José Miguel Gonçalves Duarte

Bachelor in Computer Science

Bridging the Gap between Typestates and Rust in Production Software

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

Master of Science in
Computer Science

Adviser: António Ravara
Full Professor, NOVA University Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2021

Bridging the Gap between Typestates and Rust in Production Software

To my grandparents, Amélia, Olga and José.

Acknowledgements

This work is partially supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT.IP.

The present work is the culmination of several years of effort. As one does not live in a bubble, several people have provided their help and support and I want to thank them.

I thank the NOVA organization and everyone that directly or indirectly contributed to my education.

I thank everyone that helped me revise my academic work, or has contributed with improvements to this thesis; to that end I thank Antoine Martin, Bernardo Toninho, Daniel Henry-Mantilla, João Mota, Katherine Flavel, Mathias Jakobsen, Ornela Dardha, Philip Munksgaard, Sabrina Jewson, Simon Fowler and Wen Kokke.

I would like to thank my advisor, António Ravara, I could not have asked for someone better to guide me through the thesis, being always available to provide any necessary assistance I required; this thesis would have not been possible without him.

I thank my parents Cristina and José, my brother Pedro and my grandparents Amélia, Olga and José, for their unconditional support.

Along with my family, I must thank my friends which have also supported me through thick and thin, despite all the obstacles that our friendship has faced. My friends André Rodrigues, Diogo Pereira, João Antão, João Teixeira, Li Zixiang, Pedro Feiteira and Tiago Ventura, which have been there since the first days and provided me with great memories both inside and outside college. My friends Adriano Batista, Bruno Anjos, Diogo Cebola, Gonçalo Lopes, João Carvalho, Marco Monteiro, Miguel Carrega and Ricardo Leitão, which I made along the way and have helped me throughout college both posing questions and solving many of mine. Last, but not least, all the friends I met outside of college which have played an equally important role in my life, staying by my side and not letting anything get in the way of our friendship, Catarina Guerreiro, Génesis Conceição, Joana Godinho, Jorge Catarino, Margarida Gama, Miguel Braz and Tomás Alagoa.

I am proud to carry with me all the memories and lessons learned from each one of you; I would not be the person I am today without you, and I will be forever in debt with you.

"... my motto was to know that the most powerful force is to be interested in something. (...) whatever it is, if you are interested enough to study and deepen it, then you are not in danger. If you stick to something – it can be archeology, music, sports – that is bigger than yourself, you are not in danger. The terrible thing is when people cling to nothing, to the void.
" — George Steiner

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 Application Programming Interface (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, and 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.

I propose a new macro-based approach to deal with typestates in Rust; this approach provides an embedded Domain-Specific Language (DSL) which allows developers to express typestates using only existing Rust syntax. Furthermore, Rust's macro system is leveraged to extract a state machine out of the typestate specification and then perform compile-time checks over the specification. Afterwards we leverage Rust's type system to check protocol-compliance. The DSL avoids workflow-bloat by requiring nothing but a Rust compiler and the library itself.

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

Resumo

À medida que as nossas vidas estão cada vez mais dependentes de software, os 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, mas 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 conheça a utilização dos mesmos e detecte 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, o que resulta 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*.

Eu proponho uma nova abordagem de modo a lidar com *typestates* em Rust, tal abordagem é baseada numa DSL embebida na linguagem, permitindo assim a descrição de *typestates* usando apenas a sintaxe existente. A DSL vai mais além e providencia ainda verificações estáticas sobre a especificação, tirando proveito do sistema de macros, extrai uma máquina de estados que é depois verificada, por fim, a verificação de conformidade é feita pelo compilador, tirando proveito do sistema de tipos. A DSL evita poluição do ambiente trabalho, requerendo apenas um compilador de Rust e a sua própria biblioteca.

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

Contents

List of Figures xix

List of Tables xxi

Glossary xxiii

Acronyms xxv

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem . . . 2
  - 1.2.1 The Billion Dollar Mistake . . . 2
  - 1.2.2 API Misuse . . . 3
- 1.3 State of the Art . . . 4
- 1.4 Contributions . . . 5
- 1.5 Outline . . . 6

2 Background 7
- 2.1 Systems Programming Languages . . . 7
  - 2.1.1 C . . . 7
  - 2.1.2 C++ . . . 8
  - 2.1.3 Ada . . . 8
  - 2.1.4 Go . . . 9
  - 2.1.5 Summary . . . 10
- 2.2 The Rust Language . . . 10
  - 2.2.1 What makes Rust different? . . . 10
  - 2.2.2 Ownership . . . 10
  - 2.2.3 Borrowing . . . 11
  - 2.2.4 Concurrency . . . 13
  - 2.2.5 Why Rust instead of Language X? . . . 14

2.3 Behavioral Types . . . 14
2.3.1 Session Types . . . 15
2.3.2 Typestates . . . 17

3 Related Work 23
3.1 Language Preprocessors . . . 23
3.1.1 OCaml . . . 23
3.1.2 Java . . . 25
3.1.3 Kotlin . . . 28
3.2 Rust Macros . . . 29
3.2.1 Declarative Macros . . . 30
3.2.2 Procedural Macros . . . 32
3.2.3 Summary . . . 34
3.3 Approaches to Behavioral Types . . . 34
3.3.1 Session Types . . . 35
3.3.2 Typestate . . . 36
3.3.3 Summary . . . 38

4 The `#[typestate]` macro 39
4.1 Typestates: The Hard Way . . . 39
4.1.1 Future Proofing . . . 44
4.2 Typestates: The DSL . . . 44
4.2.1 Syntax & Automaton Extraction . . . 44
4.2.2 Architecture . . . 50
4.2.3 Advanced Features . . . 51
4.3 Validation . . . 55
4.3.1 The Automaton and the Graph . . . 55
4.3.2 Implementing a validation strategy . . . 56
4.4 Visualizing Typestates . . . 57
4.4.1 Debugging with Visualizations . . . 57
4.4.2 Embedding Visualizations in the Documentation . . . 58
4.5 Summary . . . 61

5 Case Studies 65
5.1 Ring . . . 65
5.1.1 Comparison . . . 66
5.1.2 Summary . . . 68
5.2 PIN . . . 68
5.3 Auction Client . . . 73
5.4 Summary . . . 75

6 Conclusions & Future Work 81

6.1 Summary 81
6.2 Future Work 82

Bibliography **83**

List of Figures

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. 15
2.2 Session type example, equivalent to Listing 2.5. 16
2.3 The Scanner typestate automata, based on Listing 1.3. 17

3.1 Java's annotation processor lifecycle. 26
3.2 java.lang.annotation class diagram. 26
3.3 Kotlin compiler plugin architecture stack. 29
3.4 Rust macro's family tree 29

4.1 Vending machine automaton. 40
4.2 From DSL specification to Rust code. First the DSL is parsed, an intermediate graph representing the automaton in more general terms is extracted from the AST, from the graph the macro will convert the user can generate visualizations (for debugging or documentation), this step is optional. . . 50
4.3 The #[typestate] macro visitors, by running order. 51
4.4 The vending machine's DOT typestate, rendered using the command — dot -Tsvg VendingMachine.dot. 59
4.5 The vending machine's PlantUML typestate, rendered using the command — plantuml -tsvg VendingMachine.uml. 60
4.6 The vending machine API documentation page. Result of Listing 4.32 when rendered using cargo doc. 62
4.7 The original configuration, the macro depends only on typestate-deps and does not export any dependency. 62
4.8 The naive attempt, the macro depends on typestate-deps and aquamarine, but it only tries to export aquamarine, this fails because typestate is a procedural macro crate. 62

4.9 The macro was isolated in its own crate — `typestate-proc-macro`, which depends on `typestate-deps`; the `typestate` crate now depends and exports both `aquamarine` and `typestate-proc-macro`. 63

5.1 The ring example visualization. 65
5.2 The ring participants' typestates. 72
5.3 The `Reader` and `Card` typestates, the green arrows indicate the dependency relationship between states. 73
5.4 The auction client's typestate. 78

List of Tables

3.1 Rust macros properties summary. 34

4.1 Overview of the DSL's annotations. 49
4.2 Overview of the transition inference rules. 50
4.3 All configuration parameters for the DOT and PlantUML visualization features. 59

Glossary

Acronyms

API	Application Programming Interface xi, xix, 3, 6, 17, 27, 35, 36, 42, 43, 44, 46, 58, 62, 66
AST	Abstract Syntax Tree 50, 51
Camlp4	Pre-Processor-Pretty-Printer 23, 24
DFS	Depth-First Search 56
DOA	Deterministic Object Automata 42, 52
DSL	Domain-Specific Language xi, xiii, 5, 6, 66
JVM	Java Virtual Machine 3, 28
MPST	Multi-Party Session Types 65
PPX	PreProcessor eXtensions 23, 24, 25

1 Introduction

1.1 Context

Bugs permeate our lives as users — whether in an instant messaging application or a game, they are present. Luckily, since most of these applications are not critical, their impact is minimal, resulting, at worst, 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 in extreme cases, death. In 2014, the Heartbleed[1] 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[2]. In 2019 and 2020, after several crashes[3], the Boeing 737 Max was grounded to fix existing problems. While grounded, more software-related issues were found[4],[5], 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[6].

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 try and mitigate them — building tools and abstractions which allow developers to increase their code's safety.

[1] `https://heartbleed.com/` (visited in 14/01/2021)
[2] `https://hackerone.com/reports/300748` (visited in 08/06/2021)
[3] `https://tinyurl.com/DCampbell2020` (visited in 08/06/2021)
[4] `https://tinyurl.com/Okane2019` (visited in 06/08/2021)
[5] `https://tinyurl.com/Okane2020` (visited in 08/06/2021)
[6] `https://tinyurl.com/Mageit2020` (visited in 08/06/2021)

```
1  class Main {
2      public static void main(String... args) {
3          Integer a = null;
4          a + 5; // NullPointerException: `a` is `null'
5      }
6  }
```

Listing 1.1: A `null` reference in Java.

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[7] and Microsoft products[8].

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 to behavioral types 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. — Tony Hoare*[9]

Consider Listing 1.1; the program compiles but it will crash with a `NullPointerException` on line 4. While anyone 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

[7] https://www.chromium.org/Home/chromium-security/memory-safety (visited in 08/06/2021)

[8] https://git.io/JLdDc (visited in 08/06/2021)

[9] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited in 01/07/2021)

```java
class ScannerMisuse {
    public static void main(String... args) {
        Scanner s = new Scanner(System.in);         // open the stream
        s.nextLine();                               // read
        s.close();                                  // close the stream
        s.nextLine();                               // IllegalStateException
    }
}
```

Listing 1.2: Java's Scanner misuse example.

"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.

1.2.2 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.

An example of a similar bug would be Jedis' issue #1747[10]; while reading a reply from the server, a client would get a SocketTimeoutException[11], the exception then calls jedis.close(), however, processing of the underlying would (wrongly) keep going, in some cases, the processing procedure would heap-allocate a *very* large array or a negative-space, thus causing the Java Virtual Machine (JVM) to crash with an OutOfMemoryException[12].

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?*

[10] https://github.com/redis/jedis/issues/1747 (visited in 14/07/2021)
[11] https://docs.oracle.com/javase/8/docs/api/java/net/SocketTimeoutException.html (visited in 14/07/2021)
[12] https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html (visited in 14/07/2021)

```java
class TypestatedScanner {
    public static void main(String... args) {
        Scanner[Open] s = new Scanner(System.in);   // open the stream
        s.nextLine();                               // read
        Scanner[Closed] s = s.close();              // close the stream
        s.nextLine();                               // compile-time error
    }
}
```

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

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 approaches to behavioral types — session types and typestates.

