José Miguel Gonçalves Duarte

Bachelor of Science in Computer Science and Engineering

Bridging the Gap between Typestates and Rust in Production Software

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: António Ravara, Associate Professor, NOVA School of Science and Technology

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2021

Abstract

As software becomes more prevalent in our lives, bugs are able to cause significant disruption. Thus, preventing them becomes a priority when trying to develop dependable systems. While reducing their occurrence possibility to zero is infeasible, existing approaches are able to eliminate certain subsets of bugs.

Rust is a systems programming language that addresses memory-related bugs by design, eliminating bugs like *use-after-free*. To achieve this, Rust leverages the type system along with information about object lifetimes, allowing the compiler to keep track of objects throughout the program and checking for memory misusage. While preventing memory-related bugs goes a long way in software security, other categories of bugs remain in Rust. One of which would be API misusage, where the developer does not respect constraints put in place by an API, thus resulting in the program crashing.

Typestates elevate state to the type level, allowing for the enforcement of API constraints at compile-time. Relieving the developer from the burden that is keeping track of the possible computation states at runtime, preventing possible API misusage during development. While Rust does not support typestates by design, the type system is powerful enough to express and validate typestates.

This thesis' goal is to bridge the gap between typestates and production Rust. Developing a practical tool enabling developers to take advantage of typestates in their Rust code. The tool also provides extra correctness properties, by checking the typestate specification for common pitfalls.

Keywords: Behavioral types, typestates, meta-programming, macros, Rust

Resumo

Conforme as nossas vidas são cada vez mais dependentes de software, erros do mesmo têm o potencial de causar problemas significativos. Prevenir estes erros torna-se uma tarefa prioritária durante o desenvolvimento de sistemas confiáveis. Erradicar erros por completo é impossível, no entanto é possível eliminar certos conjuntos.

Rust é uma linguagem de programação de sistemas que, por desenho, endereça erros de gestão de memória. Para o conseguir, a linguagem inclui no sistema de tipos informação sobre o tempo de vida dos objetos, permitindo assim que o compilador consiga manter uma vista sobre a utilização dos mesmos e verificar por erros de utilização de memória. Apesar da prevenção de erros de memória ter um papel importante na segurança de software, existem ainda outras categorias de erros em Rust. Como o uso incorrecto de interfaces de programação, em que o programador não respeita as restrições impostas pela mesma, resultando numa falha do programa.

Typestates elevam o conceito de estado para o sistema de tipos, permitindo a aplicação das restrições da interface durante a fase de compilação. Este conceito permite assim aliviar o programador da responsabilidade que é conceptualizar e manter o estado do programa em mente durante o desenvolvimento, prevenindo o mau uso das interfaces. Apesar de Rust não suportar *typestates* de uma forma natural, o sistema de tipos permite expressar e validar *typestates*.

O objetivo desta tese é aproximar os *typestates* do Rust em produção, desenvolvendo uma ferramenta que permite aos programadores tirar partido dos *typestates*. A mesma ferramenta é também capaz de garantir propriedades extra, verificando a especificação de *typestates* por erros comuns.

Palavras-chave: Tipos comportamentais, *typestates*, meta-programação, macros, Rust

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

Bugs permeate our lives as users — whether in an instant messaging application or a game they are present. Luckily, most are harmless as such applications are not critical, resulting in some unsent messages or texture glitches.

In systems programming, one of the most demanding domains in computer science, bugs and their respective consequences come at a high cost to both service providers and consumers. There are reports from several industries where bugs lead to huge monetary losses and even death. In 2014, the Heartbleed [25] bug, caused due to a missing bound check, compromised the security of any OpenSSL user, enabling the theft of critical information (e.g. cryptographic keys). In 2018, a bug in Coinbase (a popular cryptocurrency exchange) allowed for account balance manipulation [75]. In 2019 and 2020, after several crashes [5], the Boeing 737 Max was grounded to fix existing problems. While grounded, more software-related issues were found [50, 49], delaying re-certification. In 2020, as the number of COVID-19 cases grew, contact tracing apps were deployed as a mitigation strategy — due to a bug the UK's National Health Service app failed to ask users to self-isolate [43].

The previous examples are not isolated incidents. The language and nature of the bugs are different for each case, but to put it simply, there is no silver bullet and the next best alternative is to do our best to mitigate them — building tools and abstractions which allow developers to increase their code's safety.

```
1 Integer a = null;
2 a + 5; // NullPointerException: `a` is `null'
```

Listing 1.1 – Java's null reference example.

1.2 Problem

Languages like C/C++ have dominated the systems programming landscape for years and one of the main problems with both is the lack of memory management. Leaving such responsibility to the developer has proven to be *a less than ideal* solution. Memory management is responsible for 70% of the bugs found in projects like Chromium [44] and Microsoft products [45].

To address such problem, several tools and languages have been and continue to be developed — so far, Rust has been the only one to achieve *mainstream* status. Rust aims to provide memory safety without affecting performance or productivity. To achieve such ambitious goal, Rust validates code with the borrow checker, which then enforces memory safety rules, targeting the problem at the root.

Addressing memory safety is not enough though. Languages which side-step the problem of having manual management through the use of a garbage collector (e.g. Java and Go) still suffer from other kinds of bugs. As discussed in the end of Section 1.1, we can only mitigate their occurrence, hence we are required to reach out to new mechanisms.

Typestates are an approach which aims to tame stateful computations; to do so typestates lift the concept of state to the type level, this enables the compiler to reason about state and provides the developer with information of the expected computation state at runtime.

1.2.1 The Billion Dollar Mistake

> *This led me to suggest that the null value is a member of every type, and a null check is required on every use of that reference variable, and it may be perhaps a billion dollar mistake. — [27]*

Consider Listing 1.1, the program compiles and will crash with a `NullPointerException`. While every one can see the explicit `null` attribution the compiler does not issue an error or warning. The original author of the `null`, Tony Hoare, considers this to be his "*billion dollar mistake*". Since in complex codebases, this error is hard to track down among all possible states and has supposedly caused more than a billion dollars in damages.

While in Java it manifests as an exception, in C/C++ tracking them down is usually more complicated as the only feedback the user receives is the infamous `SEGFAULT`. Again, after so many years of programming, developers ought to have better tools, as debugging errors like these is neither an effective time use nor pleasant.

```
1  Scanner s = new Scanner(System.in);          // open the stream
2  s.nextLine();                                // read
3  s.close();                                   // close the stream
4  s.nextLine();                                // IllegalStateException
```

Listing 1.2 – Java's `Scanner` misuse example.

```
1  Scanner[Open] s = new Scanner(System.in);    // open the stream
2  s.nextLine();                                // read
3  Scanner[Closed] s = s.close();               // close the stream
4  s.nextLine();                                // compile-time error
```

Listing 1.3 – Typestated `Scanner` example. Notice how the compiler is able to detect the error.

1.2.2 API Misuse

Consider Java's `Scanner`, the API allows the developer to write code like Listing 1.2. Such code will compile without issuing any errors or warnings (even with the `-Xlint:all` flag), however, it will also crash during runtime. Since it is not possible to read from a closed source, the thrown exception is an `IllegalStateException`, informing the user that the attempted operation is illegal for the current object state. Ideally we want such illegal states to be detected at compile time.

As shown in Listing 1.3, using a *typestated* Java example, the code allows us to trace the state of the object, but even better, the compiler is now able to tell us there is an error during compilation. This approach also solves Listing 1.1, as the type is required to be explicitly *nullable*. The remaining question is:

> *How can we avoid API misusages without creating a new full-fledged programming language?*

1.3 State of the Art

Behavioral types are types which capture aspects of computation, they are further discussed in Section 2.3. The current landscape of behavioral types in mainstream languages is bare. While projects exist, most are academic and of little impact in the way programmers write their code. In this document I focus on two approached to behavioral types — session types and typestates.

Session types will often refer to endpoints and their messages, capturing aspects of communication between them. Languages like ATS provide session type features and further enable the generation of source code in other languages such as Erlang [81]. ATS also serves as research playground for other topic related with session types [80]. There are also tools that plug into existing languages. These may come under the form of libraries

such as session types for Haskell [2, Section 3.3], [22, Chapter 10] and OCaml [22, Chapter 11] extending the language to provide session types through existing mechanisms. In OCaml's case, this is done without reaching for external tools or language extensions, relying purely on the existing type system. However, existing session types research is not only based on functional languages. Session C [2, Section 4.1] makes use of Scribble [82] to capture the communication pattern of the algorithm. The tool then generates the required endpoints that guide the design and implementation of the program. Java has been the target several other research efforts, for example SessionJ [2, Section 2.2.1], [30], a Java extension which is an implementation of Moose [2, Section 2.1.1]. Another session type enabling project is Mungo & StMungo [38, 76], also targeting Java. They define specification languages which check that the code complies with the required properties. StMungo converts Scribble protocols to Java classes with typestates which are then checked by Mungo, this enables developers to write effectively session-typed Java. By itself, Mungo is a typechecker with support for typestates.

Typestates capture the state of the program, allowing the developer to express the state of objects during runtime, at compile-time. I discuss typestates further in Section 2.3.2. Fugue [12] is a protocol checker that achieves similar functionality to typestates. The tool provides a series of annotations to be used in code which are then processed into protocols to be checked by Fugue. Using the tool, the authors found several errors which would inhibit application scaling in a Microsoft internal project. Languages like Plaid [1] and Obsidian [7, 8] put typestates to use, trying to bridge the gap between academia and industrial usage. Plaid is an object-oriented language with first class support for typestates. Obsidian is a relatively new language which targets the Hyperledger Fabric blockchain. The language aims to make writing smart contracts simpler and less error prone through the addition of linear types and typestate mechanisms to the language. In [7] the effectiveness of the approach is put to test, achieving positive results when compared with the Solidity programming language.

The `state_machine_future` crate [20], provides typestated futures in Rust as well as some state machine related guarantees, such as every state being reachable from the start, there are no states unable to reach the final state and that all state transitions are valid. Furthermore, these guarantees are provided at compile-time — for example, invalid state transitions will fail to compile. The crate, however, revolves around futures, requiring an asynchronous runtime and thus making it unsuitable for other kinds of applications. Other crates exist, they focus on finite state machines but are unable to provide static guarantees.

Like other mainstream programming languages, Rust does not have first class support for session types. Implementations are rare and rooted in the meta-programming system. The work done by [34, 48] introduces bi-directional session types to Rust, since then, this line of work has been expanded by [41], extending it to multiparty session types. While Rust dropped typestate support during early development (Rust 0.4), that does not

mean Rust is not able to provide them. The type system is able to emulate typestates with efficiency, the approach however comes at the cost of verbosity. Regardless of the verbosity typestates are used by the embedded systems development sphere of the Rust community.

1.4 Objectives & Contributions

In this thesis I try to bridge the gap between typestates and Rust, aiming for an elegant and usable solution, allowing for effective usage of typestates in Rust. To achieve such solution I expect to develop an embedded Rust DSL, enabling the flexibility of a dedicated language inside the Rust ecosystem. To this effect I expect the contributions of this thesis to be a typestate specification DSL to be embedded in Rust, this topic is further developed in Chapter 4.

Typestate DSL. One of the main goals of the DSL is to be non-intrusive and easy to pick up — both the syntax and tooling. Such requires the syntax to extend over Rust's current syntax, introducing minimal changes where necessary. However, it should also be powerful enough to specify useful protocols in it.

Static Guarantees. As any language, it is useless if no information is extracted from it, besides the obvious parsing step, the DSL should be able to extract a typestate model from the original specification and generate adequate output code. The extracted model should also be checked for a series of properties such as state reachability and termination.

