a smart pointer that can be used as a mutable reference, but remembers to unlock the mutex when it goes out of scope.

We also added a `ShareUpdate` trait to determine whether the value of the share has changed since the last update. This is not strictly necessary, but a useful optimization:

```
pub trait ShareUpdate {
    fn updated(&self, ids: &BTreeSet<Uuid>) -> bool;
}
```

Now we need to adapt existing task definitions to use shared values rather than owned values. Doing so is straightforward for value-based tasks, such as `ViewValue<T>`:

```

struct ViewValue<S> {
    id: Uuid,
    share: S,
}

impl<S> ViewValue<S> {
    pub fn new(share: S) -> Self {
        ViewValue {
            id: Uuid::new_v4(),
            share,
        }
    }
}

```

The new corresponding `Value` implementation simply returns the value of the shared data source:

```

impl<S> Value for ViewValue<S>
where
    S::Value: Clone + Send + Sync,
{
    type Output = S::Value;

    async fn value(&self) -> TaskValue<Self::Output> {
        self.share.value().clone()
    }
}

```

Note that the `share` now determines whether the value is stable or not, as it returns the `TaskValue<T>` directly. For most types of shares, this will never be stable; only stable or empty. This is different from our initial implementation of view tasks, which produced stable values. We cannot guarantee that the value does not change, because it might be shared with an edit task that can still change it, for example. If we want to create a stable task value, we can use the `return_value` task. Its implementation is trivial: it has no HTML representation and does not respond to events, it simply returns a stable value.

When the value of the shared data source changes, we need to update this task's representation on the front-end. `iTasks` achieves this through an observer pattern: tasks can subscribe to shared data sources, and when their value changes, they get notified and can change their representation accordingly. Doing the same in Rust would be quite challenging. An observer pattern requires keeping references to the subscribed tasks one way or another, and keeping shared references implies that we cannot create mutable references anymore. This cripples our task tree, making it impossible to, for instance, transform a sequential task. While we would only request front-end updates through the observer pattern when it is safe to do so, convincing the compiler to accept it is hard without `unsafe` code. Wrapping all nested tasks in an `Arc<Mutex<T>>` is not an option

here, because it makes the API too convoluted.

Instead, we will go with a simpler approach. So far, we handled all changes to the front-end through the `Handler` trait. However, we cannot do that here yet, because if any task's `on_event` call changes a shared value, the tasks that handled the event earlier will already be outdated. Therefore, we will do a second pass over the executed task. We add an extra field to `Feedback` that keeps track of identifiers of `Shares` that were changed during the first `on_event` call. When the executor receives feedback, it first checks if the any shared data sources were updated. In that case, it transmits a new type of event to the executed task: `Event::UpdateShares(BTreeSet<Uuid>)`. If the task, or any of its subtasks, relies on a shared value with that identifier, it can send more feedback to update its representation. The executor will then combine the feedback and send it back to the client.

This implies that even view tasks now need a nontrivial `Handler` implementation:

```
impl<S> Handler for ViewValue<S>
where
    S: Send,
{
    async fn on_event(&mut self, event: Event) -> Feedback {
        match &event {
            Event::UpdateShares(ids) if self.share.updated(ids) => Feedback::Replace {
                id: self.id,
                html: self.to_html().await,
            },
            _ => Feedback::new(),
        }
    }
}
```

If the shared data source was updated, this will simply replace the current HTML representation with a regenerated version. The `ToHtml` implementation then retrieves the up-to-date value:

```
impl<S> ToHtml for ViewValue<S>
where
    S: Sync,
    S::Value: Display
{
    async fn to_html(&self) -> Html {
        Html(format!(
            r#"<div id="{self.id}">{self.value}</div>"#,
            self.id,
            *self.share.value()
        ))
    }
}
```

We can repeat these steps for `EditValue<T>` without running into problems. The next step would be to actually implement a shared data source that we can use in these tasks. The most basic share is simply a wrapper around a value:

```
#[derive(Clone)]
pub struct ShareValue<T> {
    id: Uuid,
    value: Arc<Mutex<TaskValue<T>>>,
}

impl<T> ShareValue<T> {
    pub fn new(value: Option<T>) -> Self {
        ShareValue {
            id: Uuid::new_v4(),
            value: Arc::new(Mutex::new(value.into_unstable())),
        }
    }
}

impl<T> Share for ShareValue<T> {
    type Value = T;

    fn value(&self) -> MutexGuard<TaskValue<Self::Value>> {
        self.value.lock()
    }
}

impl<T> ShareUpdate for ShareValue<T> {
    fn updated(&self, ids: &BTreeSet<Uuid>) -> bool {
        ids.contains(&self.id)
    }
}
```

Now we can fix the `View` and `Edit` implementations of `ViewValue` and `EditValue`. We changed their constructor to have a `Share` parameter, but the current implementations attempt to store values directly. Wrapping those values in a `ShareValue<T>` will solve this problem.