Session types will often refer to endpoints and their messages, capturing aspects of communication between them [14]. Languages like ATS provide session type features and further enable the generation of source code in other languages such as Erlang [42]. ATS also serves as research playground for other topics related with session types [41]. 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], [12, Chapter 10] and OCaml [12, 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 [43] to capture an algorithm's communication pattern, 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], [16], a Java extension which is an implementation of Moose [2, Section 2.1.1]. Another session type enabling project is Mungo & StMungo [21, 39], 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 [31]. I discuss typestates further in Section 2.3.2. Fugue [9] 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 [5, 6] 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, it is considered to be a "done work" and the authors have moved on to other projects[13]. Obsidian is a relatively new language which targets the Hyperledger Fabric blockchain, in the development phase. 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 [5] the effectiveness of the approach is put to test, achieving positive results when compared with the Solidity programming language.

The `state_machine_future` crate[14], 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 [18] introduces bi-directional session types to Rust, since then, this line of work has been expanded by [23], 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 Contributions

In this thesis I present a DSL which enables developers to easily declare typestates in their Rust code. Its major features are:

Static guarantees. Alongside the guarantees provided by the type system, our macro leverages the DSL to build an automata model which is then checked for state usefulness.

Ease of use. The DSL leverages Rust existing syntax rather than expand it. The usage of macros allows the DSL to tweak the semantics of some language constructs, making them more flexible for our purposes.

[13]`https://www.cs.cmu.edu/~aldrich/plaid/` (visited in 14/07/2021)
[14]`https://github.com/fitzgen/state_machine_future` (visited in 08/06/2021)

Ease of use (Part II). The DSL does not require any extra tools, declaring the macro as a dependency is the only required step for the user to start using the DSL.

Enhanced documentation. The DSL is able to render visualizations of the typestates' automata, these can either be exported or embedded into the final client's API documentation.

To start using the DSL the user can simply add the following line to the `Cargo.toml` file.

```
typestate = "0.8"
```

1.5 Outline

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 presents the core of the developed work: the `#[typestate]` macro. It provides an introduction to typestates in Rust (Section 4.1), a discussion over its DSL (Section 4.2) along with insights into the validation (Section 4.3) and visualization processes (Section 4.4).

Chapter 5 presents a three different case studies, developed using the `#[typestate]` macro. Each case highlights a different capability of the macro.

Chapter 6 presents a summary and discusses future work.

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 [35].

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 a single computer to a distributed system, interfacing with the hardware in a more direct fashion is mostly not required. Systems programming languages emphasized being able to produce a standalone binary able to run on a variety of machines without requiring extra software.

The subjects of this analysis have been picked due to their relevance in the area of systems programming, being present in indices such as TIOBE[1], or developer surveys such as StackOverflow's[2] and JetBrains'[3].

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

[1] `https://www.tiobe.com/tiobe-index/` (visited in 14/07/2021)

[2] `https://insights.stackoverflow.com/survey/2020#most-popular-technologies` (visited in 14/07/2021)

[3] `https://www.jetbrains.com/lp/devecosystem-2020/` (visited in 14/07/2021)

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 with 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.*
>
> — *[32, 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 [35]. 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 to be solved, as they still introduce overhead in the most demanding applications[4].

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[5].

[4] https://www.youtube.com/watch?v=rHIkrotSwcc (visited in 01/07/2021)
[5] https://tinyurl.com/AdaPL2016 (visited in 08/06/2021)

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[6]. The language is also used in several other domains, such as aviation, space vehicles, financial systems and more[7].

In comparison with the other languages in this section, Ada is eclipsed, barely showing in the GitHub rankings[8]. However, given that Ada's compiler requires a paid license to take full advantage of and their main application market are critical applications, 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[9] 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 perpetuate 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 to 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 through the usage of 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 and Kubernetes, Go's categorization as a systems programming language can be discussed.

When put against its peers in the systems programming language ecosystem, Go's performance may be considered *lacking*, not being enough for certain use cases. This was Discord's case, 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[10]. In [35], one of Go's authors, Robert 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.

[6] `https://www.sigada.org/` (visited in 08/06/2021)

[7] `https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html` (visited in 08/06/2021)

[8] `https://tjpalmer.github.io/languish/` (visited in 25/01/2021)

[9] `https://tinyurl.com/AdaPower1998` (visited in 08/06/2021)

[10] `https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f` (visited in 08/06/2021)

2.1.5 Summary

In this section 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, non-nullable types by default and even some theorem proving capabilities, being the only one which does not suffer from the *"billion dollar mistake"*[11].

2.2 The Rust Language

Rust is a fairly recent systems programming language, it started as a side project of Graydon Hoare and its public history dates back to 2010[12]. 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 [19].

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 neither safety nor performance.

The key to all the promises Rust makes is its 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 its biggest defect, 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

[11] `https://tinyurl.com/Hoare2009` (visited in 25/01/2021)
[12] `https://git.io/JZ3X7` (visited in 08/06/2021)

```
1 fn main() {
2     let x = String::from("Hello"); // ok: `x` is assigned "Hello"
3     let y = x;                     // ok: `x` is moved into `y`
4     println!("{}", x);             // error: `x` was moved in the previous line
5 }
```

Listing 2.1: Example of the move-by-default mechanism to enforce ownership. Interactive example (01/07/2021): https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=44beb8b69a296943e9a1c72de4d50ac3

when required, inserting the respective deallocation procedures 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.*

— *[33, Section 4.1]*

To illustrate the rules, consider Listing 2.1, where we have two variables x and y. First, "Hello" is assigned to x (line 2), thus x now owns "Hello". After, x is assigned to y (line 3), 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 on line 4, 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 implicitly copied instead of moved.

So far, this example illustrates the first two rules. The last rule can be considered "invisible", as it happens during compilation and the user will not notice it usually. What happens is that at the end of the scope, any variable whose owner is in scope, will be freed (in Rust's terms, it will be dropped). 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.

```rust
fn main() {
    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. Interactive example (01/07/2021): https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=129727839135fcaa4651bf764242feda

```rust
fn main() {
    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. Interactive example (01/07/2021): https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=aeef7b0822bb73037b8f99ee8413d834

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:

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, on lines 3 & 4) are only read and the third (r3, on line 5) 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.

```rust
fn main() {
    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. Interactive example (01/07/2021): https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=ea8ff872605239f5916753d32d7145b7

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. — [33, 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[13] is inspired by 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. — Effective Go[14]

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.

[13]https://doc.rust-lang.org/std/sync/mpsc/ (visited in 01/07/2021)
[14]https://golang.org/doc/effective_go.html (visited in 08/06/2021)

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 minimize possible runtime overhead while providing an expressive language for typestate specification.

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 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 [17]. 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.

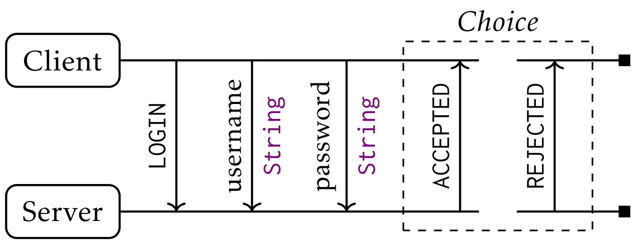


Figure 2.1: Communication protocol example. The communication establishment step is omitted for simplicity. In this protocol the client tries to login to a service by sending a message `LOGIN` followed by the username and password, both of type `String`. The server then replies with either an `ACCEPTED`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

> *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 them. 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" [14, 15]. 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 [13, 38].

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

```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.

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

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

complexity, as 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 [38], where the first four assignments (:=) are simple aliases, to simplify reading):

Consider ! to be *sends*, ? to be *receives*, . the *sequence* operator, & the *choice offering* and ⊕ 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

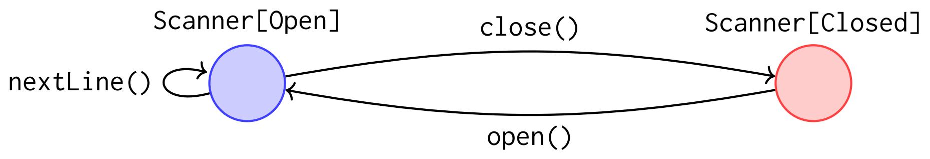


Figure 2.3: The `Scanner` typestate automata, based on Listing 1.3.

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.

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. — [31]*

The first language to make use of typestates was NIL [31], afterwards languages like Hermes [30] 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 typestate's 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 and 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 away from dynamically typed languages, to statically typed ones, or at the very least, add typing support to existing dynamic languages. Typescript[15], Reason[16] and PureScript[17] 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[18,19].

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 [39], 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[20]. Typestate-related concepts were also used in Singularity OS [2, Section 6], a reliable operating system prototype where programs were written using Sing# — a C# derived language 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.

[15]https://www.typescriptlang.org/ (visited in 08/06/2021)
[16]https://reasonml.github.io/ (visited in 08/06/2021)
[17]https://www.purescript.org/ (visited in 08/06/2021)
[18]https://docs.python.org/3/library/typing.html (visited in 08/06/2021)
[19]https://github.com/ruby/rbs (visited in 08/06/2021)
[20]https://git.io/JZ3i7 (visited in 08/06/2021)

```
1  public class Read {
2      public static void main(String[] args) {
3          Scanner s = new Scanner(System.in);
4          s.nextLine();
5          s.close();
6          s.nextLine();
7      }
8  }
```

Listing 2.6: The Read Java program, which reads two lines from stdin.

```
1  public class Read {
2      public static void main(String[] args) {
3          Scanner[Open] s = new Scanner(System.in);
4          s.nextLine();
5          Scanner[Closed] s = s.close();
6          s.nextLine(); // compiler error
7      }
8  }
```

Listing 2.7: The Read program, written in a Java-like *typestated* fashion.