Tooling & Usability. The DSL should not require more than the import of the library, building any project using the DSL should not require extra steps as it would degrade possible adoption. A survey should accompany the final product to confirm usability claims.

Artifacts. Finally, the DSL should be shipped as a crate (i.e. library) and available in `crates.io`, Rust's package registry, this implies that the documentation should be available in `docs.rs`. In addition to the DSL library, I am planning writing an article on the DSL, including the results from usability testing, and developing a library to facilitate DSL development for Rust.

1.5 Report Organization

This document is organized as follows:

Chapter 2 provides a review over existing systems programming languages (Section 2.1), the Rust programming language (Section 2.2) and behavioral types (Section 2.3).

Chapter 3 describes existing work regarding language preprocessing (Section 3.1), Rust macros (Section 3.2) and existing approaches to behavioral types (Section 3.3).

Chapter 4 illustrates the development roadmap of this project, detailing the required work to achieve the goals proposed in Section 1.4.

Chapter 2

Background

2.1 Systems Programming Languages

The definition of the term *systems programming language* is not agreed upon, being somewhat flexible and ever-changing due to constant shift in requirements for applications.

Before the cloud, in the age of C, a systems programming language would most likely be a language able to provide an adequate interface between the programmer and the machine. Nowadays, the definition is more vague, as machines and software grow in complexity, and the definition of system grows from single computer to a distributed system, interfacing with the hardware in a more direct fashion is mostly not required. Systems programming languages become about being able to produce a standalone binary able to run on a variety of machines without requiring extra software.

2.1.1 C

C is a general-purpose programming language, while it can be considered a high-level programming language when put besides assembly, it also fits the description of a low-level level programming language when besides languages like Python. It was originally designed by Dennis Ritchie for the PDP-11 and has been around since 1972 [4], C is by no means modern, being older than myself and most likely to outlive me.

Designed in a different time, C's mental model is also different, the language is simple and straight forward, the designers had goals to achieve and designed the language with them in mind. Such mentality is noticeable when using the language, it is simple as the hardware was and the level of control C provides is unparalleled, being both a major benefit and a hindrance. An expert programmer is able to take advantage of the language to produce highly-efficient software, but a novice programmer will often find himself battling memory and pointer management bugs.

The language influence echoes in the modern languages, whether in the form of syntax (i.e. the famous C-style syntax) or in the problems it tries to solve. Languages such as Java take from C their syntax as well as one problem to solve, memory management; other languages like Julia [3] aim to achieve similar performance.

While not as popular as other languages, C was able to stay relevant in the modern development landscape, some of the most used software in the world is either written or powered by C. The Linux kernel, which powers servers, the world's most powerful computers and serves as a base for Android and other mobile devices, git, Redis and nginx are also software examples which reached the top of their respective fields.

2.1.2 C++

Introduced in 1985 as an extension to C; the author, Bjarne Stroustrup writes:

> *C++ is based on the idea of providing both:*
>
> - *direct mappings of built-in operations and types to hardware to provide efficient memory use and efficient low-level operations, and*
> - *affordable and flexible abstraction mechanisms to provide user-defined types with the same notational support, range of uses, and performance as built-in types.*
>
> — *[65, Section 1.2]*

The language has since gone on to conquer the programming world, being used in a wide variety of software and hardware. Currently, companies such as Google, Amazon and Microsoft have widespread adoption of C++ in their codebases. Industries requiring the best performance as possible of the host, such as scientific computing, financial software, AAA games and visual effects will most likely be running C++.

Just like C, C++ is far from perfect. The language is enormous, with very complicated parts (e.g. templates) and compilation for big projects is very slow, the author acknowledges this in [70]. Furthermore, as the language provides a high level of control over the system, it has manual memory management, suffering from the same problems as C. Even with smart pointers (e.g. `unique_ptr`) the problem is not considered solved, as they introduce overhead in the most demanding applications.

2.1.3 Ada

Ada was developed in 1980, during a standardization effort in the USA's Department of Defense, with the goal of unifying projects spanning over 450 programming languages [66]. Ada's main focus was the development of embedded applications, currently the Ada language is mostly used in the critical domain due to the strong emphasis on safety, some Ada success stories are the London Metro Victoria Line and the Paris Metro Line

[59]. The language is also used in several other domains, such as aviation, space vehicles, financial systems and more [19].

In comparison with the other languages in this section, Ada is eclipsed, barely showing in the GitHub rankings [51]. However, given that Ada's compiler is mostly a product, it makes sense that most Ada code is not open-source. Regardless, when one views the list of features Ada has, the first arising question is "*why is Ada not popular?*".

An old article in AdaPower [24] provides some possible insight over the question, referring to the compiler's price and the Hoare's harsh critics. From my point of view, the critics to the compiler and ecosystem pricing still make sense, as access to the full tooling is limited. The lack of programmers goes on to deepen the lack of adoption in the industry and this cycle ends up limiting Ada's reach in the market.

2.1.4 Go

The Go programming language (or `golang`) is a Google project, according the language folklore, it was designed by the authors while they waited for their C++ code to compile. Go tried to address several of the criticisms to C, namely memory management, which it solved by using a garbage collector. While it has made a name for itself in the network and distributed systems sector, being the main language behind projects like Docker [13] and Kubernetes [40], Go's categorization as a systems programming language can be discussed.

Sometimes, however, the performance might not be enough, as was the case for Discord, the popular internet voice server company, as demand increased, Go was not able to meet the expected performance requirements and the company replaced it with Rust [29]. In [70], one of Go's authors, Rob Pike, says that he regrets categorizing Go as a systems programming language, being rather a server programming language that evolved into a cloud infrastructure language. Regardless of discussion, Go has proven to be a viable alternative to existing counterparts, compromising extreme performance in name of safety and simplicity.

2.1.5 Summary

I reviewed four system programming languages, suited for different kinds of environments, C, C++ and Ada can be considered the traditional system languages kind, with a strong emphasis on efficiency and support for embedded devices. Go on the other hand, could be considered a new generation systems programming language, a language for cloud infrastructure. Among the four, only Ada places strong emphasis on safety, with several features allowing for more guarantees at compile time, such as contract based programming and non-nullable types by default, being the only one which does not suffer from the "*billion dollar mistake*" [27].

2.2 The Rust Language

Rust is a fairly recent systems programming language, it started as a side project of the author Graydon Hoare and the language public history dates back to 2010 [26]. In 2012 Mozilla picked up Rust to help develop the Servo browser engine, the successor to the previous Gecko engine; as a way to test Rust's capabilities [35].

2.2.1 What makes Rust different?

In comparison with other languages, one of the first things someone new to Rust ought to notice is the emphasis put on safety. Being a competitor to C++ and achieving memory safety while still providing C++-level performance is quite an accomplishment. Rust, however, also aims to allow users to be productive without sacrificing safety or performance.

The key to all the promises Rust makes is the ownership system and borrow checker. The borrow checker is a completely new mechanism when compared with other mainstream languages. However, it is a product of years of research both in academia and the industry. This mechanism merits most of Rust's accomplishments and also the biggest problem, the learning curve. While Rust has become more accessible over the years, ownership and the borrow checker still require some effort on the part of the developer to learn. I provide a small overview of ownership, the borrow checker and their part in Rust's promise of "*fearless concurrency*".

2.2.2 Ownership

Ownership is the mechanism used by Rust to ensure no memory block stays allocated longer than it is required to. Through ownership, the compiler is able to free memory when required, inserting the respective deallocation calls in the output program. Behind ownership, there are three rules:

> - *Each value in Rust has a variable that's called the owner.*
> - *There can only be one owner at a time.*
> - *When the owner goes out of scope, the value will be dropped.*
>
> — *[68, Section 4.1]*

To illustrate the rules, consider Listing 2.1, where we have two variables `x` and `y`. First, `"Hello"`[1] is assigned to `x`, thus `x` now owns `"Hello"`. After, `x` is assigned to `y`, consider the second rule of ownership, since we can only have one owner, `x`'s value ownership is transferred to `y`. Since we transferred `x`'s value to `y`, `x` is no longer valid, consequently, when compiling the code an error will be issued due to `x` being moved.

Notice how `String::from` is used instead of another type, since `String` type does not implement `Copy` it can only be moved. If the used type implemented `Copy`, the value would have been copied instead of moved.

```
let x = String::from("Hello");  // ok: `x` is assigned "Hello"
let y = x;                      // ok: `x` is moved into `y`
println!("{}", x);              // error: `x` was moved in the previous line
```

Listing 2.1 – Example of the move-by-default mechanism to enforce ownership.

```
let x = String::from("Hello");  // ok: `x` is assigned "Hello"
let y = &x;                     // ok: `x` is borrowed to `y`
println!("{}", x);              // ok: `x` can be printed since it is still valid
```

Listing 2.2 – Example using borrowing to allow for more than one reader on the same variable.

So far this illustrates the first two rules. The last rule can be considered invisible, as it happens during compilation and the user would not notice it usually. What happens is that at the end of the scope, any variable whose owner is in scope, will be freed. While the developer is not required to explicitly free the memory, the compiler will insert the calls for the developer.

2.2.3 Borrowing

If the developer could only copy or move memory the usability of the language would be severely limited. For example, functions that read a variable and produce a new value, not requiring the variable to be consumed would be impossible. To cope with this, Rust allows values to be *borrowed*, in other words, the owner of the variable allows for it to be read by others.

To borrow a value, one writes `&value`, this creates a read-only reference to `value`. There can be an unlimited number of read-only references to a value, but only a single mutable reference. This is discussed in Section 2.2.4. Consider the example Listing 2.2. In the example, `x` is now possible to be printed since it was not moved into `y`. Rather, `y` borrowed `x` through a reference.

Going back to the rules (Section 2.2.2), Rust's references obey them just like all other values. The variable containing them has ownership *over the reference*; it still is a single owner (if `let z = y;` was to be added, the reference would be copied instead of moved); and finally, when the owner goes out of scope *the reference is dropped*, but not the original value.

Mutable Borrows

One last thing to consider are mutable borrows. As previously discussed, in Rust it is possible to create multiple immutable references but only one mutable reference. Regarding mutable references there are two cases to consider:

```
let mut s = String::from("hello");
let r1 = &mut s; // ok: first mutable borrow
let r2 = &mut s; // error: `s` was mutably borrowed in the previous line
```

Listing 2.3 – Example error while using multiple mutable borrows over the same variable.

```
let mut s = String::from("hello");
let r1 = &s;     // ok: first immutable borrow
let r2 = &s;     // ok: second immutable borrow
let r3 = &mut s; // error: `s` was immutably borrowed in the previous lines
```

Listing 2.4 – Example error while using a mutable borrow in conjunction with immutable ones.

N **mutable references,** see Listing 2.3. Understanding why only one mutable reference can exist at a time is trivial, as multiple mutable references to the same object would allow it to be mutated concurrently, which could lead to inconsistent values.

N **immutable references and** 1 **mutable reference,** see Listing 2.4. The reason behind not allowing a mutable reference to coexist is similar. Consider that each value can be executed by a different thread, the first two (`r1` and `r2`) are only read and the third (`r3`) can be read and written. While there will be no conflicts between writers, it is possible for the readers to read an inconsistent value, since it can happen during the write operation.

2.2.4 Concurrency

> *Initially, the Rust team thought that ensuring memory safety and preventing concurrency problems were two separate challenges to be solved with different methods. Over time, the team discovered that the ownership and type systems are a powerful set of tools to help manage memory safety and concurrency problems! By leveraging ownership and type checking, many concurrency errors are compile-time errors in Rust rather than runtime errors. — [68, Section 16]*

Rust provides several kinds of mechanisms to prevent concurrency related problems. Mechanisms as *message-passing*, *shared-state* and traits to enable developers to extend upon the existing abstractions.