To actually take advantage of the shared values, however, we sometimes need to construct the tasks with an existing shared data source. Let us define special `view_shared` and `edit_shared` functions for that purpose, similar to `iTasks`' `viewSharedInformation` and `updateSharedInformation`:

```
pub trait ViewShared<S>: Sized {
    type Task: Value<Output = Self>;

    fn view_shared(share: S) -> Self::Task;
}

// Use the `ViewValue<S>` task for any share from which the value implements `Display`.
```

```

// Note that these generic implementations can cause conflicts when there are other
// implementations as well. Declarative macros can help when implementing traits for
// many types at once.
impl<S> ViewShared<S> for S::Value
where
    S: Share + Send + Sync,
    S::Value: Display + Clone + Sized + Send + Sync,
{
    type Task = ViewValue<S>;

    fn view_shared(share: S) -> Self::Task {
        ViewValue::new(share)
    }
}

pub fn view_shared<S>(share: S) -> <S::Value as ViewShared<S>>::Task
where
    S: Share,
    S::Value: ViewShared<S>,
{
    <S::Value as ViewShared<S>>::view_shared(share)
}

```

After implementing the `ViewShared<S>` and `EditShared<S>` traits for all types that work with `ViewValue<S>` and `EditValue<S>`, we can easily share values between tasks like so:

```

async fn greet() -> impl Task {
    let name = ShareValue::new("Alice".to_string());

    all! {
        edit_shared(name.clone()),
        view("Hello"),
        view_shared(name),
    }
}

```

Note that the call to `.clone()` is not inefficient in this case, because `ShareValue<T>` only contains an identifier and an `Arc`. Cloning an `Arc` does not clone the value it contains, it just increments an atomic counter and copies a pointer to the data. This task provides the user with a text field containing “Alice”, and whenever they change that value, the “Hello Alice” message below will automatically update to use the new value.

This is the most barebones implementation of shared data sources we found. It allows us to define more powerful tasks than before, but it is still quite limited. We decided to make two improvements that make this implementation easier to work with.

Allow other return types

While the basic implementation works well for the single share we defined so far, the `Share` trait can be difficult to work with for other types of shares. The `value` method has to return a `MutexGuard`, but not all shares store their value in a mutex. We could improve this by creating our own `ShareGuard` type. This could be an enum consisting of every type we want to return in our shares, with `From` implementations to convert those types to the right variant. However, because users of the library cannot add variants to the enum, this can still be limiting if they want to create new types of shares. For a more flexible solution, what matters is that the caller can turn the type returned by `value` into a mutable reference somehow. The standard library defines the `AsMut<T>` trait for this purpose:

```
pub struct ShareGuard<'a, T>(MutexGuard<'a, T>);

impl<'a, T> AsMut<T> for ShareGuard<'a, T> {
    fn as_mut(&mut self) -> &mut T {
        self.0.deref_mut()
    }
}

impl<'a, T> From<MutexGuard<'a, T>> for ShareGuard<'a, T> {
    fn from(guard: MutexGuard<'a, T>) -> Self {
        ShareGuard(guard)
    }
}

pub trait Share {
    type Value;
    type BorrowMut<'a>: AsMut<Self::Value>;

    fn value(&self) -> Self::BorrowMut;

    fn updated(&self, ids: &BTreeSet<Uuid>) -> bool;
}
```

This makes the `Share` trait more flexible, allowing users and us to define shared data sources that do not store their values in a mutex. The drawback of this approach is that it relies on generic associated types, as we have seen at the beginning of this chapter. We need to enable an unstable language feature to compile this code. However, unlike the `Value` trait where we first encountered this problem, the `Share` trait is never turned into a trait object, making this a viable alternative if unstable features are acceptable for the use case.

Separate reading and writing

Another interesting improvement is to split the `Share` trait into separate `ShareRead` and `ShareWrite` traits, similar to `iTasks`:

```
pub trait ShareRead {
    type Value;
    type Read<'a>: AsRef<Self::Value>;

    fn read(&self) -> Self::Read;
}

pub trait ShareWrite {
    type Value;

    fn write(&mut self, value: Self::Value);
}
```

Note that we replaced the `AsMut` trait with `AsRef` to indicate that the type should be turned into a shared reference, rather than a mutable reference.