Typestates in action

As a simple example, consider the Java application in Listing 2.6 which simply which reads two lines from stdin. 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 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 Read 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 classes users write typestates. 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).

```
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 [26]).

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

Listing 2.9: Obsidian state declaration example.

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

In [5] 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 2.9, 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 2.10 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 2.11, 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.

[21] `https://www.hyperledger.org/use/fabric` (visited in 08/06/2021)

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

Listing 2.10: 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 2.11: Invalid state transition example in Obsidian. Since `LightSwitch` is instantiated as `Off`, calling `turnOff` is not valid.

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

Listing 2.12: 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.

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, 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 the language. Consider Listing 2.12, the example is expected to call first `F1`, followed by `F2` and finally `F3`, however such does not happen and the error is only caught at runtime.

As the next paragraph discusses, this behavior can be prevented using the language's type system. However, such utilization requires complex 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[22] Chapter 4 is dedicated to static guarantees, introducing programmers to the concepts of typestate in its Section 4.1[23], and their usage in embedded

[22] `https://rust-embedded.github.io/book/` (visited in 08/06/2021)

[23] `https://docs.rust-embedded.org/book/static-guarantees/typestate-programming.html` (visited in 01/07/2021)

systems.

As for real-world usage, typestates are abundantly used in the area (not just discussed in the book), under the `stm32-rs` GitHub organization[24] one finds several repositories (suffixed with `-hal`) which implement typestates (e.g. `gpio.rs` from `stm32h7xx-hal`).

[24] `https://github.com/stm32-rs/` (visited in 08/06/2021)

3 Related Work

In this chapter I review the work related to this thesis. I start by reviewing a series of existing tools designed to help with code preprocessing, the languages considered were OCaml, Java and Kotlin, given the existing documentation, and academic work over them, projects like OCaml's PreProcessor eXtensions (PPX), Java's JastAdd [10] and Kotlin's compiler plugins; followed by a more detailed discussion w.r.t. Rust; finally I review previous work regarding behavioral types, discussing both session types and typestates.

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, previous to version 4.02, OCaml made use of Pre-Processor-Pretty-Printer (Camlp4).

Camlp4 is a parsing library providing users with extensible grammars which enable users to modify and extend OCaml's syntax, Camlp4 is also able to redefine the core syntax, OCaml even introduced a revised syntax[1] to enable Camlp4.

In a nutshell, the Camlp4 library would allow developers to develop an extension syntax, the compiler would then 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[2].

[1] `https://caml.inria.fr/pub/docs/manual-camlp4/manual007.html` (visited in 02/07/2021)
[2] `https://tinyurl.com/Whitequark2014` (visited in 08/06/2021)

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

Listing 3.1: Example of the three kinds of attributes[3]. 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.

PPX

In OCaml version 4.02 syntax extensions were introduced, to enable preprocessor extensions. This meta programming mechanism came to replace Camlp4, which was not well liked by the community given its complexity. The resources on PPX are not as widespread as the resources for similar mechanisms in other languages (e.g. Rust's macros). There are two main entry points to the PPX system, attribute and extension nodes [24, 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.*

— *[24, Section 8.12]*

One of the main kinds of PPXs are *derivers* (see Listing 3.2.2 for the Rust equivalent). Derivers are mostly used to generate error-prone code where the implementation pattern is common to a series of situations. Examples include but are not limited to: comparison functions, pretty printers and serializers[3].

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.*

— *[24, Section 8.13]*

[3] https://tarides.com/blog/2019-05-09-an-introduction-to-ocaml-ppx-ecosystem (visited in 08/06/2021).

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
@interface Foo {}
```

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

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.

3.1.2 Java

In Java, meta programming takes the form of annotations, these can be processed by user code during the compilation process or at runtime. Besides annotations, there is another project able to "extend" Java. The ExtendJ research compiler (formerly JastAddJ) [10] 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[4], 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[5]. 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 3.2. 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`

[4]https://jcp.org/en/jsr/detail?id=269 (visited in 08/06/2021)
[5]https://dagger.dev/ (visited in 08/06/2021)
[6]https://checkerframework.org/ (visited in 02/07/2021)

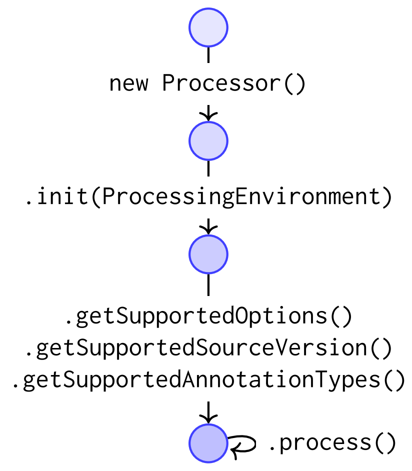

new Processor()
.init(ProcessingEnvironment)
.getSupportedOptions()
.getSupportedSourceVersion()
.getSupportedAnnotationTypes()
.process()

Figure 3.1: Java's annotation processor lifecycle.

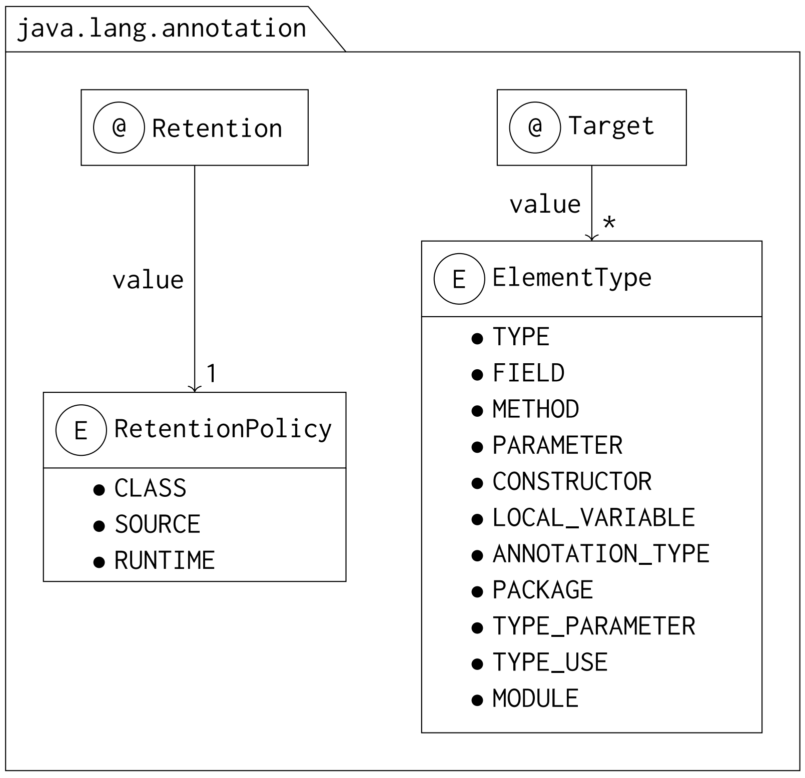


Figure 3.2: java.lang.annotation class diagram.

- `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 will 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[7]. 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[8], 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.

[7]`https://github.com/square/javapoet` (visited in 02/07/2021)
[8]`https://jastadd.cs.lth.se/web/` (visited in 08/06/2021)

```java
@any Animal animals;
animals += new Animal("Magali");
animals += new Animal("Snow");
animals += new Animal("Coal");
for (Animal animal : animals) {
    System.out.println(animal.getName());
}
```

Listing 3.3: The `@any` annotation allows an object to carry several instances of itself. In the example, `@any Animal` is rather a collection of `Animal`. 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 to 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 JVM. 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 3.3. 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.

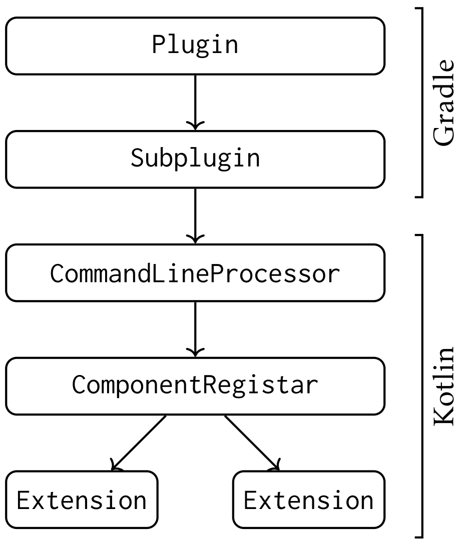


Figure 3.3: Kotlin compiler plugin architecture stack[10].

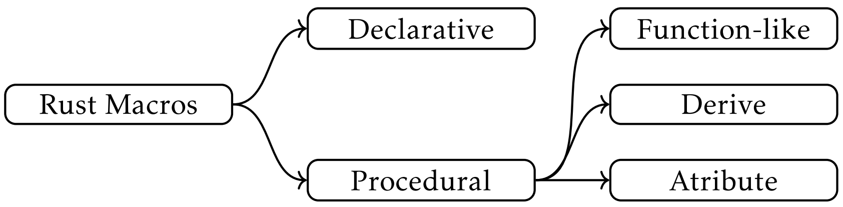


Figure 3.4: Rust macro's family tree

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 tools depending on compiler plugins are widely used (e.g. Kotlin serialization[9]), 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 3.4), *declarative* macros and *procedural* macros.

[9]`https://github.com/Kotlin/kotlinx.serialization` (visited in 08/06/2021)
[10]`https://tinyurl.com/MostK2018` (visited on 08/06/2021)

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. — [34, 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 [34, 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 [34, Section 3.1 - Metavariables]. In Listing 3.4, the kind of the `n` metavariable is `literal`, which will match literals like `'E'`, `"Elite"` and `420` [34, 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 [40]. 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[11] 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

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

Listing 3.4: Example macro_rules! usage. When executed, the code above will print “Hello, world!” five times.

```rust
1  macro_rules! using_a {
2      ($e:expr) => {
3          {
4              let a = 42;
5              $e
6          }
7      }
8  }
9  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.

```rust
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.

Listing 3.7), since line’s 3 a will not exist due to not being in the same syntactic context as line 2.

[11]The same mechanism does not apply to procedural macros since they are not hygienic. Their output will interfere with existing code if precautions are not taken [22].

```
1  error[E0425]: cannot find value `a` in this scope
2    --> src/main.rs:13:21
3     |
4     | 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 as in line 2 and 3 are considered to belong to different contexts.

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

Listing 3.8: HTML DSL embedded in Rust.[15]

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.

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[12,13] (see the example in Listing 3.8) and the possibility to run Python inside Rust[14].

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`.

[12]https://github.com/lambda-fairy/maud (visited in 08/06/2021)
[13]https://github.com/bodil/typed-html (visited in 08/06/2021)
[14]https://github.com/fusion-engineering/inline-python (visited in 08/06/2021)
[15]https://github.com/lambda-fairy/maud (visited in 08/06/2021)

```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.

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 [34, Section 6]), including items in `extern` blocks, inherent and trait implementations, and trait definitions.

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).

```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[16] the user declares that GET requests to hello/ have two path parameters (name and age) and should be handled by the hello function.

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.

3.2.3 Summary

In summary, metaprogramming in Rust is enabled by macros, these can be divided into two categories, declarative macros and procedural macros. Their main characteristics are summarized in Table 3.1.

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

[16] https://rocket.rs/ (visited in 08/06/2021)

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 [8, 21, 39] and others [13, 37] demonstrate.

StMungo it is a transpiler from Scribble [43] 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 [8, 21, 39]. 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.

A Session Type Provider. The work by Neykova et al. [27] leverages F#'s type providers to provide developers with practical session types along with refinements. The refinements enable the placement of constraints regarding the protocol's messages, these are described using Scribble [43] along with the rest of the protocol; the protocol and its respective refinements are validated and the .NET platform generates the required code for the F# APIs.

Furthermore, this approach allows users to still use features like autocomplete (something that is still being worked in Rust's ecosystem) and documentation. This approach also reduces bugs and simplifies several error-prone parts of development.

Session Types for Rust. As far as I am aware, the work on Rust session types was started by Jespersen et al. [18], such work was limited as it only supported binary session types. It builds on a Haskell-based approach [28], 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 channel's protocols do not match.

The `sesh` crate [20], in contrast to `session-types` [18], builds on a different theory, provides much cleaner types and embraces Rust's affine type system, instead of actively trying to make it linear. While the crate requires no use of `unsafe`, it does require a *nightly* compiler, unsuited for production environments.

Like `session-types`, the sesh crate provides the required abstractions to use session types in Rust, however, the crate's documentation is limited, possibly being "inaccessible" to less experienced users w.r.t. Rust and session types, along with these factors, the crate is not adequate for production due to the requirement of a nightly compiler.

Multiparty Session Types for Rust. Work on multiparty session types started with Lagaillardie et al. [23]. This work makes use of the Scribble [43] toolchain, just like StMungo; and it is a thin wrapper over previous work done by Kokke [20]. Similarly to the previously presented work [18], 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 [18, 20].

Rumpsteak [7] is a Rust library targeting asynchronous applications using `async`/`await`, it offers clean session types and its approach relies heavily on macros. The crate works by defining a global type which is projected into roles, this is done by Scribble which in turn requires an external toolchain[17]; from each role an endpoint finite state machine is extracted and optimized, in the end an API is generated which can then be used to build each process.

To enforce linear resource usage when build the processes the library makes use of the type system to enforce the consumption of each "state"; enforcing protocol completion is done through a closure, which takes the initial session type and returns a terminal `End` type, a session is then required to be run until completion otherwise the return type will not be respected and the type checker will complain.

Like `sesh` this crate is still in "research mode", however its documentation is not incomplete, but rather non-existent. This raises several difficulties when trying to evaluate the library as the only existing documentation is the paper [7], which does not cover all the library's details.

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 [9] 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 [11]. 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.

[17] `https://github.com/nuscr/nuscr` (visited in 08/06/2021)

```
[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.