Message-passing

Rust's message-passing library is inspired on Go's approach to concurrency, prioritizing message passing over other kinds of concurrent approaches, such as locking.

> *Do not communicate by sharing memory; instead, share memory by communicating. — [16, Concurrency]*

Rust defines channels which have two ends, the transmitter and the receiver. The former can also be seen as the sender, and when is declared with the message type, the latter is also declared with the message type, they can be the same or distinct.

The ownership system comes in when the transmitter sends a message, when received the ownership of the message is taken on by the receiver end. This enforces that values cannot be in both sides of the communication at the same time, preventing concurrent accesses.

Shared-state

Along with message-passing, Rust allows memory to be shared in a concurrent, safe way. Just as before, Rust's ownership system also helps with mutexes' biggest problem, locking and unlocking.

In a language like Java, whenever a thread is able to call `lock` on a mutex, it is required to call `unlock` on it, only then can other threads can use it. The problem is that this approach is subject to human error, forgetting to call unlock or calling unlock in the wrong place is possible. Making use of the ownership system, Rust is able to know when the lock reached the end of the scope and should be dropped.

2.2.5 Why Rust instead of Language X?

The main obstacle between typestates and programming languages is the requirement for aliasing control. In short, typestates are incompatible with aliasing (details are provided in Section 2.3.2).

So to implement a language from the ground up, it is required that aliasing is handled, however, instead of building a new one, the goal is to extend an existing one, Rust. As discussed in Section 2.2.3, Rust's ownership system allows for aliasing control. Using moves to enforce the consumption of values, immutable references for pure functions and mutable ones for limited mutability, it is possible to emulate typestates. This takes care of aliasing concerns.

When designing on top of another language, two more ingredients are required, a powerful extension mechanism (i.e. Rust's macros, discussed in Section 3.2) and a strong type system, able to provide the necessary abstractions. Rust provides both, the type systems goes as far as allowing zero-sized types, allowing type-level abstractions to incur no runtime overhead. This is a key ingredient in our DSL, aiming to provide the minimal possible runtime with an expressive language.

2.3 Behavioral Types

As previously discussed, with the growth in software complexity, developers are required to develop better tools to tame such complexity. Such tools require a theoretical body of work to support them, behavioral types are part of such body of work. The theory behind

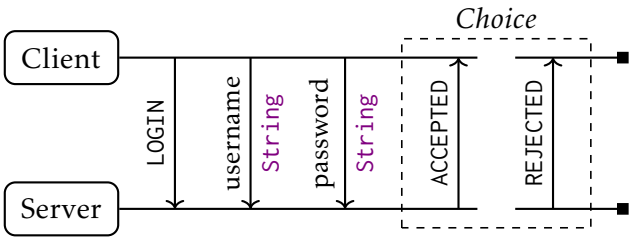

Figure 2.1 – Communication protocol example. The communication establishment step is omitted for simplicity. In this protocol the client tries to login to a service by sending a message `LOGIN` followed by the username and password, both of type `String`. The server then replies with either an `ACCEPTED` or `REJECTED`, if the login was successful or not, respectively.

them encompasses several domains, and they can be applied over a wide range of entities, from an object to a web service.

The work on behavioral types arose in the context of type systems able to capture properties of computations [31]. Session types and typestates are part of this field of study, both capturing distinct property kinds while aiming for a common goal, stronger type systems and better static assurances.

> *Roughly speaking, a behavioral type describes a software entity, such as an object, a communication channel, or a Web Service, in terms of the sequences of operations that allow for a correct interaction among the involved entities. — [2]*

Behavioral types allow developers to model a protocol, define the communication messages and possible interactions and check that certain requirements are met when implementing. Consider the protocol from Figure 2.1, where a user tries to authenticate. A developer can use it as a specification (for simplicity consider the uppercase messages to be simple strings), using behavioral types the developer could be able to specify the described interactions and all boilerplate could be generated for him. While using strings as a payload is not very "*interesting*", consider instead that the object in transit is an encrypted payload, the boilerplate will take care of decryption and deserialization. Furthermore, consider the constraint that *all interactions end with a message from the server*. If the specification has an interaction that is not compliant with such rule, the code should not compile, raising an error.

2.3.1 Session Types

Session types are a subset of behavioral types, focused on communication, from entities in a distributed system to threads in a computer. Session types are based on process calculi and can be thought as "*types for protocols*" **??**. They elevate communication to the type level, allowing expressing them as types in a program, in turn this enables the compiler to reason about the protocol during compile-time [74, 23].

```rust
enum Request {
    Login(String, String),       // login with: {username, password}
    SendMessage(String, String), // send message to a user: {user, message}
    CheckStatus(String),         // check the status of a given user: {user}
}
enum Reply {
    AuthOk,             // successful login
    AuthErr,            // failed login
    MessageOk,          // successful message
    UserStatus(String), // user status
}
fn communicate() {
    let (tx, rx) = channel<Request, Reply>();
    tx.send(Request::Login("foo", "bar"));
    match tx.recv() {
        // ...
    }
}
```

Listing 2.5 – Application login example, modelled using Rust's enums (some channel details were omitted for simplicity). Reusing channels requires the developer to clump all states in a single enum. Better state management requires the use of more channels, neither approaches are ideal.

In Rust, a channel is created with `let (tx, rx) = channel::<SenderT, ReceiverT>()`, where `SenderT` and `ReceiverT` are the types sent and received by the channel. Channels are well-typed, meaning that if `SenderT = String`, sending another type over the channel will result in a type error.

Session types extend on this notion, not only allowing for a single type to be sent or received, but also model the protocol. Consider Listing 2.5, the example has unnecessary complexity, for each receive the developer is required to match all possible replies. Ideally, we declare the steps and possible outcomes beforehand.

For example, in plain English:

1. Send login credentials.
2. If successful, send a message to user `jmgd`.
3. Otherwise, exit.

And now using session types (Figure 2.2 with syntax adapted from [74], where the first four assignments (:=) are simple aliases, to simplify reading):

Consider ! to be *sends*, ? to be *receives*, . the *sequence* operator finally, & the *choice offering* and $\oplus$ the *choice selection*. Using session types effectively offloads complexity to the type system, resulting in more complex types, but simpler implementations, since protocol compliance can be checked during compilation and boilerplate can be added by the compiler. No message is matching required, the compiler does it for the developer. Using session types it is possible to write it in a simpler form, where a type is assigned to each endpoint. Notice how the server provides more than one operation, but the user does not call them all.

$$
\begin{aligned}
\textbf{Login} &:= \{user : \texttt{String},\ password : \texttt{String}\} \\
\textbf{Message} &:= \{user : \texttt{String},\ message : \texttt{String}\} \\
\textbf{Status} &:= \{user : \texttt{String}\} \\
\textbf{SReply} &:= \{status : \texttt{String}\} \\
Server &= ?\textbf{Login}.\oplus\langle !Ok.?\textbf{Message}.\oplus\langle Ok.End, Err.End\rangle, !Err.End\rangle \\
Client &= !\textbf{Login}.\&\langle ?Ok.\&\langle ?\textbf{Status}.!\textbf{SReply}, !\textbf{Message}.\&\langle Ok.End, Err.End\rangle\rangle, ?Err.End\rangle
\end{aligned}
$$

Figure 2.2 – Session type example, equivalent to Listing 2.5.

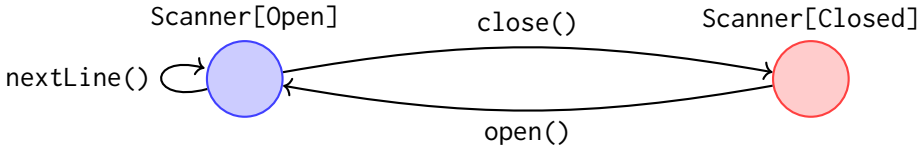

Figure 2.3 – The Scanner typestate automata, based on Listing 1.3.

2.3.2 Typestates

> *... traditional strong type checking was enhanced with **typestate checking** a new mechanism in which the compiler guarantees that for all execution paths, the sequence of operations on each variable obeys a finite state grammar associated with that variable's type. — [64]*

The first language to make use of typestates was NIL [64], afterwards languages like Hermes [63] and Plaid [1] extended the concept with new techniques.

Automata

A possible question on the reader's mind is "*how are automata and typestates related?*". This section tries to address that question and exemplify how automata helps prove typestates properties. It is possible to express typestates as automata, as the reader can observe in Figure 2.3. Each state is a possible state the object can be in, transitions are done through methods. Methods can either mutate the object state, such as `open` and `close`, or leave it unchanged, such as `nextLine`.

Real-world scenario. In production applications, the API is not this simple. In fact, the `Scanner` API is not this simple, however it was simplified for the example. Complex APIs can be designed by a team and implemented by another, specifications can be changed and during project development some details may be lost. Such details can be costly, imagine for example that a method call reaches a state, which has no outgoing edges, but it is not a final state. This is a problem addressed by existing automata algorithms. Representing typestates as automata, extracting all necessary information and applying such algorithms provides the API with extra safety.

The case for typestates

As discussed in Section 1.1, bugs in systems programming are costly, thus, bugs must be minimized. Several tools, such as static analyzers, fuzzers, testing frameworks and others, aid in this purpose, if we have all these external tools, why should we not try and leverage the programming language itself?

Moving towards better languages. Programming languages allow the programmer to express a set of actions to be taken by the computer, they are tools which enable us to achieve a goal. Being essential to our work, better tools enable developers to be more productive and achieve higher quality work. The remaining question is *"why do we not create better languages?"*. Even when considering languages to be cheap to develop, the amount of work between a *working* language to be *production ready* is not cheap. Furthermore, while adopting a new language for a hobby project is easy, the same does not apply for enterprise level projects, requiring several developers to know the ins and outs of the language.

Static typed languages. The current trend is to move from dynamically typed languages, to statically typed ones, or at the very least, add typing support to existing dynamic languages. Typescript [71], Reason [55] and PureScript [53] are all examples of languages built to bridge the gap between static type systems and JavaScript. Python and Ruby, two popular dynamic languages, have also pushed for type adoption with the addition of type hint support in recent releases [72, 58].

Where do typestates fit? Typestates are a complex subject, able to be adopted at several levels, just like type hints, they can be partially used in some languages, through tools such as Mungo [76], by contract-style assertions as in Ada2012, Eiffel or pre-0.4 Rust, or finally by leveraging the existing type system to write typestate enabled code as it is possible in Rust [14]. Typestate-related concepts were also used in Singularity OS [2, Section 6], a reliable operating system prototype where programs were written using Sing# — a language derived from C# which supports behavioral typing, specifically contracts in a similar capacity to typestates.

Why use typestates? By leveraging the state to the type system, the compiler is able to aid the programmer during development, a given set of transitions will be impossible by default, since the types do not implement them. By reducing the need for developers to check for a certain set of conditions through the use of typestates, it becomes possible to reduce the number of runtime assertions and completely eliminate the need for illegal state exceptions since illegal transitions are checked at compile time.

Typestates in action

As a simple example, consider the Java application in Listing 2.6 which simply takes two numbers and multiples them together. The application will throw an exception on line 6, since the programmer closed the `Scanner` in line 5. In this example, the error is simple

```java
public class Mult {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        s.nextLine();
        s.close();
        s.nextLine();
    }
}
```

Listing 2.6 – The `Mult` Java program, which reads two integer and multiplies them together.

```java
public class Mult {
    public static void main(String[] args) {
        Scanner[Open] s = new Scanner(System.in);
        s.nextLine();
        Scanner[Closed] s = s.close();
        s.nextLine(); // compiler error
    }
}
```

Listing 2.7 – The `Mult` program, written in a Java-like *typestated* fashion.

to catch, the program is short and the `Scanner` can either be open or closed, however, real-world applications are not that simple.