Edit tasks generally require both read and write access to data, but view tasks can work with view access alone. Therefore, splitting the trait opens up opportunities for other interesting types of shared data sources. One example is a *mapped* share:

```
async fn greet() -> impl Task {
    let name = ShareValue::new("Alice".to_string());
    let greeting: Map<ShareValue<String>> = name.map(|value| format!("Hello, {}",
↪ value));

    all! {
        edit_shared(name),
        view_shared(greeting),
    }
}
```

The `map` method wraps the given share in a `Map` struct, which implements the `ShareRead` trait. When the user updates the value of `name`, `greeted_name` will recalculate its value using the provided closure. We can then view the resulting value because `ShareRead` is sufficient for view tasks. If we attempt to use it in edit tasks, however, the compiler will warn us that `ShareWrite` is not implemented for `Map<ShareValue<String>>`.

We could also create a `bimap` method with two closure parameters, which converts the value to another value and back. That way, the `Bimap` struct could implement both `ShareRead` and `ShareWrite`, allowing them to be used in edit tasks. However, such two-way conversions are not always practical. In the example above, it is not even possible to convert a string that does not start with “Hello” back to a name. Therefore, having an infallible, read-only alternative is useful if view tasks are sufficient.

This concludes the part on the more successful aspects of our shared data source implementation. While it is still quite minimal, the concepts described here can be used to implement a wide array of different types of shares, and enable us to define tasks

that are more powerful and responsive than before. For instance, a shared data source that represents a table row from a database is entirely possible to create with the ideas described here. However, because the implementation would be highly dependent on third-party libraries, and because a database is not strictly necessary for our demonstration in chapter 6, we will not elaborate on such a share here.

5.3.3 Supporting Shares for Advanced Tasks

The implementation of shared data sources described above is backwards compatible with the implementation described in chapter 4, allowing us to incrementally support shares for new types of tasks. One interesting, but particularly difficult example would be to support shared values for `EditVec<T>`. There are two reasons why this is a complex problem.

The main reason is that `EditVec<T>` uses subtasks to calculate its own value. So far, `EditVec<T>` does not own its value directly: each subtask owns the value of one element, and when the value of `EditVec<T>` is requested, it creates the vector on the fly. Adding a share to `EditVec<T>` implies that this share must derive its own value from the shares of subtasks automatically. When any of the subtasks changes its value, the resulting share should update automatically.

The second reason this is complex, is that `EditVec<T>` provides buttons that allow us to add or remove subtasks. This also changes its value. Adding an element should create a new subtask with a shared value, and add that shared value to the share of `EditVec<T>` itself. Analogously, removing an element should remove it from the share and the subtasks. Subtasks are still the main cause of difficulty here, but this is worsened by the fact that there is a variable number of them.

Currently, `EditVec<T>` is defined as follows:

```
pub struct EditVec<T> {
    container_id: Uuid,
    elements_id: Uuid,
    add_id: Uuid,
    rows: Vec<Row>,
    tasks: Vec<T>,
}

impl<T> EditVec<T> {
    pub fn new(tasks: Vec<T>) -> Self {
        EditVec {
            container_id: Uuid::new_v4(),
            elements_id: Uuid::new_v4(),
            add_id: Uuid::new_v4(),
            rows: tasks.iter().map(|_| Row::new()).collect(),
            tasks,
        }
    }
}
```

```
}
```

Notice how the type parameter `T` does not indicate the type of elements within the vector, but rather the type of task used to edit one of the elements. We cannot pick any specific task to edit each element, such as `EditValue<S>`, because `EditVec<T>` would not support every element type in that case. Editing a vector of vectors, for instance, would be impossible. Therefore, the new definition will keep track of both the element task `T` and the shared data source `S`:

```
pub struct EditVec<S, T> {
    container_id: Uuid,
    elements_id: Uuid,
    add_id: Uuid,
    rows: Vec<Row>,
    share: S,
    tasks: Vec<T>,
}

impl<S, T> EditVec<S, T>
where
    S: ShareChildren,
    S::Subshare: ShareRead<Value = T::Output> + Clone,
    T: Value,
    T::Output: EditShared<S::Subshare, Task = T>,
{
    pub fn new(share: S) -> Self {
        let tasks: Vec<T> = share.children().iter().cloned().map(edit_shared).collect();
        let rows = tasks.iter().map(|_| Row::new()).collect();
        EditVec {
            container_id: Uuid::new_v4(),
            elements_id: Uuid::new_v4(),
            add_id: Uuid::new_v4(),
            rows,
            share,
            tasks,
        }
    }
}
```

To construct an `EditVec<S, T>` from a shared data source `S`, we first retrieve its *subshares* through a new `ShareChildren` trait:

```
pub trait ShareChildren {
    type Subshare;

    fn children(&self) -> MutexGuard<Vec<Self::Subshare>>;
}
```