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 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 [9]. 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.

```
@Typestate("StateIteratorProtocol")
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`.

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 [8, 21, 39] for Java programs. It checks typestate properties and can be divided into two components, a Java-like language to define typestate 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. — [8, 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. Possibly, the main factor working against newer academic ideas is that they are *bleeding-edge* research, making companies wary of their application, preferring to keep their old but tested development methods, instead of investing on newer and untested ones. 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.

4 The #[typestate] macro

This chapter presents the contributions of the present work, the `#[typestate]` macro. In Section 4.1, I start by demonstrating how to implement Rust typestates by hand. In Section 4.2.2 I discuss the macro high-level architecture, the DSL is discussed in Section 4.2, followed by the validation process in Section 4.3 and visualization options offered by the macro in Section 4.4.

4.1 Typestates: The Hard Way

I will start be demonstrating the development process from a state machine specification to a functional prototype, developing all the required components by hand. The example will be a vending machine, its automaton is illustrated in Figure 4.1. To simplify the example, consider the following:

- The machine houses an infinite stock of each of the available snacks.
- Each snack is addressed by its index and the only information available about it is its price.
- The machine does not make change.

We start by designing our *typestated* structure, the `VendingMachine`; to do so, we will use a `State` generic type parameter to model the current state.

```
struct VendingMachine<State>;
```

The compiler will issue an error since the `State` type parameter is currently unused; fixing the error can be done in one of two ways:

- Declaring a `PhantomData`[1] field using `State` as its type parameter. This approach is useful if the types used in `State` do not carry more information other than its type.

  ```
  struct VendingMachine<State> { state: PhantomData<State> }
  ```

- Declaring a field of type `State`. This approach allows us to use more information other than its type alone, such as structure fields.

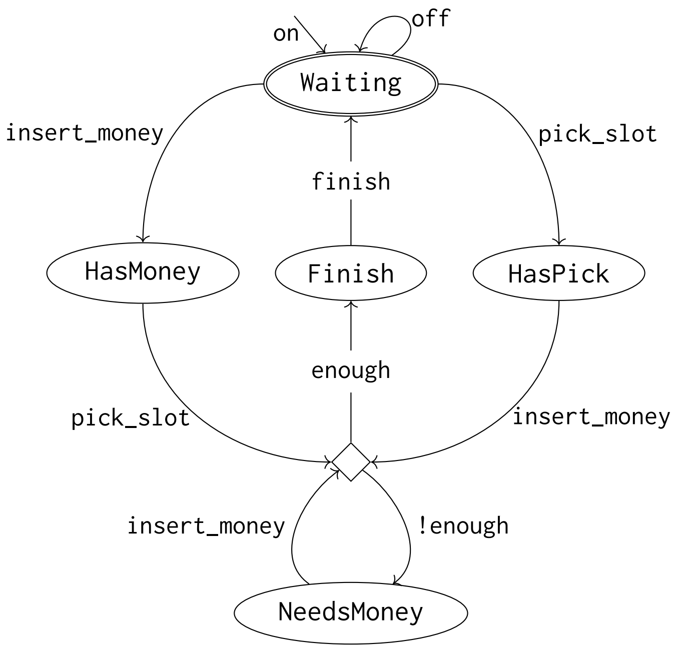


Figure 4.1: Vending machine automaton.

```
struct VendingMachine<State> { state: State }
```

The former is useful where all states are simply markers (i.e. do not carry additional information), however, consider that the vending machine is required to keep track of both the client's pick along with the inserted money so far.

Looking back at Figure 4.1, we can infer the following:

- The `Waiting` and `Finish` states do not require any fields.
- The `HasMoney` and `HasPick` states require their own fields, the money inserted so far and the slot picked by the client, respectively.
- The `NeedMoney` state requires both the money and picked slot.

With that in mind, we are required to take the second approach, enabling states to have inner values.

While our vending machine is now able to deal with the concept of state, it is unable to sell anything, we need some place to store the items available for sale and all the money we made. We will use a vector for the items and an unsigned 64-bit integer for monetary values, see Listing 4.1. These values are available for any state, as they are "part of the machine" and not specific to a given state.

Another problem resides in the fact that the machine supports states, but does not have any. To address this we need to declare each state as a structure; each structure can

[1] `PhantomData` is a zero-sized type used to "pretend" that it owns a previously-unused type parameter (or lifetime). This is required since Rust's compiler will complain in the case that a type parameter is unused. To know more about `PhantomData`, please refer to its documentation page — `https://doc.rust-lang.org/std/marker/struct.PhantomData.html`; or to its page on *The Rustonomicon* — `https://doc.rust-lang.org/nomicon/phantom-data.html`.

```rust
struct VendingMachine<State> {
    /// The money made so far.
    balance: u64,
    /// The available item's prices.
    items: Vec<u64>,
    /// The current machine state.
    state: State,
}
```

Listing 4.1: The vending machine main `struct`.

```rust
/// The machine is waiting for interaction.
struct Waiting;
/// The machine has received some amount of money
struct HasMoney {
    /// The insert amount of money
    money: u64
}
/// The machine has received a slot pick.
struct HasPick {
    /// The selected slot.
    picked_slot: usize
}
/// The machine has received both money and a slot pick,
/// but not enough money to complete the purchase.
struct NeedMoney {
    money: u64,
    picked_slot: usize
}
/// The purchase is complete.
struct Finish;
```

Listing 4.2: The vending machine's states, as illustrated in Figure 4.1.

then contain its own fields, only available for that state; listed in Listing 4.2.

Moving on to transitions, we need to ensure that there are no aliases to the current state; Rust's borrow checker helps us achieve that goal, we can restrict the usage of the current state to only be possible in the case `self` is owned, the borrow checker will then make sure that is true when time comes to use the method.

To declare a transition, we first open an `impl`[2] block which will contain our transition, the block will implement a concrete state of the state machine by specifying the generic type parameter to be one of the declared states, line 1 of Listing 4.3; inside the block, we declare the transition function, it will take `self` as a first parameter, consuming the first state, and return the next state; exemplified in lines 4 & 5 of Listing 4.3.

[2]The `impl` keyword is used for implementation blocks, whether it is for inherent or trait implementations. For further details, refer to *The Rust Reference* — `https://doc.rust-lang.org/reference/items/implementations.html`.

[3]`Self` is a keyword which acts as a type alias to the "current" type, it is native to Rust and works in the context of traits and their implementations. In Listing 4.3 the `Self` will refer to `VendingMachine<Waiting>`. For more information on `Self`, refer to `https://doc.rust-lang.org/std/keyword.SelfTy.html`

```rust
impl VendingMachine<Waiting> {
    fn on() -> Self { /* ... */ }
    fn off(self) { /* ... */ }
    fn insert_money(self, money: u64) -> VendingMachine<HasMoney> { /* ... */ }
    fn pick_slot(self, picked_slot: usize) -> VendingMachine<HasPick> { /* ... */ }
}
```

Listing 4.3: The vending machine's `Waiting` implementation[3].

```rust
1  impl VendingMachine<Waiting> {
2      /// The user has inserted some amount of money into the machine.
3      fn insert_money(self, money: u64) -> VendingMachine<HasMoney> {
4          VendingMachine::<HasMoney> {
5              contents: self.contents, // pass the machine's contents
6              state: HasMoney {        // new state
7                  money                // pass the received money
8              }
9          }
10     }
11     // ...
12 }
```

Listing 4.4: The implementation of `insert_money` for the machine's `Waiting` state.

```rust
// To simplify naming, we reuse the state's names
enum CheckFinish {
    NeedsMoney(VendingMachine<NeedsMoney>),
    Finish(VendingMachine<Finish>),
}
```

Listing 4.5: Vending machine's decision node as a Rust `enum`.

To better understand what is going on, lets implement the `insert_money` function; the function will perform the transition from the `Waiting` state (declared as the generic parameter in line 1 of Listing 4.4), to the `HasMoney` state, declared as the generic parameter of the `VendingMachine` return type, line 3 of Listing 4.4.

Before going further, a quick recap over what has been done so far — we have declared the vending machine, its states and its *some* of its transitions.

I say "some" transitions, because we have not addressed how the diamonds in Figure 4.1 work. We use the diamonds to represent a decision between N possible paths, I will refer to them as *decision nodes*; this representation closely resembles Deterministic Object Automata (DOA) [36]. To model our decision nodes, we can use Rust's enumerations, these allow us to declare possible outcomes and force the API client to match them.

We continue our path, following the `pick_slot` transition from the `HasMoney` state to a decision node, we have *either* the `Finish` state or the `NeedsMoney` state; the implementation of the decision node is described in Listing 4.5.

```
 1  impl VendingMachine<HasMoney> {
 2      fn pick_slot(self, picked_slot: usize) -> CheckFinish {
 3          let money = self.state.money;
 4          let price = self.contents[picked_slot]; // get the pick's price
 5          // Check if there is enough money
 6          if money >= price {
 7              // If yes, return the `Finish` state
 8              CheckFinish::Finish(VendingMachine::<Finish> {
 9                  // update the machine's balance
10                  balance: self.balance + money,
11                  contents: self.contents,
12                  state: Finish,
13              })
14          } else {
15              // If not, return the `NeedMoney` state
16              CheckFinish::NeedMoney(VendingMachine::<NeedMoney> {
17                  balance: self.balance,
18                  contents: self.contents,
19                  state: NeedMoney { money, picked_slot },
20              })
21          }
22      }
23  }
```

Listing 4.6: The pick_slot implementation for the vending machine during the HasMoney state.

```
1  let mut vm: CheckFinish = vm.pick_slot(0);
2  while let CheckFinish::NeedMoney(vm_) = vm {
3      vm = vm_.insert_money(1);
4  }
5  match vm {
6      CheckFinish::Finish(vm) => vm.finish().off(),
7      CheckFinish::NeedMoney(_) =>
8          unreachable!("if we left the loop this should be unreachable"),
9  }
```

Listing 4.7: Matching CheckFinish in two different ways; lines 2-4 — using a while loop, lines 5-9 — using common match.

Using the CheckFinish enumeration, we are now able to properly define HasMoney's pick_slot function; if the user has inserted enough money, a purchase is made (Listing 4.6 — lines 6-13), otherwise, the vending machine asks for more money (Listing 4.6 — lines 14-22), in either case, it returns a variant of the declared enum.

The API client will now be required to match the enumeration, which implies the user needs to (or at least try to) deal with all possible outcomes; exemplified in Listing 4.7.

This concludes the implementation of the state machine, the states I did not cover follow the same implementation pattern, as the automaton is "symmetric", although the functions perform different actions.

To test if our typestates work, we can try to call a function in a state where such

```
let vm = VendingMachine::<Waiting>::on() // Start the vending machine
    .finish();                           // Finish a purchase
```

Listing 4.8: Calling the `finish` function in the `Waiting` state.