In the case of *typestated* programming, the type system will provide the programmer with better tools to express state, furthermore, the compiler will then catch errors regarding state, such as the previous *use-after-close*.

Listing 2.7 shows the `Mult` program written in a *typestated* fashion, notice that the `Scanner` type is now augmented with state and the compiler is able to catch the misuse of the `Scanner[Closed]` interface.

Plaid is a typestate-oriented programming language [1], instead of `class`es users write `typestate`s. Each typestate represents a class and possible states, the class methods and behavior change during runtime as state changes, in contrast with other languages (e.g. Java) where public methods and fields are always available. In Listing 2.8, the `File` passes through states as it is open, read and closed in `readClosedFile`.

This property allows the type system to enforce certain properties at compile time, such as certain methods will never be called in a given state since it is not possible by design (i.e. they are not available in the interface).

Rust. As discussed in Section 2.2, Rust takes the commitment with safety with seriousness, providing the necessary tools to users. While Rust does not support first-class typestates, it is possible to emulate them using the type system (as demonstrated in [14]), this is discussed in further sections of this document.

While the file example does not apply in Rust, since files and other objects are closed as they leave the scopes, enforcing protocols is important and an aspect not covered by

```
state File {
    val filename;
}
state OpenFile case of File = {
    val filePtr;
    method read() { ... }
    method close() { this <- ClosedFile; }
}
state ClosedFile case of File {
    method open() { this <- OpenFile; }
}
method readClosedFile(f) {
    f.open();
    val x = f.read();
    f.close();
    x;
}
```

Listing 2.8 – The `File` declaration and usage in Plaid (taken from [47]).

```
fn main() {
    let protocol = Protocol::new();
    protocol.F1();
    protocol.F3(); // possible crash during runtime
    protocol.F2();
}
```

Listing 2.9 – Rust example of an unchecked failure protocol compliance. The protocol expected operation order is `F1`, `F2`, `F3`, however, the developer placed the operations in the wrong order. This mistake is only caught during runtime.

the language. Consider Listing 2.9, the example is expected to call first `F1`, followed by `F2` and finally `F3`, however such does not happen and the error may only be caught during runtime.

As the next paragraph discusses, this behavior can be prevented using the language's type system. However, such utilization requires complex (and possibly hard to read) types. Since it is not "*part*" of the language, most users will neither use it nor be aware of it.

Embedded Rust. As any systems programming language, Rust penetrated the embedded development space. Providing features in line with the area's requirements, along with community efforts to make Rust viable in embedded systems.

The Embedded Rust Book's [67] Chapter 4 is dedicated to static guarantees, introducing programmers to the concepts of typestate in Section 4.1, and their usage in embedded systems.

As for real-world usage, typestates are abundantly used in the area (not just discussed in the book), under [61] one finds several repositories (suffixed with `-hal`) which implement typestates (e.g. `gpio.rs` from `stm32h7xx-hal`).

```
contract LightSwitch {
    state On {
      int brightness;
    }
    state Off;
    int switchLocation available in On, Off;
}
```

Listing 2.10 – Obsidian state declaration example.

```
transaction OffToOn() {
   LightSwitch s = new LightSwitch(); // LightSwitch is in Off upon instantiation
   s.turnOn();
}
```

Listing 2.11 – Correct state usage example in Obsidian.

```
transaction OffToOff() {
   LightSwitch s = new LightSwitch(); // LightSwitch is in Off upon instantiation
   s.turnOff(); // error: turnOff() requires that s is On, but here s is Off
}
```

Listing 2.12 – Invalid state transition example in Obsidian. Since `LightSwitch` is instantiated as `Off`, calling `turnOff` is not valid.

Obsidian is a language targeting Hyperledger Fabric [21], among other features it makes use of typestates to reduce the amount of bugs when dealing with assets.

In [7] an empirical study tested and proved Obsidian claims, when compared with Solidity, the leading blockchain language, users inserted fewer bugs and were able to start developing safer code faster.

Consider Listing 2.10, in which a light switch is modeled, the same can either be `On` or `Off`, but not both. The `brightness` field can only be accessed if `LightSwitch` is in the `On` state, however the `switchLocation` field can be accessed from both states. Furthermore, consider that upon instantiation, the `LightSwitch` is set to the `Off` state. Notice that in Listing 2.11 the user is able to call `turnOn`, as the switch is in the `Off` state, as expected. However, the user is unable to call `turnOff` in Listing 2.12, since the switch is already set to the `Off` state. The Obsidian compiler is able to notice such invalid transitions and provide an error during compile time.

Chapter 3

Related Work

3.1 Language Preprocessors

Language preprocessors are a mechanism which runs during compilation, some languages will apply the preprocessor during different compilation stages while others will only apply the preprocessor in a single stage.

3.1.1 OCaml

The OCaml ecosystem currently uses OCaml PPX (PreProcessor eXtensions), previous to version 4.02, OCaml made use of p4 (Pre-Processor-Pretty-Printer), also known as Camlp4.

Camlp4 is a parsing library which provides extensible grammars, allowing users to extend OCaml syntax, Camlp4 is also able to redefine the core syntax, OCaml even introduced a revised syntax [54] to enable Camlp4.

In a nutshell, the Camlp4 library would allow developers to develop an extension syntax, when the compiler would pass the source code as text to the preprocessor, which, in turn would generate valid OCaml source code. The library has been deprecated due to being confusing to users and tools alike. Users were required to learn the revised OCaml syntax which complicates the development process. These criticisms are found throughout documents which discuss Camlp4 [77].

PPX

In OCaml version 4.02 syntax extensions were introduced, to enable preprocessor extensions. This meta programming mechanism came to replace p4, which was not well liked by the community given it was too complex. The resources on PPX are not as widespread as the resources for similar mechanisms in other languages (i.e. Rust macros). There are

```
1 let a = 12 [@attr pl]
2 let b = "some string" [@@attr pl]
3 [@@@attr pl]
```

Listing 3.1 – Example of the three kinds of attributes, taken from [56]. The first line attaches to the 12 expression. The second attaches to the whole `let` binding (i.e `let b = "some string"`). Finally, the third line, does not attach to a particular member of the AST.

two main entry points to the PPX system, attribute and extension nodes [42, Sections 8.12 & 8.13].

Attribute Nodes are attached to the existing AST nodes, they are not forcefully compiled, that is, if the compiler is not aware of a matching extension they will be ignored. There are three kinds of attribute nodes (example in Listing 3.1):

- `[@attr payload]` - *attached with a postfix notation on "algebraic"categories.*
- `[@@attr payload]` - *attached to "blocks" such as type declarations, class fields, etc.*
- `[@@@attr payload]` - *not attached to any specific node in the syntax tree.*

— *[42, Section 8.12]*

One of the main kinds of PPXs are *derivers* (see Listing 3.2.2 for the Rust equivalent). Deriveres are mostly used to generate error-prone code where the implementation pattern is common to a series of sittuations. Examples include but are not limited to: comparison functions, pretty printers and serializers [56].

Extension Nodes are similar in syntax to the attribute nodes (instead of `@` they use `%`). Extension nodes are meant to be placeholders in the syntax tree. That means they get replaced with the expanded code (like attribute macros in Rust Listing 3.2.2). They are also required to be expanded by a PPX during compile-time, if such does not happen an `Uninterpreted expression` error is issued.

- `[%attr payload]` - *used for "algebraic" categories.*
- `[%%attr payload]` - *used in structures and signatures, both in the module and object languages.*

— *[42, Section 8.13]*

Ecosystem Presence. The current state of affairs regarding the PPX brings up mixed reactions. From my research, the environment is well maintained, with regular commits to the main PPX repositories. However, the entry-barrier is high due to the lack of introductory materials. Despite this, PPX has seen use in the ReasonML community, more specifically in the ReasonReact framework, where the Tailwind CSS dialect is supported by PPX to enhance developer ergonomics.

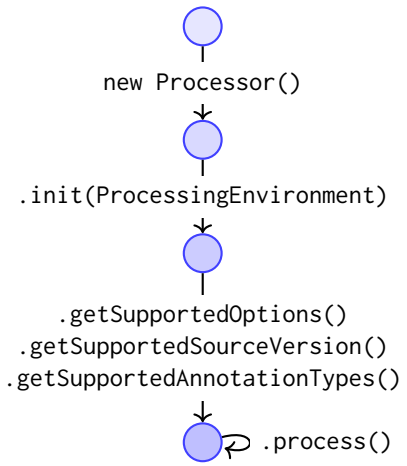

Figure 3.1 – Java's annotation processor lifecycle.

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.ANNOTATION_TYPE)
3 @interface Foo {}
```

Listing 3.2 – Example code for Java's annotation declaration.

3.1.2 Java

In Java, meta programming takes the form of annotations, these are processed by user code during the compilation process. Besides annotations, there is another project able to *"extend"* Java. The ExtendJ research compiler (formerly JastAddJ) [17] aims to provide a *"hackable"* Java compiler for research purposes, such as static analysis tool development to Java features prototyping.

Java Annotations

Java annotations were first introduced in Java 5 [10], they are a form of metadata which can be added to Java source code. Annotations can be used in conjunction with several components of the Java language, such as classes, interfaces, documentation and others. These are processed by build-time tools or by run-time libraries to achieve new semantic effects, a popular example of such library would be the compile-time dependency injection framework Dagger 2 [9]. Another popular library using annotations is the Checker Framework [6], besides the classic `@NonNull` example, the tool provides several other kinds of annotations. The annotations are then checked by Checker Framework annotation processor. An example would be the `@Tainted`/`@Untainted` annotations, which serve the purpose of annotating data to indicate whether it can be trusted. This helps avoid potentially harmful code from being executed (e.g. malicious SQL queries).

Implementing an Annotation. To implement an annotation, start by declaring it as in Listing 3.3. The annotation may contain parameters that allow to add configuration when the declaration is used. Supported types are:

- Primitive types (e.g. `int`, `long`, etc).
- `String`
- `Class<T>`
- `enum` types.
- Other annotation types.
- An array of the above.

At this point the annotation is processed by the compiler but does not do anything useful. To address that, the code needs to either handle the annotation at runtime, through reflection. Or at compile-time, through an annotation processor. Since processing the annotation at runtime incurs a cost, I'll only discuss the annotation processor approach.

Annotation Processor. Putting it simply, is a specific class registered at compile-time as able to process annotations. The compiler can then make use of the class to process the new annotations. The class itself will usually extend the `AbstractProcessor` class, overriding some methods present in Figure 3.1. The processor will then be called for each annotation belonging to the package.

Annotation processors are also able to generate code. This is usually done by means of a library such as JavaPoet [60]. After generation, the output code is then compiled and subject to the same treatment as handwritten files. If the generated code, generates more code, this process repeats itself until no more code is generated.

Ecosystem Presence. Java annotations are ubiquitous. Examples include but are not limited to the development of REST APIs, Android applications and database tools. As discussed in Section 3.1.2, annotations are picked up by several tools and serve a plethora of purposes, from cutting boilerplate to providing an extra layer of security and assurance. However, being ubiquitous does not imply that resources are widely available. Learning to develop annotations seems to be an almost exotic topic in Java, with few quality resources available.

ExtendJ & JastAdd

ExtendJ is an extensible compiler aiming at facilitating the development of Java compiler tools. The compiler supports Java from version 5 up to 8. The extensions are written in JastAdd [33], a meta-compilation system upon which ExtendJ is built. It is possible to extend the compiler during any of the following phases: *Scanning*, *Parsing*, *Analysis* and *Code Generation*. Extending the language with new syntax requires the modification of the *Scanner* and *Parser*. The *Analysis* phase occurs after parsing, when types are checked. Hence, to extend type analysis, one must modify it in the compiler. Finally, *Code Generation* has two possible extension methods in ExtendJ: *direct bytecode generation* and *desugaring*. The latter being the simpler approach and recommended being tried before the former. Desugaring can be used to prototype new languages constructs, by mapping them to the respective Java code.