We then create subtasks for each of these shares, using the `EditShared<S::Subshare>`

trait. The next step would be to create the new type of share that represents this vector:

```
pub struct ShareVec<S> {
    shares: Arc<Mutex<Vec<S>>>,
}

impl<S> ShareVec<S>
where
    S: ShareWrite,
{
    pub fn new(value: Option<Vec<S::Value>>) -> Self {
        ShareVec::create(value.into_unstable())
    }
}
```

Retrieving the subshares from this type is straightforward:

```
impl<S> ShareChildren for ShareVec<S> {
    type Subshare = S;

    fn children(&self) -> MutexGuard<Vec<Self::Subshare>> {
        self.shares.lock().unwrap()
    }
}
```

Reading values from this share is also quite simple, although inefficient:

```
// Newtype to implement `AsRef<Vec<T>>` for
pub struct VecWrapper<T>(TaskValue<Vec<T>>);

impl<T> AsRef<TaskValue<Vec<T>>> for VecWrapper<T> {
    fn as_ref(&self) -> &TaskValue<Vec<T>> {
        &self.0
    }
}

impl<S> ShareRead for ShareVec<S>
where
    S: ShareRead,
    S::Value: Clone,
{
    type Value = Vec<S::Value>;
    type Read<'a> = VecWrapper<S::Value> where S: 'a;

    fn read<'a>(&'a self) -> Self::Read<'a> {
        let vec = self
            .shares
            .lock()
            .unwrap()
            .iter()
```

```

        .map(|share| share.read().as_ref().clone())
        .collect();
    VecWrapper(vec)
}
}

```

This `read` method clones the values of each of the subshares, only to have them used as a reference later. This is necessary because the associated type `Read` requires that the return type mimics `&Vec<T>`, rather than `Vec<&T>`.

We elaborated on a similar problem at the beginning of this chapter, where the `value` method generally cloned the value of primitive tasks. Returning references instead did not solve the problem, because nested tasks would then need to clone the values of subtasks in order to create a reference to the right type. We were able to solve the problem using generic associated types, but unfortunately that does not apply here. The return type of the `value` method is only used in sequential tasks, where the user supplies functions for the condition and transformation of the task. Because the user handles the value manually, we need no further assumptions for the type. That is not the case here, because the return type of `read` is used by many kinds of tasks and other shared data sources. Therefore, we need the `AsRef` trait bound, or something similar, to turn the type into something we can work with generically. Unfortunately, that trait bound cannot generalize over references, i.e. support both `AsRef<Vec<T>>` and `AsRef<Vec<&T>>`, which makes cloning the only viable option here.

Now we need to implement `ShareWrite` for this share. When we receive the new vector value, we first need to determine how we should assign the new element values to the subshares. Multiple elements might have been added, changed, moved, or deleted in the new vector. We could use a diff algorithm to determine exactly where to insert or remove shares, or change an existing share's value. Doing this properly would also allow the front-end to animate changes to the vector, making it easier for the user to see what changed. However, for the purpose of this explanation, we can also simply delete all current shares and insert new shares with the updated values. Both options require inserting and removing shared data sources regardless:

```

impl<S> ShareWrite for ShareVec<S>
where
    S: ShareWrite + ShareConstruct,
{
    type Value = Vec<S::Value>;

    fn write(&mut self, value: TaskValue<Self::Value>) {
        self.shares = match value {
            TaskValue::Stable(value) => value
                .into_iter()
                .map(|value| S::construct(TaskValue::Stable(value)))
                .collect(),
            TaskValue::Unstable(value) => value
        }
    }
}

```

```

        .into_iter()
        .map(|value| S::construct(TaskValue::Unstable(value)))
        .collect(),
    TaskValue::Error(error) => vec![S::construct(TaskValue::Error(error))],
    TaskValue::Empty => vec![S::construct(TaskValue::Empty)],
};
}
}

```

Note that we added a new trait bound to `S: ShareConstruct`. It contains the `construct` method to create new instances of the shared data source, which is necessary when new elements are inserted into the vector. `construct` resolves to a call to `new` for both shares that we implemented so far.

Finally, let us implement `ShareUpdate` for this shared data source:

```

impl<S> ShareUpdate for ShareVec<S>
where
    S: ShareUpdate,
{
    fn updated(&self, _ids: &BTreeSet<Uuid>) -> bool {
        true
    }
}

```

While far from elegant, this implementation is sufficient. It indicates that tasks depending on this share should always update themselves to reflect a possibly new value. Ideally, we would make this more efficient by determining whether the value has actually been changed. However, this is more difficult than it seems. We could use `self.shares.iter().any(|share| share.updated(ids))` to see if any of the sub-shares have been updated, but this will not detect new or removed shares, or reordered current shares. The latter requires keeping extra state, such as a timestamp.