```
no method named `finish` found for struct `VendingMachine<Waiting>`
in the current scope items from traits can only be used if the trait is
implemented and in scope the following trait defines an item `finish`,
perhaps you need to implement it:
candidate #1: `Hasher`
```

Listing 4.9: The error resulting from Listing 4.8.

function is unavailable (Listing 4.8); this will not compile, but the compiler will be helpful enough to issue a very complete error, stating that — `finish` was not found for the `Waiting` state (Listing 4.9).

4.1.1 Future Proofing

Our API seems to be rock-solid, methods cannot be called in state they do not belong to and the compiler will even provide helpful messages.

However, there is a problem, nothing stops a developer from extending the API by implementing a "foreign" type (in this context, consider "foreign" to be a type which is not a state), such as the unit type — `()`. Disregarding the fact that implementing the unit type as a vending machine state makes no sense; we need to avoid these situations and to do so *The Rust API Guidelines*[4] offer an answer!

We can implement the "sealed trait pattern", which is just a way of stopping downstream users from modifying our state hierarchy. Following the guidelines, we need to first create a public `trait` which every state will implement (Listing 4.10 — lines 13-23); we need to further restrict the state set with a private `trait` (Listing 4.10 — lines 1-11), also implemented by every state, it is required to be private so downstream users are unable to access and implement it.

4.2 Typestates: The DSL

Now that we know how to build our own typestates, we want to automate the error-prone parts of the process. In this section I present the macro's DSL, I start presenting the DSL's syntax and semantics, followed by a dive into its internals, going through the simpler constructs and how they relate to typestates first, finishing on more advanced features.

4.2.1 Syntax & Automaton Extraction

[4]https://rust-lang.github.io/api-guidelines/future-proofing.html#sealed-traits-protect-against-downstream-implementations-c-sealed

```rust
mod private {
    /// The `Sealed` trait, unable to implemented by downstream users.
    pub trait Sealed {}

    // The trait implementations for each state.
    impl Sealed for Waiting {}
    impl Sealed for HasMoney {}
    impl Sealed for HasPick {}
    impl Sealed for NeedMoney {}
    impl Sealed for Finish {}
}

/// The `State` trait. While any user can *technically* implement it,
/// its bound requires `private::Sealed` to also be implemented,
/// which is impossible because it is not accessible to downstream users.
pub trait State: private::Sealed {}

// The `State` trait implementations.
impl State for Waiting {}
impl State for HasMoney {}
impl State for HasPick {}
impl State for NeedMoney {}
impl State for Finish {}
```

Listing 4.10: The implementation of the sealed trait pattern for our vending machine automaton.

```rust
// The entry point to the DSL
#[typestate]
mod typestate_dsl {
    // Only one automaton per typestate specification
    #[automaton] struct Automaton;
    // N-states are possible
    #[state] struct StateA;
    #[state] struct StateB;
    // Functions are defined inside traits
    // Traits share their name with an existing structure
    trait StateA {
        // Transitions are functions that consume `self`
        // and return an existing state
        fn transition(self) -> StateB;
        // Functions can declare states as initial/final
        // Initial state declarations do not take `self`
        fn new() -> StateA;
        // Final state declarations consume `self` and
        // do not return an existing state
        fn end(self);
    }
}
```

Listing 4.11: The main elements for the #[typestate] DSL.

```
#[typestate] mod vending_machine_api {}
```

Listing 4.12: The vending machine's API module, annotated with the #[typestate] macro.

```
error: Missing `#[automaton]` struct.
   |
   | #[typestate]
   | ^^^^^^^^^^^^
```

Listing 4.13: The error issued by the code in Listing 4.12.

```
#[typestate] mod vending_machine_api {
    #[automaton] pub struct VendingMachine;
}
```

Listing 4.14: Listing 4.12; with an automaton declaration.

#[typestate]'s DSL syntax is interlinked with its automaton extraction process, as such, I will discuss them in conjunction.

A quick primer on the DSL's syntax is presented in Listing 4.11; this section covers each functionality present in the primer by building towards a complete example. I will present parts of the syntax and explain how it relates with the automaton. We will reuse the vending machine example, illustrated in Figure 4.1 and model it using our DSL.

The #[typestate] macro is the DSL's entrypoint and it only supports being attached to modules (Listing 4.11 lines 1-3). Given that we want to access several parts of Rust's syntax (e.g. `struct`, `enum`, etc) we can take one of two approaches — either analyze the whole file with an external tool, or annotate and process the best next thing, the module.

The module provides most of the syntax elements available to "top-level" Rust, while being possible to analyze using the macro system; inside a module we can declare structures, enumerations, free functions and so on.

To start modeling the vending machine we first declare a module, to which we will call `vending_machine_api`, and annotate it with `#[typestate]`; as shown in Listing 4.12. This alone is not enough, as the macro will throw an error due to the lack of an automaton; shown in Listing 4.13.

The #[automaton] annotation is attachable to structures only, and allows the macro to know which of the declared structures is the automaton (Listing 4.11 lines 4-5).

Listing 4.14 fixes the error of Listing 4.12, by adding the `VendingMachine` structure and annotating it with `#[automaton]` the macro is now able to know which structure is the main state machine (i.e. which structure will be *typestated*).

Notice how the code from Listing 4.15 does not contain any reference to the current state, that is added by macro through the `#[automaton]` annotation, along with the sealed pattern skeleton; described in Section 4.1.1.

```
mod vending_machine_api {
    mod private {
        pub trait Sealed {}
    }
    pub trait State: private::Sealed {}
    pub struct VendingMachine<S> where S: State {
        state: S
    }
}
```

Listing 4.15: Code resulting from Listing 4.14 expansion.

```
#[typestate] mod vending_machine_api {
    #[automaton] pub struct VendingMachine;
    #[state] pub struct Waiting;
    #[state] pub struct HasMoney { money: u64 }
    #[state] pub struct HasPick { picked_slot: usize }
    #[state] pub struct Finish;
    #[state] pub struct NeedMoney {
        pub money: u64,
        pub picked_slot: usize,
    }
}
```

Listing 4.16: Listing 4.14; with all states declared.

```
mod vending_machine_api {
    // ...
    pub struct NeedMoney {
        pub money: u64,
        pub picked_slot: usize,
    }
    // using the qualified path (i.e. `private::Sealed`) we sidestep the
    // requirement of being *inside* the `private` module
    // to implement the `Sealed' trait
    impl private::Sealed for NeedMoney {}
    impl State for NeedMoney {}
}
```

Listing 4.17: Expansion of the NeedMoney state, declared in Listing 4.16.

Once again, this still does not make the macro happy, while we now have an automaton, we are lacking initial and final states; as shown in Listing 4.18.

The #[state] annotation, like the #[automaton] annotation, is only attachable to structures (Listing 4.11 lines 6-8); it marks them as states and also implements the necessary traits to include the state in the sealed trait state set (Section 4.1.1).

As we can observe in Listing 4.16, declaring states is as simple as attaching the annotation to an existing structure. In Listing 4.17 we can see the expansion of the NeedMoney state; implementing the sealed trait pattern.

```
error: Missing initial state. To declare an initial state you can use a
function with signature like `fn f() -> T` where `T` is a declared state.
--> vm-typestate/src/main.rs:15:1
    |
    | #[typestate]
    | ^^^^^^^^^^^^
    |

error: Missing final state. To declare a final state you can use a
function with signature like `fn f(self) -> T` where `T` is not a declared state.
--> vm-typestate/src/main.rs:15:1
    |
    | #[typestate]
    | ^^^^^^^^^^^^
    |
```

Listing 4.18: The error issued by the code in Listing 4.12.

While we have declared our states, we still have the same error (Listing 4.18); that is because, currently, we only have loose states, we have not connected them in any meaningful way.

Function declarations allow us to declare transitions without any kind of annotations (Listing 4.11 lines 9-21); we can simply check the function signature and infer the kind of transition, however to do so, we first need to establish rules, those are:

- If a function takes `self` and returns a valid state, the function is considered to be a transition between the current state and the returned state.

  ```
  fn (self, ...) -> State;
  ```

- If a function *does not* take `self` as an argument and returns the current state, it describes the current state as an initial state.

  ```
  fn (...) -> State;
  ```

- If a function takes `self` as an argument and *does not* return a valid state, it describes the consumed state as a final state.

  ```
  fn (self, ...) -> ...;
  ```

To declare functions, we first need to declare a `trait` with the same name as the target state, by doing this, the macro is able to know which state we are currently referring to; inside the trait, we can declare all functions to be implemented by the current state.

If the reader is familiarized with Rust, they might have realized that traits cannot share names with structures, enumerations or others; in our DSL that works because during expansion the trait is renamed as: `TraitName + State => TraitNameState`.

In Listing 4.19, we use the `Waiting` state as it contains all the previously described types of transitions.

```
#[typestate] mod vending_machine_api {
    #[state] pub struct Waiting;
    // The trait is named after the `Waiting` state,
    // thus, the macro knows which state is the *current* one.
    pub trait Waiting {
        // Does not consume self, returns the current state: initial state
        fn on() -> Waiting;
        // Consumes self, does not return: final state
        fn off(self);
        // Consume self and return a valid state: transitions
        fn insert_money(self, money: u64) -> HasMoney;
        fn pick_slot(self, picked_slot: usize) -> HasPick;
    }
}
```

Listing 4.19: Declaration of the `Waiting` state functions.

Annotation	Attaches to	Declares
#[typestate]	Module	API
#[automaton]	Structure	Automaton/Typestate
#[state]	Structure	State

Table 4.1: Overview of the DSL's annotations.

Implementing the states and transitions is similar to what we did in Section 4.1, while we have taken care of the sealed pattern, how the machine behaves is left to us.

When using the DSL, instead of declaring an implementation for the target state, as follows:

```
impl VendingMachine<Waiting> { /* ... */ }
```

You implement a trait for the target state:

```
impl WaitingState for VendingMachine<Waiting> { /* ... */ }
```

This way, the compiler is able to point out which methods are missing (and in the future, tools like `rust-analyzer`[5] might add all missing signatures for the developer). The rest of the implementation is made in the same way as the one in Section 4.1.

4.2.1.1 Summary

In Section 4.2.1 I have introduced the basic features of the DSL; in Table 4.1 I provide a quick overview of the available annotations, `#[typestate]`, `#[automaton]` and `#[state]`; in Table 4.2 I review the transition inference rules for function declarations.

[5]https://rust-analyzer.github.io/

Function signature	Consumes a state	Returns a state	Inferred
`fn (self, ...) -> State;`	✓	✓	Transition
`fn (...) -> State;`		✓	Initial State
`fn (self, ...) -> ...;`	✓		Final State

Table 4.2: Overview of the transition inference rules.

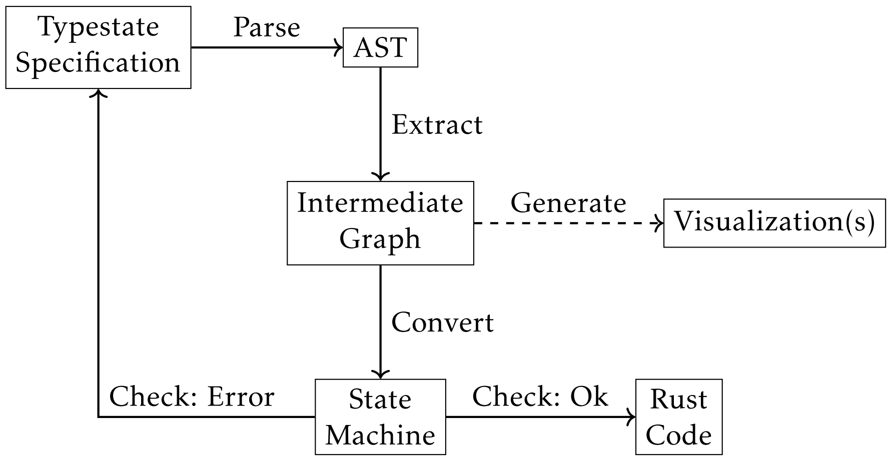


Figure 4.2: From DSL specification to Rust code. First the DSL is parsed, an intermediate graph representing the automaton in more general terms is extracted from the AST, from the graph the macro will convert the user can generate visualizations (for debugging or documentation), this step is optional.

4.2.2 Architecture

This section provides an overview over the macro architecture, afterwards I will dive into the parsing and code generation details.

The macro's architecture is illustrated in Figure 4.2, as demonstrated in the previous section, the DSL's entry point is the `#[typestate]` attribute, attaching it to a module will cause the macro system to run our code during expansion and process the module's code as we wish. During expansion the Abstract Syntax Tree (AST) of the module is passed in to the macro code where a series of visitors are defined and run (described Section 4.2.2.2), performing the state's machine extraction and verification (described in Section 4.3). If verification fails the macro issues an error (or errors), pointing the user to the relevant code; if all checks pass, the user is ready to start implementing the typestate's functionalities.

4.2.2.1 Parsing

The macro's parsing procedure leverages the syn crate[6] to simplify the parsing process, this allowed me to focus on getting the most information out of the user's code rather than worrying about how to parse Rust.

[6]syn is a parsing library for Rust's `TokenStream`, for more information please see `https://docs.rs/syn/1.0.72/syn/index.html`.

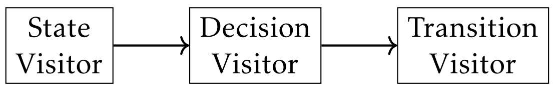


Figure 4.3: The `#[typestate]` macro visitors, by running order.

As the `#[typestate]` macro can only be attached to modules, we instruct syn to expect and parse a module item[7]; this simple step already saves us from manually ensuring the macro is attached to the right item. The resulting item is the module's AST, from which we will run a series of visitors, each analyzing a different part of the code.

4.2.2.2 Visitors

The macro is split into three separate visitors, each performs a pass over different item kinds and if necessary, mutates the tree by generating new code and either adding or replacing existing nodes. The following visitors are described in their running other; shown in Figure 4.3.

The structure visitor will visit all structures, as the name states; currently, a user can declare one of three possible structures;

- A structure annotated with `#[automaton]` (of which there can only be one).
- One annotated with `#[state]` (of which there can be N).
- One without annotations (of which there can be *none*).

From the visited structures we can extract the automaton's structure and its states, these are added to the graph and the sealed trait pattern is implemented for each structure.

The enumeration visitor solely visits enumerations; it checks that all enumeration variants exist as states and establishes edges between them and the decision node. This visitor is also known as the decision state visitor since each enumeration represents a decision to be made during runtime.

The trait visitor, or the transitions visitor, is responsible for the extraction of all the transitions out of the declared traits. This visitor conflicts with the first one since it generates traits which would then be visited; to avoid this problem `typestate` declares an additional, undocumented and inert macro (i.e. a macro that returns its input); the `#[generated]` macro. When the visitor sees an item attached with `#[generated]` it ignores the item, the `#[generated]` macro is later processed as a regular macro by the macro system and the annotation is removed.

4.2.3 Advanced Features

Section 4.2.1 presents the basic features of the DSL, armed with them, the reader should be able to write typestates. However, some typestates will require more complex mechanisms, both to develop and use, that is the purpose of this section. Once more, we will expand over our recurring example, the vending machine; illustrated in Figure 4.1.

[7] `https://doc.rust-lang.org/reference/items/modules.html`

```rust
#[state] struct NextState;
#[state] struct ErrorState;

enum FallibleOperationResult {
    NextState,
    ErrorState
}
```

Listing 4.20: Fallible operations can be using enumerations like the `FallibleOperationResult` decision state.

```rust
enum FallibleOperationResult {
    #[metadata(label="Success!")]
    NextState,
    #[metadata(label="Error happened because network failed.")]
    ErrorState
}
```

Listing 4.21: Listing 4.20 enumeration with the `metadata` attribute.

4.2.3.1 Decision states

Decision states were previously discussed in the final of Section 4.1, and were first listed in Listing 4.5. I now show how the DSL handles this kind of state.

There are cases where it is required that some uncertainty is modelled into the typestate, maybe the typestate depends on a procedure that can fail or maybe, given the input, the resulting state may differ; such states are required to be modelled, the used representation is based on DOA [36].

To achieve such goal, I took advantage of Rust's enumerations as they can represent several types under a single `enum`, my DSL tweaks their semantic from normal Rust. Instead of allowing enumerations containing several kinds of variants, each with their respective name, the DSL enforces that each variant is of the `Unit`[8] type and shares its name with an existing state; an example is provided in Listing 4.20.

In contrast to Listing 4.5, the DSL's decision state's variant do not have arguments, these are generated from their name by the macro.

Transition labels can be added through the use of the `metadata` attribute, this attribute allows the addition of relevant metadata to the transition edge. This is especially useful to specify under which conditions the transition from a decision state to a specific state happens; currently, the attribute only supports the `label` value. A usage example of the `metadata` attribute is listed in Listing 4.21.

[8]`https://docs.rs/syn/1.0.73/syn/enum.Fields.html#variant.Unit` (visited in 08/07/2021)

```
 1  pub trait NeedMoney {
 2      fn insert_money(self, money: u64) -> CheckFinish;
 3      fn get_message(&self) -> String;
 4      fn update_pick(&mut self, new_pick: usize);
 5  }
```

Listing 4.22: The NeedMoney, extended with the get_message and update_pick functions.

```
 1  impl NeedMoneyState for VendingMachine<NeedMoney> {
 2      // ...
 3      /// Return the message to be displayed.
 4      fn get_message(&self) -> String {
 5          let state = &self.state;
 6          let unpaid_amount = self.contents[state.picked_slot] - state.money;
 7          format!("{} left to go!", &unpaid_amount)
 8      }
 9      /// Update the current user pick.
10      fn update_pick(&mut self, new_pick: usize) {
11          self.state.picked_slot = new_pick;
12      }
13  }
```

Listing 4.23: The implementation of NeedMoney's new functions, as declared in lines 3 & 4 of Listing 4.22.

4.2.3.2 Self-transitions

Consider that we are asked to display a message containing the amount left to pay in the NeedMoney state; to do so we can simply add a new function to the NeedMoney trait, like in line 3 of Listing 4.22. Notice how the new method takes &self instead of self, thus, it takes the state as an *immutable* reference, instead of consuming the state; disabling mutation of the current state. Mutable references are also supported, line 4 of Listing 4.22 declares a method taking a mutable reference to the current state, which in turn allows the user to update its snack selection.

When working with automata we need to consider that every transition has both a source and a destination, in the case of functions that take references to self, immutable or not, they still represent transitions, in this case, the source and destination are the same state; hence the name of *self-transitions*.

4.2.3.3 State enumeration

There are some cases in which an enumeration might come in handy, one of them is when you are required to loop forever, and you may "stop during processing" (i.e. not complete a full Waiting to Waiting cycle). In this case you will need a variable that can contain one of the many possible states the machine might be in, given that you cannot replace a variable's type once it is assigned, you will need to use Rust's enumerations.

```
#[typestate(enumerate)] mod vending_machine { /* ... */ }
```

Listing 4.24: Using the enumerate macro attribute.

```
enum EVendingMachine {
    Waiting(VendingMachine<Waiting>),
    HasMoney(VendingMachine<HasMoney>),
    HasPick(VendingMachine<HasPick>),
    NeedMoney(VendingMachine<NeedMoney>),
    Finish(VendingMachine<Finish>),
}
```

Listing 4.25: The resulting enumeration of the enumerate attribute, demonstrated in Listing 4.24.

```
#[typestate(state_constructors)] mod vending_machine { /* ... */ }
```

Listing 4.26: Using the state_constructors macro attribute.

For large state machines, writing the enumeration by hand is error-prone and not practical; to address these issues #[typestate] offers the enumerate macro attribute.

By changing the attached #[typestate] annotation to #[typestate(enumerate)], as demonstrated in Listing 4.24, it will generate the enumeration described in Listing 4.25. Along with the enumeration, the macro will also implement the From conversion trait between the enumeration and the respective states; this way, if the API client wishes to convert a VendingMachine<Waiting> into EVendingMachine, they will be able to perform the conversion using .into() or .from().

4.2.3.4 State constructors

A small quality-of-life improvement is the automatic declaration of state constructors, shortening the declaration of a new state instance; these are only generated for states containing named fields and the constructor's parameters will be named after them. Its usage is similar to that of the enumerate attribute, as shown in Listing 4.26, declaring #[typestate(state_constructors)] will generate the constructors with the default name new_state, demonstrated in Listing 4.27.

```
impl NeedMoney {
    pub fn new_state(money: u64, picked_slot: usize) -> Self {
        Self { money, picked_slot }
    }
}
```

Listing 4.27: The generated constructor for the NeedMoney state; using the attribute shown in Listing 4.26.