```
1 @any Person people;
2 people += new Person("Bob");
3 people += new Person("Gene");
4 people += new Person("Tina");
5 for (Person person : people) {
6     System.out.println(person.getName());
7 }
```

Listing 3.3 – The `@any` annotation allows an object to carry several instances of itself. In the example, `@any Person` is rather a collection of `Person`. This extension is enabled by the ExtendJ compiler.

Ecosystem Presence. While both ExtendJ and JastAdd are powerful tools, they lack of support for versions after Java 8. Their usage is generally confined to academia being unsuited for industrial usage. Documentation on getting up and running is also limited, being mostly based on papers and examples rather than entry-level explanations.

3.1.3 Kotlin

While Kotlin also allows and makes use of Java annotations, it is also possible to write plugins for the Kotlin compiler. Compiler plugins are much more complex pieces of software in comparison with annotations, due to the amount of detail required to take into account. An example of such detail is the amount of Kotlin backends available, not all targeting the Java Virtual Machine. This is also a motivation to write a compiler plugin, as annotations may not be compatible with all backends.

Kotlin Compiler Plugins

The Kotlin compiler plugin stack is illustrated in Figure 3.2. From top to bottom, the first two parts are related to Gradle, the main build system for Kotlin. These parts to not work on the plugin itself, but rather help the plugin coexist with the rest of the Kotlin ecosystem.

`Plugin`. The plugin interacts only in the Gradle segment, it provides an entrypoint from a `build.gradle` plugin and allows plugin configuration.

`Subplugin`. The subplugin acts as the glue between Gradle and Kotlin. It sets up a series on options for the layer below from the configuration provided in the first layer. It also defines a plugin ID to avoid name clashing with other plugins and Maven coordinates, which allow the plugin to be downloaded.

`CommandLineProcessor`. This layer reads the arguments for `kotlinc -Xplugin`. The options from the previous layer are passed through here. Finally, it writes `CompilerConfigurationKeys` which will be passed to the layer bellow.

`ComponentRegistar`. This component just reads the passed keys and registers extensions to be used by the compiler. It is possible to register several extensions at a time.

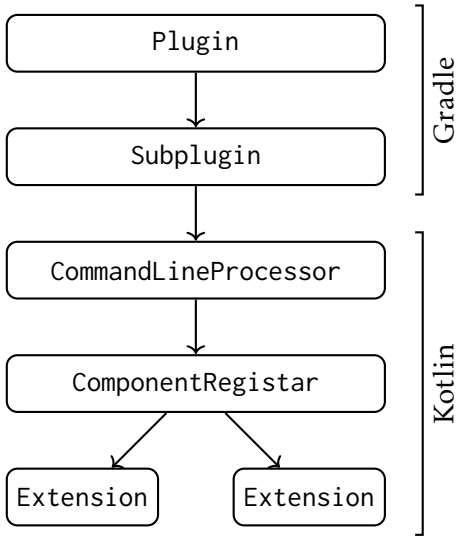

Figure 3.2 – Kotlin compiler plugin architecture stack [46].

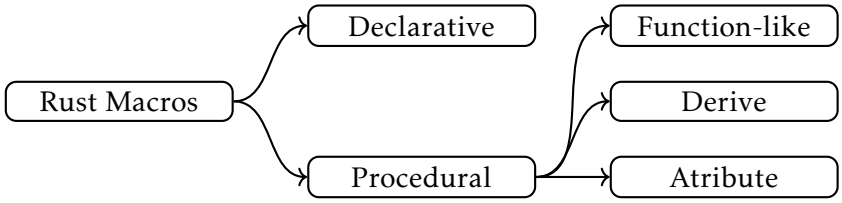

Figure 3.3 – Rust macro's family tree

`Extension`. The extension is the main part of the plugin. There are multiple types of extensions able to deal with the input at different levels, from the class level to the code generation.

Ecosystem Presence. Just like the previous languages, the Kotlin compiler plugins suffer from the same discoverability problem. While depending on it are widely used (e.g. Kotlin serialization [37]), the resources to learn how to develop such tools are rare.

3.2 Rust Macros

Just like C and C++, Rust offers macros as part of the language. In essence, Rust macros are just like other languages macro's, running during compile-time to generate code. In Rust, macros refer to a family of features (see Figure 3.3), *declarative* macros and *procedural* macros.

3.2.1 Declarative Macros

Declarative macros (also known as *macros-by-example*) can be declared with `macro_rules!` and are called with function syntax (see Listing 3.4).

> *Each macro by example has a name, and one or more rules. Each rule has two parts: a matcher, describing the syntax that it matches, and a transcriber, describing the syntax that will replace a successfully matched invocation. Both the matcher and the transcriber must be surrounded by delimiters. Macros can expand to expressions, statements, items (including traits, impls, and foreign items), types, or patterns. — [69, Section 3.1]*

Transcribing. When a macro is invoked, the macro expander loops through the declared rules, transcribing the first successful match. It transcribes the first successful match; if this results in an error, then future matches are not tried. An error is thrown if the compiler cannot determine unambiguously how to parse the macro [69, Section 3.1 - Transcribing].

Metavariables. To specify a macro a user first declares a pattern which will match a given form of syntax. *Metavariables* are used to achieve such goal, they are declared with "`$ name : fragment-specifier`" in the macro matcher and can match thirteen different kinds of syntax fragments [69, Section 3.1 - Metavariables]. In Listing 3.4, the metavariable `n` is of kind `literal` which will match literals such as `'E'`, `"Elite"` and `420` [69, Section 8.2.1].

Repetitions are indicated by placing the tokens to be repeated inside `$(...)`, followed by a repetition operator, optionally with a separator token between. This is valid both for the matcher and the transcriber. Repetition operators are the same as the regular expression ones:

- `*` — indicates zero or more repetitions.
- `+` — indicates at least one repetition.
- `?` — indicates zero or one repetition.

Hygiene works by attaching an invisible *syntactic context* to all identifiers [78]. Identifiers are compared over two pieces of information, the *textual value* and their *syntactic context*. The textual value consists of the variables name (e.g. `four`), the syntactic context is a kind of scope added to variables declared inside the macro. This is done to keep the macro declared variables from interfering with existing ones.

When expanding a declarative macro[1] variables declared inside the macro belong in a different scope, consider the macro declared in Listing 3.5 and the respective expansion in Listing 3.6. As illustrated by Listing 3.6, line 2 is considered to be in a different context than the rest of the expanded code. This will rightfully raise an error (shown in Listing 3.7), since line's 3 `a` will not exist due to not being in the same syntactic context as line 2.

[1] The same mechanism does not apply to procedural macros, which are not hygienic. Their output will interfere with existing code if precautions are not taken [39].

```
1 macro_rules! say_hello {
2     ($n:literal) => { for 0..$n { println!("Hello, world!"); } }
3 }
4 fn main() { say_hello!(5); }
```

Listing 3.4 – Example `macro_rules!` usage. When executed, the code above will print "`Hello, world!`" five times.

```
1 macro_rules! using_a {
2     ($e:expr) => { { let a = 42; $e } }
3 }
4 let four = using_a!(a / 10);
```

Listing 3.5 – Definition of the `using_a` macro and usage. The macro simply declares a variable `a`, set to 42 and then writes an expression which was passed in.

```
1 let four = {
2     let a = 42;
3     a / 10
4 };
```

Listing 3.6 – Listing 3.5 line 9's macro expansion. Declarations with a blue background will be placed in a different *scope* than the others, thus the `a` for lines 2 and 3 will not be considered the same.

3.2.2 Procedural Macros

Rust also has another macro mechanism, *procedural macros*, these can take three forms: *function-like macros*, *derive macros* and *attribute macros*. In a nutshell, procedural macros allow users to run code at compile time, consuming and producing Rust syntax.

Function-like Macros

Function-like macros and declarative macros are similar regarding invocation, being indistinguishable from each other, and output, completely replacing the original call. However, the similarities stop there as their implementation methods are completely different.

```
1 error[E0425]: cannot find value `a` in this scope
2   --> src/main.rs:13:21
3    |
4  9 | let four = using_a!(a / 10);
5    |                     ^ not found in this scope
```

Listing 3.7 – The expansion in Listing 3.6 will result in an error during compile time since the `a`s in line 2 and 3 are considered to belong to different contexts.

```
1  html! {
2      h1 { "Hello, world!" }
3      p.intro {
4          "This is an example of the "
5          a href="https://github.com/lambda-fairy/maud" { "Maud" }
6          " template language."
7      }
8  }
```

Listing 3.8 – HTML DSL embedded in Rust. Example taken from [79].

Definition. Function-like macros are defined by a public function with the `proc_macro` attribute and a signature of type `(TokenStream) -> TokenStream`. Everything contained inside the call delimeters of the macro invocation is input to the function, as previously referred, the output will completely replace the macro call.

Domain Specific Languages. While the macros discussed next also provide their contribution for domain specific languages in Rust, function-like macros provide the necessary tools to write an embedded DSL. The Rust ecosystem developers have developed HTML DSLs [79, 62] (see the example in Listing 3.8) and the possibility to run Python inside Rust[32].

Derive Macros

Derive macros likely are the most common kind of procedural macro in Rust, they are usually used to *derive* a `trait` implementation from a `struct` (see Listing 3.9). They define new inputs for the `derive` attribute, and can also create new items given the token stream of a `struct`, `enum` or `union`.

Definition. Just like function-like macros, derive macros are defined as a public function with the `proc_macro_derive` attribute and a signature of `(TokenStream) -> TokenStream`. The input is a token stream of the item with the `derive` attribute, the output is a set of items that are appended to the module or block where the input token stream is in. In Listing 3.9 the `Debug` implementation will be appended to the end of the structure.

Helper Attributes. Derive macros are also able to add additional attributes to the scope of the current item. Such attributes are called *derive macro helper attributes* and they are *inert*, that is, they are not processed by themselves but rather serve as annotations (see Listing 3.10).

Attribute Macros

Attribute macros define new outer attributes, in contrast to the attributes discussed in Listing 3.2.2, attribute macros are processed as independent units and not as an annotation. They can be attached to items (see [69, Section 6]), including items in `extern` blocks, inherent and trait implementations, and trait definitions.

```rust
#[derive(Debug)]
struct Coordinate {
    x: f32,
    y: f32,
    x: f32,
}
```

Listing 3.9 – Example usage of `#[derive(...)]`, in this case deriving `Debug` enables the structure to be printed with "`println!("{:?}", coord)`".

```rust
#[derive(Error)]
enum CoordinateError {
    #[error("Invalid coordinates {0}")]
    InvalidCoordinates(Coordinates),
}
```

Listing 3.10 – Example usage of a derive macro with helper attributes, in this case the `error(...)` defines an error message with a `Coordinates` parameter.