With the shared data source completed, the `Handler` implementation of `EditVec<S, T>` requires some adaptations:

```

impl<S, T> Handler for EditVec<S, T>
where
    S: ShareChildren + Send,
    S::Subshare: ShareWrite,
    T: Value + Handler + Send + Sync,
    T::Output: Clone,
{
    async fn on_event(&mut self, event: Event) -> Feedback {
        match event {
            Event::Press { id } if id == self.add_id => {
                // Add a new row
                self.share
                    .children()
            }
        }
    }
}

```

```

        .push(<S::Subshare as ShareWrite>::create(TaskValue::Empty));
    Feedback::new()
}
Event::Press { id } if self.rows.iter().any(|row| row.remove_id == id) => {
    // Remove an existing row
    let index = self
        .rows
        .iter()
        .position(|row| row.remove_id == id)
        .unwrap();
    self.share.children().remove(index);
    Feedback::new()
}
Event::UpdateShares(ids) => if self.share.updated(&ids) {
    self.tasks = self
        .share
        .children()
        .iter()
        .cloned()
        .map(edit_shared)
        .collect();
    self.rows = self.tasks.iter().map(|_| Row::new()).collect();
    Feedback::from(Change::Replace {
        id: self.container_id,
        html: self.to_html().await,
    })
} else {
    Feedback::new()
}
_ => future::join_all(
    self.tasks
        .iter_mut()
        .map(|task| task.on_event(event.clone())),
)
    .await
    .into_iter()
    .collect(),
}
}
}

```

Pressing buttons no longer updates the layout of the task. Instead, it updates its front-end when the share changes value. Then, it recreates its subtasks with the right values, and regenerates its HTML representation.

Both `ShareVec<S>` and `EditVec<S, T>` are now complete. We can now create tasks such as the following:

```

async fn vecs() -> impl Task {
    let vec: ShareVec<ShareValue<String>> = ShareVec::new(Some(Vec::new()));

    all! {

```

```

        edit_shared(vec.clone()),
        edit_shared(vec),
    }
}

```

This creates a page with two tasks to edit a vector. Whenever the user interacts with one of them, the other is updated automatically. The values of the vectors remain synchronized with each other.

This result could be interpreted as a victory of sorts. It further increases the flexibility of the library. Moreover, the details of this implementation can be applied to other types of tasks as well, such as those for optionals or tuples, which also rely on subtasks. Those have a fixed number of subtasks, making it is easier to determine whether their values were updated.

However, these shares come with a major limitation: they depend on the `ShareChildren` trait. To illustrate why this is a problem, suppose we define a new share-less task `EditSimpleVec<T>`, that edits vectors without relying on subtasks. It does this by storing a `Vec<T>` directly, rather than a vector of subtasks that produce `T` values. When we add support for shared data sources for this task, we need to replace its `Vec<T>` with a type that implements `ShareRead<Value = Vec<T>>` and `ShareWrite<Value = Vec<T>>`.

The `ShareVec<S>` we defined above fulfills this requirement – provided that the value of `S` is `T`. However, `ShareValue<Vec<T>>` is also sufficient. The difference between these two is that the former implements `ShareChildren`, is much less efficient, and much more difficult to implement. Since `ShareSimpleVec<T>` has no use for subshares, because it has no subtasks, it should use the latter.

Having two types of shares for practically the same purpose is confusing, especially because the simpler variant is the right choice in this case. Recall that we defined a `view` task to choose a fitting task out of `ViewValue<T>`, `ViewVec<T>`, among others. `ViewValue<T>` could also theoretically support vectors if we implement `ToHtml` for them, but `ViewVec<T>` provides more display options and is more or less as efficient, making it the better choice. We repeated this pattern with `edit` for `EditValue<T>` and `EditVec<T>`. Since we have `ShareValue<T>` and `ShareVec<S>` now, it would make sense to define a `share` function as well. Unfortunately, if we choose `ShareVec<S>` as the default share for vectors, `ShareSimpleVec<T>` would be inefficient. If we choose `ShareValue<Vec<T>>` instead, the `EditVec<S, T>` defined above would not work, because the share does not implement `ShareChildren`.

To summarize, while feature-complete shares are possible to implement in Rust, they do not interoperate well with tasks that use subtasks to compute their value. Those require specialized types of shares that are less efficient and more difficult to implement. `ShareValue<T>` can be used for any task without subshares, but a `share` function that picks a default share *must* pick the ones that implement `ShareChildren` in order to be

usable for any task. This leads to a confusing API where the user must choose between `ShareValue::new(Some(value))` and `share(value)`.