```
impl<S, T> From<IntermediateGraph<S, T>> for GenericAutomaton<S, T>
where
    S: Hash + Eq + Debug + Clone + Display,
    T: Hash + Eq + Debug + Clone + Display,
{
    fn from(i: IntermediateGraph<S, T>) -> Self {
        // ...
    }
}
```

Listing 4.28: Implementation of the `From` trait for the conversion between `IntermediateGraph` and `GenericAutomaton`

4.3 Validation

In this section I will be discussing the validation strategies used in my work, I start by discussing the guarantees provided by the macro, followed by the typestate validation strategy, finally, I present the automaton validation strategy.

Guarantees. My library aims to provide guarantees related with automata, not typestates; this is the case because Rust's compiler is able to reason over typestates already, as long as we put it to good use we can rely on the borrow checker and type system to catch typestate related errors (i.e. calling a method in the wrong state).

Regarding automata, my macro provides the following:

Non-empty language. The language of the automaton should not be empty; the macro ensures the presence of final states.

Usefulness. All states should be useful, that is, all states should be reachable from the initial state and be able to reach the final state.

Minimality is not present in the list of provided guarantees because it is unclear how the macro could provide feedback to the user; while the macro can simply state — *"The presented automaton is not minimal"*, such message does not help the user correcting the problem. Furthermore, non-deterministic automata are required to first be *determinized*, producing new states which the user is unable to see and thus providing feedback on them is not the best solution.

4.3.1 The Automaton and the Graph

The automaton is extracted from the intermediate graph, this is done by implementing Rust's `From` conversion trait (see Listing 4.28), while the representations are very similar, the `GenericAutomaton` is a stripped and specialized version of the intermediate graph.

The automaton is modelled as a directed graph, this approach is more efficient than that of projects like OFlat [25], which uses sets. Originally I used the `petgraph` library[9],

[9]https://docs.rs/petgraph/0.5.1/petgraph/index.html

```rust
pub trait Property {}

pub trait Validate<P: Property> {
    type Out;
    fn validate(&self, _: P) -> Self::Out;
}
```

Listing 4.29: The `Validate` trait is generic over the property being validated, but not its output, hence the `Out` type being declared as an associated type.

however the library turned out to be inadequate to the problem at hand, `petgraph`'s `GraphMap`[10] required node types to be `Copy`[11] and the types I am storing are neither `Copy` nor defined by me.

To fix the problem, I have implemented a directed graph using adjacency lists, that is, for each node, there is a list containing its neighboring nodes and the respective connecting edges. This approach takes from the FAdo [29] project which also uses adjacency lists; to simplify some algorithms, I have added extra structures such as the inverse δ, or δ^{-1} (in automata, the delta set δ represents an automaton's transitions, δ^{-1} is the delta set where each transition has had their direction inverted).

4.3.2 Implementing a validation strategy

The current macro architecture implements an intermediate graph over which several transformations and analysis can be done, one of which being validation. The typestate validation is performed over a general representation of automata (extracted from the intermediate graph), the representation has been adapted from the FAdo library [29]. To allow for multiple validation strategies I defined a `Validate` trait, presented in Listing 4.29, the trait is implemented for a structure, over a `Property` (a blanket trait used to define strategies).

Implementing the `Validate` trait requires a struct which will implement the `Property` trait, and an implementation of the `Validate` trait for a structure over the property being validated; this is listed in Listing 4.30.

The empty language verification is easy to address, since the automaton is required to have end states, I simply store all end states in a set, then I am only required to check if the set is empty or not.

Usefulness can be checked by first performing a graph search (I have opted for Depth-First Search (DFS)). Useful states are those that reach the end state(s) (i.e. productive states) and reachable from the initial state(s). The end state set comes in useful here, since we can simply iterate over all states in the set and start the graph exploration, visited nodes will be the productive ones, in the case that not all states are productive the

[10] https://docs.rs/petgraph/0.5.1/petgraph/graphmap/struct.GraphMap.html
[11] https://doc.rust-lang.org/std/marker/trait.Copy.html

```rust
pub struct UsefulStates;

impl Property for UsefulStates {}

impl<State, Transition> Validate<UsefulStates> for
    GenericAutomaton<State, Transition>
where
    State: Hash + Eq + Debug + Clone + Display,
    Transition: Hash + Eq + Debug + Clone + Display,
{
    type Out = HashSet<State>

    fn validate(&self, _: UsefulStates) -> Self::Out {
        // ...
    }
}
```

Listing 4.30: Implementation example of a validation strategy, in this case it checks that all states are useful (i.e. all states are part of a path from the initial state to the final state.), implementation details are omitted but can be seen in https://github.com/rustype/typestate-rs/blob/16da7790ef864054eb5bddde4f10c64ed2bcd511/typestate-proc-macro/src/igraph/validate.rs#L178-L209.

verification stops here and issues an error, highlighting the culprits in the code. Checking for useful states is then as simple as repeating the exploration procedure starting on the initial states, the set of visited states can then be intersected with the set of productive states to compute the useful states, once more, in the case that not all existing states are useful, an error is issued, highlighting the state at fault.

4.4 Visualizing Typestates

Along with all the previously described features, I have implemented an embedded typestate visualization; the visualization supports exporting to Graphviz DOT[12] and PlantUML state diagrams[13], along with documentation embedding using Mermaid.js[14] and the `aquamarine` crate[15] (detailed in Section 4.4.2). All formats have their own feature flags (only `aquamarine` is part of the default set of features), and all formats leverage the graph used by the macro for analysis to export their visualizations.

4.4.1 Debugging with Visualizations

The visualization feature was born out of necessity, having a tool to visualize the typestate graph extracted by the macro was invaluable, reducing the time spent debugging the macro and the generated typestates. However, while it was born to debug the macro,

[12]https://graphviz.org/doc/info/lang.html
[13]https://plantuml.com/state-diagram
[14]https://mermaid-js.github.io/mermaid/#/stateDiagram
[15]https://docs.rs/aquamarine/0.1.9/aquamarine

```rust
pub struct Mermaid;

impl super::Format for Mermaid {}

impl<S, T> Export<Mermaid> for IntermediateGraph<S, T>
where
    S: Hash + Eq + Debug + Clone + Display,
    T: Hash + Eq + Debug + Clone + Display,
{
    fn export<W: std::io::Write>(&self, w: &mut W, f: Mermaid) -> Result {
        // ...
    }
}
```

Listing 4.31: Implementation example of the `Export` trait for the `Mermaid` format. The full code is available at https://github.com/rustype/typestate-rs/blob/16da7790ef864054eb5bddde4f10c64ed2bcd511/typestate-proc-macro/src/igraph/export.rs#L18-L118

it can also be used to visually debug the typestate; in large systems it is hard for one to visualize the typestate in their mind, having an actual picture of the system helps immensely.

The implementation of export formats follows the same pattern used for validation, having an `Export` trait which declares an `export` function and a `Format` trait which is then implemented by a structure, in the `Export` trait case, it is implemented by the intermediate graph. An example is provided in Listing 4.31.

DOT was the first format to be exported as it was the first one that came to mind, it is very simple and there are a lot of tools which leverage the format. To use this feature the user can simply run cargo with any command that expands the code along with `-features typestate/export-dot`; this will export all typestates of the project into their own `$TYPESTATE_NAME.dot` file.

PlantUML offers the *state diagram* format, providing a more concise way of describing the typestate by allowing us to have dedicated initial and final states, as well as decision nodes. Just like the previous feature, exporting PlantUML is done using features; in this case the flag is `-features typestate/export-plantuml`, which will export all typestates into separate `$TYPESTATE_NAME.uml` files.

Customization of the exported formats is possible through environment variables, these are listed in Table 4.3.

4.4.2 Embedding Visualizations in the Documentation

Exporting typestates in a way that enables the developer to visualize them is a valuable tool during development and debugging, also improving communication. However, when focusing on communication, the best way to ensure the API client gets to see the typestate

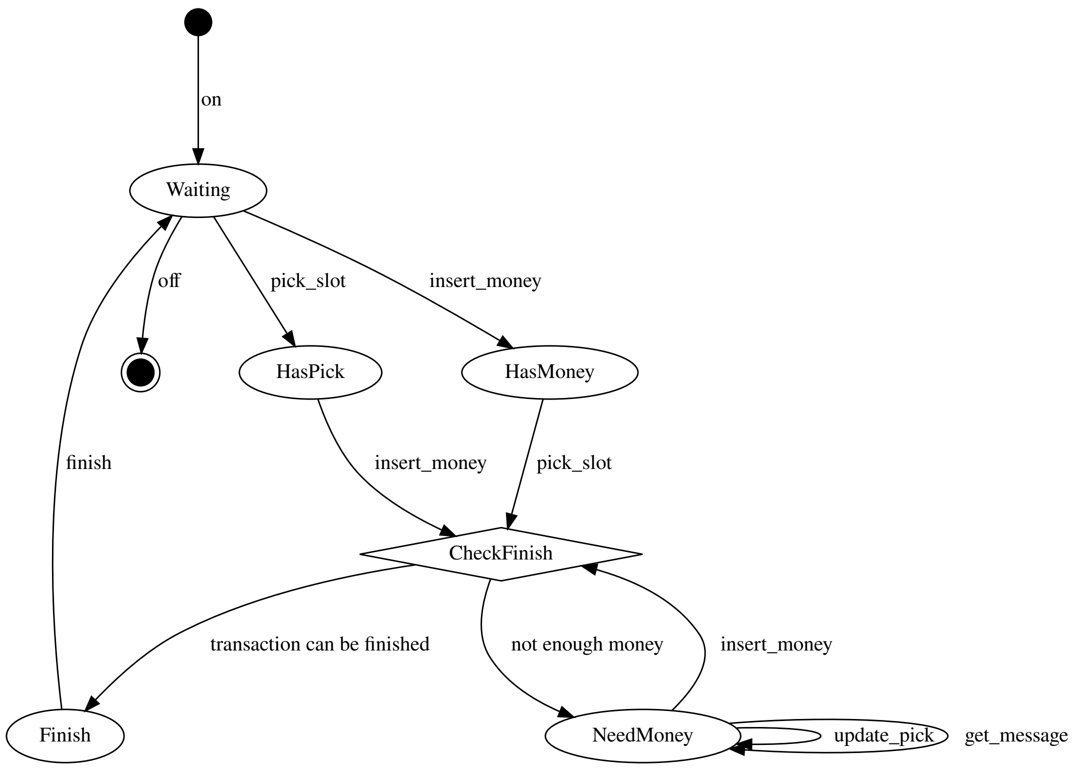


Figure 4.4: The vending machine's DOT typestate, rendered using the command — `dot -Tsvg VendingMachine.dot`.

Tool	Environment Variable	Description
DOT	`DOT_PAD`	Specifies how much, in inches, to extend the drawing area around the minimal area needed to draw the graph.
	`DOT_NODESEP`	In `DOT`, nodesep specifies the minimum space between two adjacent nodes in the same rank, in inches.
	`DOT_RANKSEP`	In `DOT`, sets the desired rank separation, in inches.
PlantUML	`PLANTUML_NODESEP`	nodesep specifies the minimum space between two adjacent nodes in the same rank.
	`PLANTUML_RANKSEP`	Sets the desired rank separation.
Both	`EXPORT_FOLDER`	Declare the target folder for the exported files.

Table 4.3: All configuration parameters for the DOT and PlantUML visualization features.

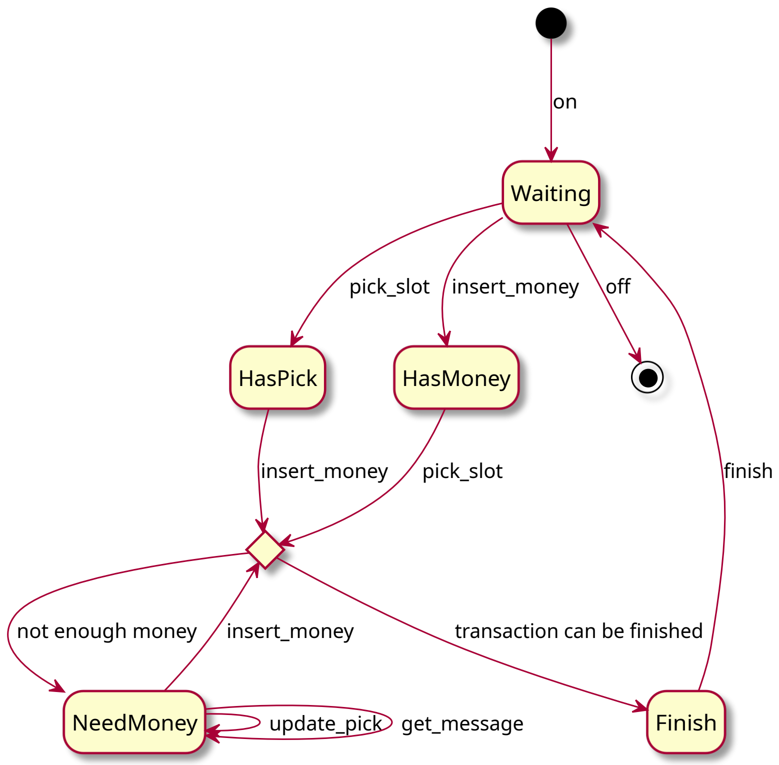


Figure 4.5: The vending machine's PlantUML typestate, rendered using the command — `plantuml -tsvg VendingMachine.uml`.

would be to embed it in the documentation; as documentation is the *de facto* way to communicate between a library's author and its users.

Unfortunately, embedding images in Rust documentation requires a link to that image, which in turn, requires some other place to host the image; this constraint makes it more complicated to embed the DOT or PlantUML render inside the documentation. Ideally, we want everything in one place, generated in one step!

Fortunately, `aquamarine` addresses that problem; it allows the declaration of Mermaid.js diagrams as documentation and then renders them as HTML inside the Rust documentation. Its syntax is *very similar* to PlantUML's, thus this feature's implementation process was mostly porting the PlantUML generation code and fixing any bugs which appeared.

Rendering the state diagram starts by adding *doc comments*[16] to the module during the macro processing, the *doc comment* contains the diagram description in the Mermaid.js specification language; the expanded code for the vending machine example (Figure 4.1) can be seen in Listing 4.32. When the user runs the documentation command — `cargo doc`; the `aquamarine` attribute is attached and the comment is processed by the `aquamarine` macro, the final diagram is then made available in the documentation; pictured in Figure 4.6.

[16] https://doc.rust-lang.org/reference/comments.html#doc-comments