```rust
#[get("/hello/<name>/<age>")]
fn hello(name: String, age: u8) -> String {
    format!("Hello, {} year old named {}!", age, name)
}
```

Listing 3.11 – Attribute macros are commonly used in web frameworks to provide an easy way to declare an endpoint. In this example (taken from [57]) the user declares that `GET` requests to `hello/` have two path parameters (`name` and `age`) and should be handled by the `hello` function.

Definition. Like the other macros, attribute macros are also declared by a public function with the `proc_macro_attribute`, however, their function signature takes two parameters instead of one, being `(TokenStream, TokenStream) -> TokenStream`.

The first parameter is the token tree following the attribute name, for example, in Listing 3.11 it would contain the token tree of `("/hello/<name>/<age>")`, in the case the attribute is written as a bare attribute name (e.g. `#[attribute]`), the token tree is empty.

The second parameter is the token tree of the item the macro is attached to, the function output will *replace* such item with the return item or items.

While attribute macros are able to replace the input stream, they can also leave the stream unchanged and check for code properties (e.g. if all variables start with a given prefix).

3.2.3 Summary

In summary, Rust enables metaprogramming through macros, the same can be divided into two categories, declarative macros, with work through pattern matching, and procedural macros. Their main characteristics are summarized in Table 3.1.

Macro Type	Input Processing	Output Processing	Invocation
Declarative	Pattern Matching	Replace	`macro!`
Function-like	User programmed	Replace	`macro!`
Derive	User programmed	Append	`#[derive(...)]`
Attribute	User programmed	Replace	`#[attribute]`

Table 3.1 – Rust macros properties summary.

Declarative macros (Section 3.2.1) work mainly through pattern matching, they are the best tool to avoid code repetition without putting in the effort of writing a token parsing macro. However, for more complex tasks, declarative macro's readability quickly degrades leading to an unpleasant developing experience.

Procedural macros (Section 3.2.2) can be further subdivided into three categories, being *function-like*, *derive* and *attribute* macros. Function-like macros can be considered as an alternative to declarative ones, they allow for more functionality and flexibility being possible for the code behind them to be replaced from one to the other without changes on the user's part. In comparison with other procedural macros, function-like macros allow for the creation of an embedded DSL inside Rust while the others are mainly annotations. Derive macros are mainly used to extend existing structures with traits that can be derived automatically (e.g. `Debug`). Finally, attribute macros can be used to modify existing code or simply check for code properties (e.g. if an `enum` fields are sorted).

3.3 Approaches to Behavioral Types

As previously discussed, there are several kinds of approaches to behavioral types, some aim to bridge modern languages and behavioral types, others build a language from scratch. Building a new language is a more attractive approach since there is no requirement for retrofitting. This approach is more common in the typestate domain, with Vault and Plaid being prime examples. The library approach receives more attention from the session types domain, where projects aim to implement them in existing languages such as Java, Go and Rust.

3.3.1 Session Types

As established so far, session types are mostly used for communication protocols, defining message types and their order in a conversation. Session types also share common ground with typestates as works StMungo [11, 38, 76] and others [23, 73] demonstrate.

StMungo it is a transpiler from Scribble [82] to Java based on session types and typestate. The transpilation process takes Scribble local protocols as input, generating Mungo typestate specifications and Java skeleton implementation code. The output is then checked by Mungo [11, 38, 76]. This process is based on a formal translation of session types into

typestate specifications for channel objects, and extends the translation from binary to multiparty session types.

Session Types for Rust. As far as I am aware, the work on Rust session types was started by Jespersen et al. [34], such work was limited since it only supported binary session types for channels. It builds on a Haskell-based approach [52], mirroring the implementation interface.

The type constructs in the original session types formulation have correspondents in the Rust implementation, this is part of a DSL embedded in the Rust type system. The library makes use of `unsafe` to allow for transmutation (i.e. unsafe type casting) and sending untyped values over the channels. Finally, the library is able to provide compile-time safety, that is, the code will not compile if the channels protocols do not match.

Multiparty Session Types for Rust. Work on multiparty session types started with Lagaillardie et al. [41]. This work makes use of the Scribble [82] toolchain, just like StMungo; and it is a thin wrapper over previous work done by Kokke [36]. Similarly to the previously presented work [34], this work also takes advantage Rust's type system to provide compile-time safety. While using Scribble allows the library to make use of a tried and tested toolchain, it also implies the usage of an external tool, which in previous works was not necessary [34, 36].

3.3.2 Typestate

In the work of Ancona et al. [2, Section 2.3] several approaches to typestates are enumerated. While most approaches create a new language, approaches like Fugue [12] simply build on top of existing languages. This kind of approach is extremely valuable as it bridges the gap between existing programming languages and the theoretical field.

Vault is a programming language with the aim of researching lifetime tracking and the symbolic state of objects [18]. Vault introduces two new concepts — *adoption* and *focus*, which serve to relax constrains imposed by a linear type system. Since aliasing can be controlled through the linear type system, Vault is able to check for states, hence supporting typestate. Vault bridges the best of both worlds by splitting programs into two groups: the ones able to be checked for protocols (i.e. *typestated*) and the ones free of aliasing restrictions and thus unable to verify protocol rules.

The adoption mechanism works by means of an *adopter* (i.e. which adopts a linear reference) and an *adoptee* (i.e. the adopted reference). Through adoption, the adopted linear reference is consumed, and thus cannot be directly accessed. Furthermore, the lifetime of each reference alias is tied to the lifetime of the adopter. When the adopter is freed, all adopted references recover their linear type.

The focusing mechanism provides a temporarily linear view on a nonlinear object. The focused object is required to be live and of the same type in the end of the focus

```
[WithProtocol("open", "closed")]
class OuterSocket {
    [InState("connected", WhenEnclosingState="open"),
        NotAliased(WhenEnclosingState="open")]
    [Unavailable(WhenEnclosingState="closed")]
    private Socket innerSocket;
}
```

Listing 3.12 – Relating a class's states with the `innerSocket` states. In this example, the `OuterSocket`'s `open` state is related with the `connected` state of the socket. This ensures that the `OuterSocket` is a well-behaved client of `innerSocket`.

usage. Access to the parent of the focused object is temporarily revoked, disabling alias access.

Fugue is a software checker that enables interface protocols (i.e. typestates) to be specified as annotations [12]. It provides two main protocol checking functions, *resource protocols* and *state-machine protocols*. Resource protocols relate to the allocation and release of resources, since Rust takes care of such concerns through ownership I will not discuss this feature of Fugue.

State-machine protocols allow the programmer to constrain the sequence of method calls on an object. This is also known as typestate, as one can only transition between valid states. In Fugue, the developer adds annotations to the object's methods and from them, a state-machine is derived. Fugue also allows for states to relate to one another. Consider the example in Listing 3.12; by relating the states in the `OuterSocket` class with the `innerSocket` field states Fugue can ensure that `OuterSocket` is a well-behaved client of the field's class.

Plaid is a typestate-oriented language. The idea, proposed in [1], is based on support for first-class typestates in an object-oriented setting. In Plaid, objects are described by their state rather than members. While the object is able to have fields common to all states, there is also the possibility for fields to be exclusive to a given state. For the example of the `File` which can be either in the `Open` state or the `closed`, the former state would have an OS file descriptor, while the closed state would not. The path to the file could be available for both states, since it would allow the file to be re-opened.

In Plaid, methods can make the object transition between states. Building on the file example, the method `open` would transition the file from the `Closed` state to the `Open` state. Plaid also introduces a series of aliasing control keywords, `unique` disallows aliasing on an object while allowing for state transitions, `immutable` disallows mutation (i.e. state transitions), `shared` makes an object behave like it normally would in Java, allowing aliasing (since it allows aliasing, it also requires runtime checks over state on sensitive operations).

Mungo is a static analysis tool [11, 38, 76] for Java programs. It checks typestate properties and can be divided into two components, a Java-like language to define typestate