Chapter 6

Demonstration

To demonstrate that the library is suitable for real-world problems, we will implement an example program that demonstrates the concepts described in this thesis. The program will be a simple group chat application. Users first enter their own name. After that, they can see all messages that have been sent by other users, and send new messages themselves.

The implementation relies on a persistent shared data source that contains a list of all sent messages. All users can read messages from the share concurrently, and when they submit a new message, the front-end of all other users will automatically update to display it.

6.1 Implementation

The first part of the program is simple, have the user enter their name:

```
async fn index() -> impl Task {
  view("Please enter your name: ")
    .right(enter::<String>())
    .step()
    .on(
      Trigger::Button(Button::new("Ok")),
      has_value,
      |name| chat(name.unwrap())
    )
}
```

We have seen this pattern before. It provides a text input for the user to enter their name. Afterwards, they can press the ‘Ok’ button to move on to the followup task `chat`, which we will define later.

We first create a persistent share to hold all sent messages. There are multiple ways to

achieve persistence. The best option would be to use the idiomatic procedure of sharing state for the web framework that we use. This would be most intuitive for its users. For `axum`, this would mean using the `FromRequest` trait to automatically retrieve the persistent messages as a parameter in the task handler. However, we will define a global variable here instead, as this approach will work in any context.

Usually, defining a global variable looks as follows:

```
static MESSAGES: Arc<Mutex<String>> = Arc::new(Mutex::new(String::new()));
```

This will calculate the initial value at compile time, and include it in the binary. Doing this requires that the value is `const`, which implies that it is constructed only from other constants and `const` functions. Constant functions are pure: their output is solely dependent on their inputs. However, this functionality is still quite limited in Rust. Many pure functions are not `const` due to compiler limitations. One such limitation states that constants cannot have allocations, which rules out `Vec<T>`. This is why we use the `lazy_static!` macro to define our global variable instead. It does not calculate the value at compile time, but rather the first time it is used:

```
lazy_static! {
    static ref MESSAGES: ShareVec<ShareValue<Message>> = share(Vec::new());
}

#[derive(View)]
pub struct Message {
    sender: String,
    content: String,
}

impl Display for Message {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}: {}", self.sender, self.content)
    }
}
```

Now we can implement the chat task itself:

```
fn chat(name: String) -> impl Task {
    view_shared(MESSAGES.clone())
    .right(enter::<String>())
    .step()
    .on(
        Trigger::Button(Button::new("Send")),
        has_value,
        move |content| {
            let message = Message {
                sender: name.clone(),
                content: content.unwrap()
            }
        }
    )
}
```

```

    };
    let mut current = MESSAGES.read().as_ref().clone().unwrap();
    current.push(message);
    MESSAGES.write(TaskValue::Unstable(messages_value));
    chat(name)
  },
)
}

```

This task first displays the list of messages to the user. At the bottom of the list is a text field in which the user can enter a new message. These tasks are combined using the `right` combinator, which ignores the value of the list and keeps the message that the user entered. When the user presses the ‘Send’ button, the continuation will first append this message, along with the name of the sender, to the global message share. After that, it will transition back to the `chat` task, which will update the list of messages and allow the user to send a new message.

This example reveals that a `ShareModify` trait would be a useful addition to the API. First reading all messages, then inserting one new message, and finally writing those messages back to the share is a bit verbose. It is also less efficient than inserting the new message directly, because we would not need to clone the vector in that case. Most importantly, it resolves a race condition in this code: read and write both lock the inner mutex, but it is freed for a short while in between. If another user acquires the mutex in that time, the clone of the messages would not contain the new message of the other user yet. This will result in either of their messages being overwritten. Luckily, `ShareModify` is simple to implement: it is the same as the `Share` trait we used before we split it into `ShareRead` and `ShareWrite`. This allows us to simplify the transformation function:

```

move |content| {
  let message = Message {
    sender: name.clone(),
    content: content.unwrap()
  };
  MESSAGES
    .modify()
    .as_mut()
    .as_mut()
    .map(|vec| vec.push(message));
  chat(name)
}

```

The first call to `as_mut` is used to retrieve the `&mut TaskValue<Vec<Message>>` of the share, the second `as_mut` is to turn that into a `TaskValue<&mut Vec<Message>>`. Then we can insert the new message in the vector directly.

We believe this project demonstrates the power of TOP well. It declaratively specifies the behavior of a nontrivial, multi-user system, using relatively few lines of code. Un-

fortunately, this code does not compile just yet, due to a usability issue that we did not foresee. Notice that this task is recursive: its only continuation creates another `task(name)` as followup. This is completely valid, and returning to a previous task is a common pattern. However, the type checker will complain about this definition.

The problem lies in the return type: `impl Task`. Using `impl Trait` as return type indicates that the compiler should infer the concrete type returned by the function. In this case, that will result in a cycle:

```
error[E0391]: cycle detected when computing type of `chat::{opaque#0}`
  --> examples/axum/src/main.rs:26:26
   |
26 | fn chat(name: String) -> impl Task {
   |                               ^^^^^^
note: ...which requires borrow-checking `chat`...
  --> examples/axum/src/main.rs:26:1
   |
26 | fn chat(name: String) -> impl Task {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
note: ...which requires type-checking `chat`...
  --> examples/axum/src/main.rs:30:10
   |
30 |         .on(
   |         ^^
   = note: ...which requires evaluating trait selection obligation `impl`
   ↪ top::task::Task: core::marker::Sync...
   = note: ...which again requires computing type of `chat::{opaque#0}`, completing the
   ↪ cycle
```

It is impossible to infer the concrete return type of `chat` if it cannot verify the return type of the transformation function. It can see that the type implements `Task`, which is enough to support this use case, but Rust has to determine the concrete return type at compile time to generate code.

To circumvent this limitation, we have to specify the concrete type ourselves, which is extremely verbose:

```
type ChatTask = Sequential<
    Parallel<
        ViewVec<ShareVec<ShareValue<Message>>, ViewValue<ShareValue<Message>>>,
        EditValue<ShareValue<String>>>,
        fn(TaskValue<Vec<Message>>, TaskValue<String>) -> TaskValue<String>,
    >,
    Infallible,
>;

fn chat(name: String) -> ChatTask { /* ... */ }
```

Let us break this type down. The outermost type of `chat` is a sequential task. The `Sequential` task has two type parameters: one signifies the type of its starting task, the other the type of the intermediate value of the followup task:

- The starting task is a parallel task. `Parallel` has three type parameters: the type of the first task, the type of the second task, and the type of the function that combines the values of these tasks into one value:
 - The first task shows the shared vector of messages. `ViewVec`, like `EditVec`, has two type parameters: the type of share it uses, and the type of task used to view its elements:
 - * The share used is the infamous `ShareVec<ShareValue<T>>` from the last section of chapter 5.
 - * To view `ShareValue<Message>` types, we use a simple `ViewValue<S>` task.
 - The second task represents the text input where the user can enter their messages. It is the default `EditValue<ShareValue<String>>` in this case.
 - These functions are combined with the `right` combinator, which ignores the value of the first task (`Vec<Message>`), and returns the value of the second (`String`).
- The return type of the sequential task is never constructed: the task always creates followup tasks that the user must complete first. As a result, we can choose any type here, and the type checker will accept it. We chose `Infallible` here, because it is an enum without variants, and therefore not possible to construct. This indicates that we should not expect output from this task.

This return type is huge, even though the task definition is relatively small. Other recursive tasks will likely require much larger return types. Fortunately, we do not have to write the return type ourselves. If we specify `()` as return type, the compiler will tell us exactly what the type of the task is:

```
note: expected unit type ()
      found struct Sequential<Parallel<ViewVec<ShareVec<ShareValue<Message>>,
↳ ViewDisplay<ShareValue<Message>>>, EditValue<ShareValue<String>>,
↳ fn(TaskValue<Vec<Message>>, TaskValue<String>) -> TaskValue<String>>, _>
```

We only have to replace any underscores with `Infallible`. Regardless, this is an obnoxious limitation, especially because the output above proves that the compiler *can* infer the type. Using `impl Task<Output = Infallible>` as return type would fill in the blanks. However, it would need to infer this type before type-checking the transformation function, but the parameters are evaluated first.

6.2 Output

With some additional styling, the front-end generated from the demonstration program can look like figures 6.1 and 6.2.



Please enter your name:

Figure 6.1: The `index` task that asks the user for their name.



Alice: Hello
Bob: World

Figure 6.2: The `chat` task. Alice has already sent “Hello” previously, while Bob transmitted “World”.

As a small disclaimer: we did not perform input validation of any kind. Users can transmit raw HTML or JavaScript code as message, making this site vulnerable to cross-site scripting attacks. It should not be used in production in its current state.