```
1 @Typestate("StateIteratorProtocol")
2 class StateIterator { /* ... */ }
```

Listing 3.13 – Mungo's `Typestate` annotation. Normal Java code ends up ignoring the annotation. However, Mungo is able to process it and check the class calls against the specification to ensure typestate compliance. In this case the class specification is `StateIteratorProtocol`.

specifications and a typechecker, which checks that objects follow the typestate specification. Specifications are written as separate files and can then be used in Java classes through annotations, as demonstrated in Listing 3.13. The annotations enable Mungo to be unobtrusive in projects since annotations are not required to be processed (as seen in Section 3.1.2).

> *If a class has a typestate specification, the Mungo typechecker analyses each object of that class in the program and extracts the method call behaviour (sequences of method calls) through the object's life. Finally, it checks the extracted information against the sequences of method calls allowed by the typestate specification. — [11, Section 1.2]*

3.3.3 Summary

In summary, behavioral types is a topic which for now, is still mostly confined to the academia circles. Despite the efforts put into the development of tools for "*business*" languages, the tools were either abandoned (e.g. Fugue) or superseded by other developments in the field (e.g. the initial work in session types for Rust). Languages developed for research purposes (e.g. Vault and Plaid) seem to make little to no effect on the outside world. While adoption of the language itself is not expected, such could be expected for the mechanisms, though it does not seem to be the case. Finally, in the case of tools (e.g. Scribble and Mungo), they seem to pick the most traction from academia. The motivation seems to be based on the possibility of extension and continuous improvement. However, they seem to suffer the same destiny as others, causing little to no impact in the outside world.

Chapter 4

Contributions & Planning

4.1 Objectives

As previously discussed, the present work proposes the design of a new DSL embedded in Rust. The core goal of the DSL is to provide typestates without getting in the way of the developer. This means providing a syntax that stays close to Rust, only introducing new constructs when required. Errors should be helpful and point to the original DSL code (see Listing 4.2) , avoiding scenarios like the "*infamous*" C++ template errors. Features should be well documented allowing anyone to get up and running quickly, along this idea, the language itself should be a simple dependency and avoid any complicated build process.

Finally, as discussed in Section 1.4, the language should provide static guarantees — such as state usefulness and automata minimality. That is, respectively, if a state is reachable from the start state and if it is possible to reach the end state from it; and if the automata is minimal. Along with usefulness checks, the presence of start and final states are also checked, otherwise such check would not work.

$$Start \rightarrow A \; ; \; A \rightarrow End \tag{4.1}$$

$$Start \rightarrow End \; ; \; Start \rightarrow B \tag{4.2}$$

$$Start \rightarrow End \; ; \; B \rightarrow End \tag{4.3}$$

As an example, consider the previous equations as simple state machines. From the three, the last two would not compile as it is impossible to reach End from B, and it is impossible to reach B from $Start$, respectively.

4.2 Architecture

The DSL should provide a specification language, allowing the developer to describe a structure, along with possible states and functions — which can be pure, impure (i.e. mutating some aspect of the structure without transitioning states) or transitions (i.e. making the transition between states). The specification is then transformed into the necessary Rust structures and traits. In the end, the developer should only be required to write the `impl`s for each state.

4.2.1 Selecting the correct macro

In short, there are two valid candidates, function-like macros and attribute macros. From the two I selected attribute macros, not only they are easier to parse (since I can leverage existing libraries), they are also close to Fugue [12]. The latter provides validation over the overall approach, as Fugue constitutes a tool developed in an industrial environment, for real world software.

Derive macros seem to be a great candidate for our purpose. We can add them to an enumeration (just like [20]) and derive the necessary states from there. The main limitation of this approach is the lack of transparency — the derived methods are fixed by the macro and the developer does not have control over them.

Attribute macros Initially, I thought that leveraging attribute macros would not work. Macro expansion does not keep state hence I could not relate several attributes together. However, attribute macros are attachable to modules and inside them, one can write any kind of Rust code. This approach allows full control over the module token stream, thus enabling processing and analysis over user code.

Function-like macros enable expressive DSLs. Their input can be *almost* arbitrary and as previously discussed it is even possible to sketch a full language inside them. In comparison with attribute macros they allow for more flexibility, however this may come at the cost of requiring the user to learn a new language, even if small.

4.2.2 How does it work?

As stated, this approach provides a DSL. Such DSL is responsible for allowing the developer to write an expressive typestate specification. The DSL processing is done as follows (illustrated in Figure 4.1):

1. The DSL is first parsed during macro expansion.
2. The extracted structure is converted into a state-machine.
3. The state-machine is verified against our checks (see Section 4.2.5).
4. Extract all required `trait`s and `struct`s from the specification.
5. Expand the macro into valid Rust code.

After this process, all that is left to do is the implementation of the `trait`s.

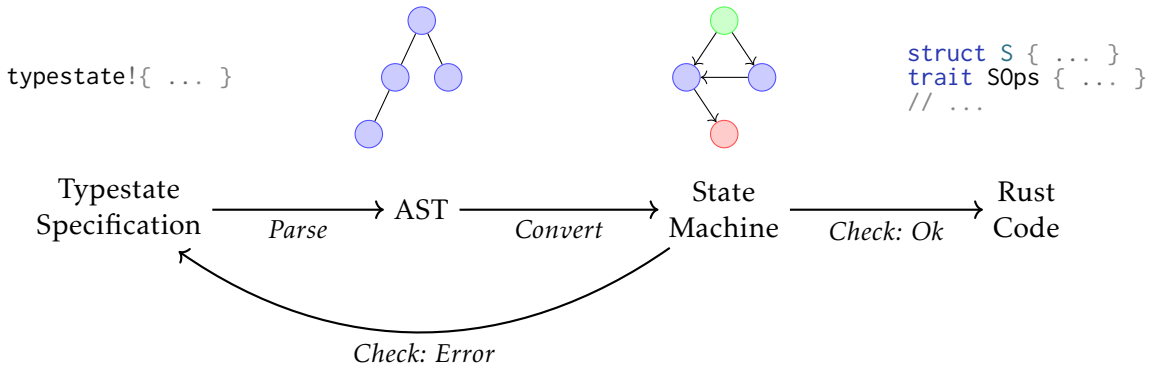

Figure 4.1 – From DSL specification to Rust code. First the DSL is parsed, then converted to a state machine and the properties checked (in the case some property is not respected, an error is issued). Once the properties are validated, the Rust code is generated.

4.2.3 Syntax

The macro is invoked by attaching the `#[typestate]` attribute to a module declaration. This section reviews some important aspects, such as state usage and possible enhancements to function declarations. The syntax itself is the same as Rust and can be seen in Listing 4.1.

The module declaration is attached with `#[typestate]`, this is the macro entrypoint (line 1 of Listing 4.1). This syntax separates different typestates into individual modules. The restriction can be lifted by adding a macro parameter that does not include `mod` in the final expansion.

The main structure is declared inline with all others, the key difference being the lack of attributes (lines 2-4 of Listing 4.1). The structure could be omitted in the case that no variables are required in all states, the name could then be assumed to be the one in the module, or also given as parameter to the macro.

A state declaration uses the `struct` keyword (lines 5-13 of Listing 4.1). The choice of `struct` over `state` is based on the following reasons:

- Keeping the syntax close to Rust by avoiding a new keyword.
- Each state boils down to a structure, hence it is a natural mapping made explicit.
- Since the code inside the DSL already implies typestates, each structure can easily be thought of as a state. This follows the previous point idea.

A function declaration in regular Rust, without attributes (lines 14-20 of Listing 4.1). They can be one of three possible categories:

- `pure` functions, which do not mutate state (e.g. line 14 of Listing 4.1). Their signature takes an immutable reference to `self` as an argument, disabling mutation.
- `impure` functions, which can mutate state but not transition between states (e.g. line 15 of Listing 4.1). Their signature takes a mutable reference to `self` as an argument.

This allows for mutation of the current state, but not transition.

- `transition` functions, which transition between states (lines 16-20 of Listing 4.1). Their signature takes ownership of `self`, forcing the consumption of the current state disabling possible aliasing.

These attributes can be explicit, but they can also be inferred from the code. The reader may have noticed two details in Listing 4.1 — the `Flying` state is not used, the `fly_to` function returns the same state as `self`. This is the case since the function will make the transition between `Hovering`, `Flying` and back internally. Thus requiring `self` to be consumed.

Initial and final states are special cases as they are defined with both structures and functions. The syntax to declare initial and final states is presented in lines 17-18 of Listing 4.1. As the reader can observe, the syntax uses functions to declare with states are initial and final. They can be declared explicitly or be inferred using the following rules:

- **Initial states** are declared with functions that do not take `self` as a parameter and return a state. The `self` keyword is omitted since the state is yet to be built.
- **Final states** are declared as "*consumers*", that is, they take `self` as a parameter and do not return a value. This syntax makes clear that the state becomes unavailable for further processing.

Non-deterministic transitions are useful to cover cases where the output is dependent on some external factors. Consider a process that can fail, the developer is required to model both the success and failure cases. To do so, the developer can declare an `enum`, containing possible outcomes. Such enumeration has one key difference from regular Rust enumerations, the fields must contain existing states only, that is, structures declared inside the same macro call (lines 8-11 and 15 of Listing 4.1). Enforcing these transitions can either be done by leveraging Rust's enumerations and leaving the matching up to the user, or through determinization of the automata.

4.2.4 State-Machine Conversion

The reader might have noticed that the specification syntax cleanly maps to a state machine. As discussed in Section 4.2.3, each `struct` is a state and each function is either a transition or a function which does not change state. From the code, we can establish states as nodes and functions as edges, effectively converting the specification into automata.

4.2.5 State-Machine Verification

After extraction the state-machine is subject to verification, this process is done using existing algorithms [28], to check the following properties:

- **Minimality.** The state machine should be minimal in the sense that there should be no redundancy in states and transitions.

```rust
#[typestate] mod M {
    struct Drone { location: Coordinates } // available in every state structure
    #[state] struct Grounded;
    #[state] struct Hovering;
    #[state] struct Flying {
        destination: Coordinates
    }
    #[state] enum Landed {
        Success(Grounded),  // touchdown!!
        Error               // crashed
    }
    fn get_location(self: &Grounded) -> &Coordinates;
    fn correct_coordinates(self: &mut Grounded);
    fn take_off(self: Grounded) -> Hovering;
    fn fly_to(self: Hovering, dst: Coordinates) -> Hovering;
    fn land(self: Hovering) -> Landed;
    fn new() -> Grounded;   // defines the start state
    fn end(self: Grounded); // defines the final state
}
```

Listing 4.1 – Example specification of the Drone typestate using the proposed DSL.

```
#[state] struct State;
                ^^^^^ : error `State` is not a useful state
```

Listing 4.2 – Example error issued by the DSL.

```rust
trait DroneState {}
struct Grounded { location: Coordinates }
impl DroneState for Grounded {}
struct Drone<State: DroneState> { state: State }
trait GroundedOps {
    fn get_location(&self) -> &Coordinates;
    fn take_off(self) -> Drone<Hovering>;
}
```

Listing 4.3 – Example generated Rust code for the Grounded state. Notice the DroneState trait, which bounds valid drone states. The trait should follow the sealed trait pattern, but it was simplified in this example.