Chapter 7

Conclusion

During this research, we have implemented a working example of a task-oriented programming library in Rust. In this section, we will summarize our findings, and answer our research question: *What benefits and drawbacks does the Rust programming language bring to Task-Oriented Programming?*

Unfortunately, we were not able to find many benefits of Rust when comparing our implementation to iTasks. In theory, Rust should have lower-level control over memory, which could allow it to be more performant. However, our implementation makes insufficient use of this control. The library clones the intermediate values of all tasks every time they are retrieved, even when a reference would be sufficient. Returning references instead would force tasks with subtasks to clone those values when calculating their own value. While we found a work-around involving generic associated types, that solution will not work until traits with generic associated types become object-safe, which will likely take multiple years. Moreover, the work-around does not work for shared data sources that use subshares. Those require a trait bound that allows other parts of the library to use them generically, whereas the the values of tasks are handled by the user directly.

It is perhaps unfair to compare our implementation with a framework as mature as iTasks. Instead, let us focus on how well Rust supports the principles of TOP in general, which is overall positive. The object-based definition of tasks works quite well. It supports primitive tasks as well as composed tasks, and the combinators used to construct them. Tasks, and their intermediate values, remain strongly typed across the entire library. View and edit tasks also work with many different types. The library provides tasks to interact with types from the standard library, while derive macros and conversion tasks can handle most other types. Shared data sources are fully functional, although a bit confusing. Finally, while the API is a bit more verbose, it is not difficult to use, and conceptually not that different from iTasks.

In short, the library supports all of the concepts of task-oriented programming. Its main

drawbacks arise from practical usability issues:

- Types from other libraries likely do not implement `View` or `Edit`. We also cannot derive these traits for them, as derive macros need to be put directly above the type definition. The workaround using conversion tasks makes this problem manageable. However, `EditFromStr` only works for types that implement both `Display` and `FromStr`, and only if they are each other's inverse. Moreover, having just one text field for these types might not be user-friendly. The alternatives `EditFrom` and `ViewInto` solve these problems, at the cost of being verbose. They are impractical for projects that use many external types.
- The library lacks a viable option to support multiple layouts for interaction tasks. For example, the programmer can choose to use a different edit task than the default, but it is not possible to derive multiple different edit tasks for the same type. If another task for that type does not already exist, they would need to create it themselves. While, they can use the `EditFrom` workaround, it is still rather verbose.
- Shared data sources for tasks with subtasks need to implement `ShareChildren`. They are also significantly more work to implement, and rather inefficient. To make matters worse, they make it ambiguous which type of shares should be used. A helper `share` function cannot determine the best candidate, and would have to fall back on the inefficient share that implements `ShareChildren` to support all use cases.
- Recursive tasks require the programmer to specify the concrete return type, which is, again, verbose.

Aside from verbosity, Rust is a viable candidate for task-oriented programming. Combined with its recent popularity, it may help bring more attention to the paradigm and its research.

7.1 Future Work

Many of the problems with our implementation seem to arise from the existence of tasks with subtasks. Without subtasks, all tasks could simply return a reference to their value, without them or a supertask having to clone it. Eliminating subtasks would also allow us to remove the `ShareChildren` trait, making shares much easier to implement and use. However, our library uses subtasks to prevent double work in interaction tasks. For instance, `EditVec` reuses another edit task to render each of the elements. Future research should show whether Rust can support a solution with the best of both worlds.

One possible direction to search is to split view and edit tasks into two parts: one containing the intermediate value of the task, and one *editor*, which takes care of the visual representation of that value. It is important that the editor does not have ownership over (part of) the value, as that would lead to the same problem as subtasks. Instead,

they should operate on references of the value, and can forward those references freely to subeditors. This adaptation would make the library more similar to iTasks. It could also eliminate the need for multiple different view and edit tasks.

References

- Gjengset, J. (2021). *Rust for rustaceans: Idiomatic programming for experienced developers*. USA: No Starch Press.
- Klabnik, S., & Nichols, C. (2018). *The rust programming language*. USA: No Starch Press.
- Koopman, P., Plasmeijer, R., van Eekelen, M., & Smetsers, S. (2001). *Functional programming in clean*. Setem.
- Lijnse, B. (2022). *Toppyt*. <https://gitlab.com/baslijnse/toppyt>.
- Plasmeijer, R., Achten, P., & Koopman, P. (2007). itasks: executable specifications of interactive work flow systems for the web. *ACM SIGPLAN Notices*, 42(9), 141–152.
- Plasmeijer, R., Achten, P., Lijnse, B., & Michels, S. (2011). Defining multi-user web applications with itasks. In *Central european functional programming school* (pp. 46–92).
- van Gemert, D. (2022). *Task oriented programming in lua*. https://www.cs.ru.nl/bachelors-theses/2022/Dante_van_Gemert___1032684___Task_Oriented_Programming_in_Lua.pdf.