```rust
impl GroundedOps for Drone<Grounded> {
    fn take_off(self) -> Drone<Hovering> { /* ... */ }
}
```

Listing 4.4 – To make the drone usable, the developer must implement the generated traits. In this case, only the Grounded state is considered.

- **Non-empty language.** The language of the state machine should not be empty; thus we need to ensure the presence of final states.
- **Usefulness.** As discussed, all states should be reachable from the start state and be able to reach the end state.

In this case, the user should get feedback over any infraction, explaining the reasoning behind the error or warning.

4.2.6 Developer Workflow

The final user workflow should be as follows:

1. The developer starts by writing the specification, as in Listing 4.1.
2. When done, the developer compiles the code. In turn, the compiler will begin the process of expanding the macro. This process is illustrated in Figure 4.1.
    - During expansion, if the state machine fails any of the verifications described in Section 4.2.5, a compiler error is issued.
    - If no errors are reported, the process continues as normal and generates all required boilerplate, like in Listing 4.3. This boilerplate does not appear for the user, hence it is required the generation process is as predictable as possible.
3. After generation, the user is now required to implement all functions, as in Listing 4.4.

4.3 Previously Developed Work

Currently, I have developed a really simple proof-of-concept macro for Rust typestates. The macro is based on declarative macros and has several limitations, the syntax is also distant from the current proposal. The macro takes a simple typestate specification and expands into several traits and structures. This project can be found on GitHub [15].

4.3.1 Improvement Points

As it stands, the project has several possible improvement points to be addressed during the development phase. Such points are as follows:

- The macro's syntax does not support functions. This implies that it does not support transitions.
- The fact that the macro is written using declarative macros makes the underlying code complex and barely readable. This is due to the fact that declarative macros are not suited for bigger scale macros.
- The macro does not verify state-machine properties, such as the previously enumerated, reachability and termination.
- The supported syntax adds new language constructs, such as the `limited` and `strict` keywords.

- The macro only supports *"phantom"* states, that is, states do not hold any data (the name comes from Rust's `PhantomData`, which is used to implement these kind of states).

References

[1] J. Aldrich et al. "Typestate-oriented programming". In: *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*. New York, New York, USA: ACM Press, 2009, p. 1015. ISBN: 9781605587684. DOI: 10.1145/1639950.1640073. URL: http://www.cs.cmu.edu/%7B~%7Daldrich/papers/onward2009-state.pdf%20http://dl.acm.org/citation.cfm?doid=1639950.1640073.

[2] D. Ancona et al. "Behavioral types in programming languages". In: *Foundations and Trends in Programming Languages* 3.2-3 (2016), pp. 95–230. ISSN: 23251131. DOI: 10.1561/2500000031.

[3] J. Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. ISSN: 00361445. DOI: 10.1137/141000671. arXiv: 1411.1607.

[4] D. M. R. Brian W. Kernighan. *The ANSI C Programming Language*. 2nd ed. Prentice Hall, 1988. ISBN: 9780131103627,0131103628.

[5] D. Campbell. *The many human errors that brought down the Boeing 737 Max*. 2019. URL: https://www.theverge.com/2019/5/2/18518176/boeing-737-max-crash-problems-human-error-mcas-faa.

[6] Checker Framework. "The Checker Framework Manual : Custom pluggable types for Java". In: 0.page 15 (2018), p. 253. URL: https://checkerframework.org/.

[7] M. Coblenz et al. "Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in Obsidian". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–28. ISSN: 2475-1421. DOI: 10.1145/3428200.

[8] M. Coblenz et al. "Obsidian: Typestate and assets for safer blockchain programming". In: *ACM Transactions on Programming Languages and Systems* 42.3 (2020). ISSN: 15584593. DOI: 10.1145/3417516. arXiv: 1909.03523.

[9] *Dagger 2*. URL: https://dagger.dev/.

[10] J. Darcy. *JSR 269: Pluggable Annotation Processing API*. URL: https://jcp.org/en/jsr/detail?id=269.

[11] O. Dardha et al. "Mungo and StMungo: Tools for typechecking protocols in Java". In: *Behavioural Types: from Theory to Tools English* (2017), pp. 309–328.

[12] R. DeLine and M. Fähndrich. "The Fugue protocol checker: Is your software Baroque?" In: *Submitted manuscript* January (2004). URL: http://research.microsoft.com/apps/pubs/default.aspx?id=67458%7B%5C%%7D5Cnhttp://research.microsoft.com/en-us/projects/fugue/.

[13] *Docker GitHub organization*. URL: https://github.com/docker (visited on 01/20/2021).

[14] J. Duarte. *Rusty Typestates*. 2020. URL: https://rustype.github.io/notes/notes/rust-typestate-series/rust-typestate-index.html.

[15] J. Duarte. *rustype/typestate-rs - commit 93cd8f3d2c*. URL: https://github.com/rustype/typestate-rs/tree/93cd8f3d2c6b0ecba75764d2c56eb2ad002c1569.

[16] *Effective Go*. URL: https://golang.org/doc/effective%7B%5C_%7Dgo.html (visited on 01/22/2021).

[17] T. Ekman and G. Hedin. "The JastAdd Extensible Java Compiler". In: 42.10 (2007), pp. 1–18. DOI: 10.1145/1297105.1297029.

[18] M. Fähndrich and R. DeLine. "Adoption and focus: Practical linear types for imperative programming". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 13–24. ISSN: 1558-1160. DOI: 10.1145/543552.512532.

[19] M. B. Feldman. *Who's Using Ada*. 2014. URL: https://www2.seas.gwu.edu/%7B~%7Dmfeldman/ada-project-summary.html (visited on 01/20/2021).

[20] N. Fitzgerald. *state_machine_future - Easily create type-safe 'Future's from state machines — without the boilerplate.* 2019. URL: https://github.com/fitzgen/state%7B%5C_%7Dmachine%7B%5C_%7Dfuture.

[21] T. L. Foundation. *Hyperledger Fabric*. (Visited on 01/15/2021).

[22] S. Gay and A. Ravara. *Behavioural types: From theory to tools english*. 2017, pp. 1–375. ISBN: 9788793519817. DOI: 10.13052/rp-9788793519817.

[23] S. J. Gay et al. "Modular session types for objects". In: *Logical Methods in Computer Science* 11.4 (2015), pp. 1–76. ISSN: 18605974. DOI: 10.2168/LMCS-11(4:12)2015.

[24] M. Heaney. *Why Ada isn't Popular*. 1998. URL: http://www.adapower.com/index.php?Command=Class%7B%5C&%7DClassID=Advocacy%7B%5C&%7DCID=39 (visited on 01/25/2020).

[25] *Heartbleed Bug*. 2021. URL: https://heartbleed.com/ (visited on 01/14/2021).

[26] G. Hoare. *rust-lang/rust - first commit*. 2010. URL: https://github.com/rust-lang/rust/commit/c01efc669f09508b55eced32d3c88702578a7c3e.

[27] T. Hoare. *Null References: The Billion Dollar Mistake*. 2009. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited on 01/25/2020).

[28] Hopcroft, Motwani, and Ullman. *Introduction to Automata Theory Languages, and Computation*. 3rd. Pearson, 2013. ISBN: 978-1292039053.

[29] J. Howarth. *Why Discord is switching from Go to Rust*. 2020. URL: https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f (visited on 01/20/2021).

[30] R. Hu, N. Yoshida, and K. Honda. "Session-based distributed programming in Java". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5142 LNCS (2008), pp. 516–541. ISSN: 03029743. DOI: 10.1007/978-3-540-70592-5_22.

[31] H. Hüttel et al. "Foundations of session types and behavioural contracts". In: *ACM Computing Surveys* 49.1 (2016), pp. 1–36. ISSN: 15577341. DOI: 10.1145/2873052.

[32] *inline-python - Inline Python code directly in your Rust code*. URL: https://github.com/fusion-engineering/inline-python (visited on 01/19/2021).

[33] *JastAdd*. URL: https://jastadd.cs.lth.se/web/.

[34] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. "Session types for rust". In: *WGP 2015 - Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, co-located with ICFP 2015* (2015), pp. 13–22. DOI: 10.1145/2808098.2808100.

[35] S. Klabnik. "The History of Rust". In: *Applicative 2016 on - Applicative 2016*. New York, New York, USA: ACM Press, 2016, p. 80. ISBN: 9781450344647. DOI: 10.1145/2959689.2960081. URL: http://dl.acm.org/citation.cfm?doid=2959689.2960081.

[36] W. Kokke. "Rusty Variation Deadlock-free Sessions with Failure in Rust". In: *Electronic Proceedings in Theoretical Computer Science, EPTCS* 304.Ice 2019 (2019), pp. 48–60. ISSN: 20752180. DOI: 10.4204/EPTCS.304.4.

[37] *Kotlin/kotlinx.serialization*. URL: https://github.com/Kotlin/kotlinx.serialization (visited on).

[38] D. Kouzapas et al. "Typechecking protocols with Mungo and StMungo: A session type toolchain for Java". In: *Science of Computer Programming* 155 (2018), pp. 52–75. ISSN: 01676423. DOI: 10.1016/j.scico.2017.10.006. URL: https://doi.org/10.1016/j.scico.2017.10.006.

[39] Koxiaet. *How to Write Hygienic Rust Macros*. 2020. URL: https://gist.github.com/Koxiaet/8c05ebd4e0e9347eb05f265dfb7252e1 (visited on 01/18/2021).

[40] *Kubernetes GitHub organization*. URL: https://github.com/kubernetes (visited on 01/20/2020).

[41] N. Lagaillardie, R. Neykova, and N. Yoshida. *Implementing multiparty session types in rust*. Vol. 12134 LNCS. Springer International Publishing, 2020, pp. 127–136. ISBN: 9783030500283. DOI: 10.1007/978-3-030-50029-0_8. URL: http://dx.doi.org/10.1007/978-3-030-50029-0%7B%5C_%7D8.

[42] X. Leroy et al. *The OCaml system release 4.11 - Documentation and user's manual*. 2020. URL: https://caml.inria.fr/pub/docs/manual-ocaml/ (visited on 02/02/2021).

[43] S. Mageit. *NHS COVID-19 contact tracing app fails to ask users to self-isolate*. 2020. URL: https://www.healthcareitnews.com/news/emea/nhs-covid-19-contact-tracing-app-fails-ask-users-self-isolate.

[44] *Memory safety - The Chromium Projects*. URL: https://www.chromium.org/Home/chromium-security/memory-safety.

[45] (. Miller, Matt. *Trends, challenges and strategic shifts in the software vulnerability mitigation landscape*. 2019. URL: https://git.io/JLdDc.

[46] K. Most. *Writing Your First Kotlin Compiler Plugin*. 2018. URL: https://resources.jetbrains.com/storage/products/kotlinconf2018/slides/5%7B%5C_%7DWriting%20Your%20First%20Kotlin%20Compiler%20Plugin.pdf (visited on 02/04/2021).

[47] J. Mota. "Coping with the reality : adding crucial features to a typestate-oriented language". PhD thesis. FCT-NOVA, 2020.

[48] P. Munksgard. *session-types*. URL: https://github.com/Munksgaard/session-types (visited on 01/20/2021).

[49] S. O'Kane. *Boeing finds another software problem on the 737 Max*. 2020. URL: https://www.theverge.com/2020/2/6/21126364/boeing-737-max-software-glitch-flaw-problem.

[50] S. O'Kane. *Southwest and United Airlines extend 737 Max cancellations after new flaw found*. 2019. URL: https://www.theverge.com/2019/6/27/18761434/737-max-new-flaw-cancellation-extended-southwest-united-airlines-faa.

[51] T. Palmer. *tjpalmer/languish*. URL: https://tjpalmer.github.io/languish/ (visited on 01/25/2021).

[52] R. Pucella and J. A. Tov. "Haskell session types with (almost) no class". In: *Haskell'08 - Proceedings of the ACM SIGPLAN 2008 Haskell Symposium* (2008), pp. 25–36. DOI: 10.1145/1411286.1411290.

[53] *PureScript*. 2021. URL: https://www.purescript.org/ (visited on 01/14/2021).

[54] D. de Rauglaudre. *Camlp4 - Reference Manual*. 2003. URL: https://caml.inria.fr/pub/docs/manual-camlp4/manual007.html.

[55] *Reason · Reason lets you write simple, fast and quality type safe code while leveraging both the JavaScript & OCaml ecosystems*. URL: https://reasonml.github.io/.

[56] N. Rebours. *An introduction to OCaml PPX ecosystem*. 2019. URL: https://tarides.com/blog/2019-05-09-an-introduction-to-ocaml-ppx-ecosystem.

[57] *Rocket Web Framework*. 2021. URL: https://rocket.rs/ (visited on 01/19/2021).

[58] *ruby/rbs*. 2021. URL: https://github.com/ruby/rbs (visited on 01/14/2021).

[59] *SIGAda*. URL: https://www.sigada.org/ (visited on 01/20/2021).

[60] Square. *square/javapoet*. URL: https://github.com/square/javapoet.

[61] *stm32-rs*. 2021. URL: https://github.com/stm32-rs/.

[62] B. Stokke. *typed-html - Type checked JSX for Rust*. URL: https://github.com/bodil/typed-html (visited on 01/19/2021).

[63] R. E. Strom et al. *Hermes : A Tutorial and Reference Manual*. 1990. URL: https://researcher.watson.ibm.com/researcher/files/us-bacon/Strom90HermesTutorial.pdf.

[64] R. E. Strom. "Mechanisms for compile-time enforcement of security". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83*. New York, New York, USA: ACM Press, 1983, pp. 276–284. ISBN: 0897910907. DOI: 10.1145/567067.567093. URL: http://portal.acm.org/citation.cfm?doid=567067.567093.

[65] B. Stroustrup. *The C ++ Programming*. 1986, p. 1346. ISBN: 978-0321563842.

[66] *The Ada Programming Language*. URL: https://web.archive.org/web/20160522063844/http://groups.engin.umd.umich.edu/CIS/course.des/cis400/ada/ada.html (visited on 01/20/2021).

[67] *The Embedded Rust Book*. 2021. URL: https://rust-embedded.github.io/book/.

[68] *The Rust Programming Language*. 2021. URL: https://doc.rust-lang.org/book/ (visited on 01/18/2021).

[69] *The Rust Reference*. 2021. URL: https://doc.rust-lang.org/reference (visited on 01/18/2021).

[70] C. Torre et al. *Panel: Systems Programming in 2014 and Beyond*. 2014. URL: https://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2014/Panel-Systems-Programming-Languages-in-2014-and-Beyond.

[71] *Typed JavaScript at Any Scale*. URL: https://www.typescriptlang.org/.

[72] *typing — Support for type hints — Python 3.9.1 documentation*. URL: https://docs.python.org/3/library/typing.html (visited on 01/14/2021).

[73] C. Vasconcelos and A. Ravara. "From object-oriented code with assertions to behavioural types". In: *Proceedings of the ACM Symposium on Applied Computing* Part F1280 (2017), pp. 1492–1497. DOI: 10.1145/3019612.3019733.

[74] V. T. Vasconcelos, S. J. Gay, and A. Ravara. "Type checking a multithreaded functional language with session types". In: *Theoretical Computer Science* 368.1-2 (2006), pp. 64–87. ISSN: 03043975. DOI: 10.1016/j.tcs.2006.06.028.

[75] Vicompany. *Coinbase disclosed on HackerOne: Ethereum account balance manipulation*. 2018. URL: https://hackerone.com/reports/300748.

[76] A. L. Voinea, O. Dardha, and S. J. Gay. "Typechecking java protocols with [st]mungo". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12136 LNCS (2020), pp. 208–224. ISSN: 16113349. DOI: 10.1007/978-3-030-50086-3_12.

[77] Whitequark. *A Guide to Extension Points in OCaml*. 2014. URL: https://whitequark.org/blog/2014/04/16/a-guide-to-extension-points-in-ocaml/ (visited on 01/14/2021).

[78] L. Wirth. *The Little Book of Rust Macros*. 2021. URL: https://veykril.github.io/tlborm.

[79] C. Wong. *maud - Compile-time HTML templates for Rust*. URL: https://github.com/lambda-fairy/maud (visited on 01/19/2021).

[80] H. Xi and H. Wu. "Linearly Typed Dyadic Group Sessions for Building Multiparty Sessions". In: (2016). arXiv: 1604.03020.

[81] H. Xi et al. "Session Types in a Linearly Typed Multi-Threaded Lambda-Calculus". In: (2016). arXiv: 1603.03727.

[82] N. Yoshida et al. "The scribble protocol language". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8358 LNCS.October 2007 (2014), pp. 22–41. ISSN: 16113349. DOI: 10.1007/978-3-319-05119-2_3.