```
///```mermaid
///stateDiagram-v2
///[*] --> Waiting : on
///state CheckFinish <<choice>>
///CheckFinish --> NeedMoney : not enough money
///CheckFinish --> Finish : transaction can be finished
///HasMoney --> CheckFinish : pick_slot
///HasPick --> CheckFinish : insert_money
///Waiting --> HasMoney : insert_money
///Waiting --> [*] : off
///Waiting --> HasPick : pick_slot
///NeedMoney --> CheckFinish : insert_money
///NeedMoney --> NeedMoney : get_message
///NeedMoney --> NeedMoney : update_pick
///Finish --> Waiting : finish
///```
mod vending_machine_api { /* ... */ }
```

Listing 4.32: *Doc comments* resulting for the expansion of the vending machine example (Figure 4.1).

Bundling the macro in way that users can depend on this feature is not a trivial task; we want users to simply import the `typestate` library and be able to embed their typestates in the documentation. Rust applies some restrictions to procedural macro libraries, namely, such libraries cannot export anything else other than the defined macros; this is a deal-breaker since the `typestate` crate *is* a procedural macro crate and forcing the user to explicitly import `aquamarine` into their project is more overhead than necessary.

The solution for this is to create a *frontend* crate which imports both the `typestate` macro and `aquamarine`, and then exports both, this sidesteps the previous issue since the crate exporting the items *is not* the macro crate; this process is pictured in Figure 4.7, Figure 4.8 and Figure 4.9.

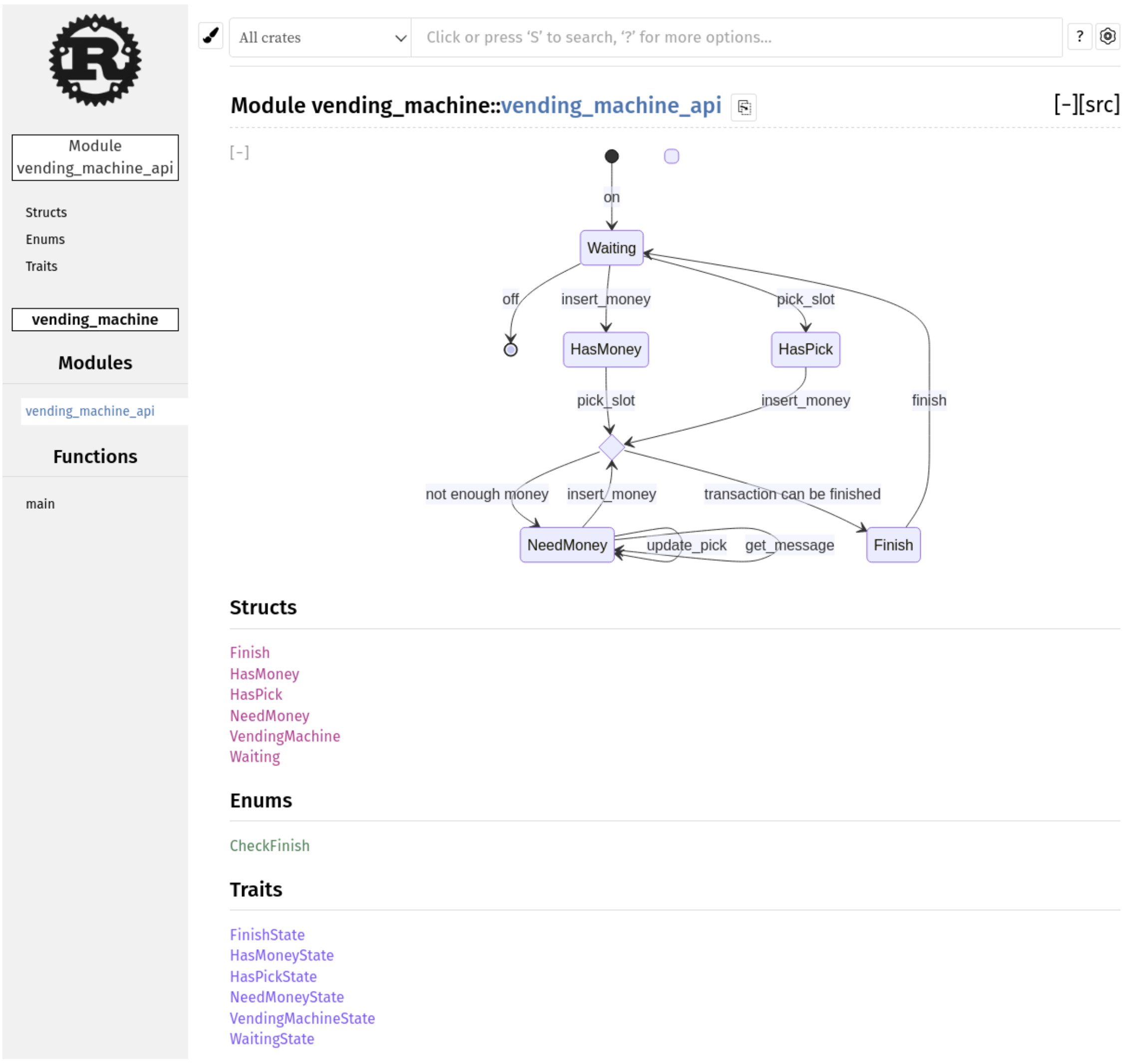
4.5 Summary

In this chapter I have presented:

- How the user can write typestates by hand (Section 4.1).
- How the DSL is architected, its syntax and more advanced features (Section 4.2).
- How the built typestate is validated (Section 4.3).
- How the user can visualize and document their typestates (Section 4.4).

The use of procedural macros is nothing new to the ecosystem, however, DSLs are usually built with function-like procedural macros. While such approach has advantages (e.g. the usage of new syntax and more expressive constructs) it comes at the cost of not

[17] `typestate-deps` is the set of dependencies for the macro (e.g. `syn`, `quote`, etc); it is dashed as it is not exported.



Structs

Finish
HasMoney
HasPick
NeedMoney
VendingMachine
Waiting

Enums

CheckFinish

Traits

FinishState
HasMoneyState
HasPickState
NeedMoneyState
VendingMachineState
WaitingState

Figure 4.6: The vending machine API documentation page. Result of Listing 4.32 when rendered using cargo doc.

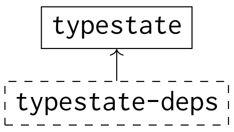


Figure 4.7: The original configuration, the macro depends only on typestate-deps[17] and does not export any dependency.

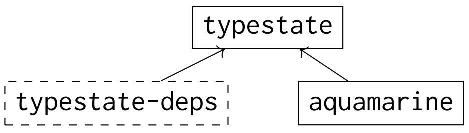


Figure 4.8: The naive attempt, the macro depends on typestate-deps[17] and aquamarine, but it only tries to export aquamarine, this fails because typestate is a procedural macro crate.

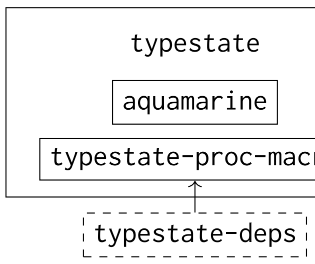


Figure 4.9: The macro was isolated in its own crate — `typestate-proc-macro`, which depends on `typestate-deps`[17]; the `typestate` crate now depends and exports both `aquamarine` and `typestate-proc-macro`.

only being harder to develop, but also requiring the developer to pay the upfront cost of learning it before they can use the DSL.

The present work is by no means final, there is a lot to improve upon (this is detailed in Chapter 6), but nonetheless, it achieves interesting insights:

DSLs do not need to be a complete new language or even extend their host language; attaching a macro to a module provides a near complete language by itself, upon which we can simply tweak its semantics to our purposes.

My DSL minimizes the cost of getting up and running by lightly tweaking Rust's semantics and building new constructs on top of it, such as the automata and state structures. While learning cost is reduced, a disadvantage is that some users may find the tweaks to be "non-natural".

Diagrams are powerful communication tools, as the popular saying goes — *"A picture is worth a thousand words"*. Leveraging the graph-like nature of typestates to produce diagrams provides an extraordinary tool to aid in any development related to the typestate. Taking advantage of Rust's powerful macro system, one can eventually adapt these to other areas such as session types, or other areas where the line between code and information is blurred.

To the best of my knowledge, my work is the first to leverage an existing embedded DSL and generate a moderately complex form of documentation. As a visual-leaning learner, I believe this feature can help users of the final APIs to understand how their typestates work, especially in more complex settings where state relationships may be difficult to visualize mentally. Moreover, these diagrams, being specifications, also facilitate debugging, maintenance and extension of the applications.

5 Case Studies

In this chapter I present some case studies where my library was used and discuss some of its strengths and weaknesses. This chapter's example's full code is available at https://github.com/rustype/typestate-examples.

5.1 Ring

The ring example is taken out of the Rumpsteak [7] repository. The general example is composed of N participants, in a *ring*, where the first participant sends the value to the participant on the right, the other participant receives the value from the left and each participant repeats the same process; visualized in Figure 5.1.

Before comparing both APIs, it is important to notice that Rumpsteak is aimed at Multi-Party Session Types (MPST) for Rust's `async`/`await` syntax, enabling the description and enforcement of communication patterns using the type system. `#[typestate]` was designed with typestates and API constraints in mind, while it is unable to describe and enforce communication patterns, it is able to enforce function call ordering at compile-time, avoiding API misuses.

Taking it a step further, consider Figure 5.1, Rumpsteak is enforces constraints on the edges of that graph (i.e. "external" protocol), such as where they come from and where they go to, while `#[typestate]` enforces constraints on the nodes (i.e. "internal" protocol), such as the possible steps the node will follow.

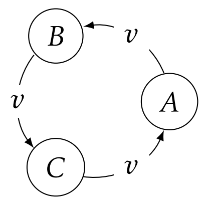

$$\begin{tikzcd} B \arrow[d, "v"'] & & \\ C \arrow[r, "v"'] & A \arrow[ul, "v"'] \end{tikzcd}$$

Figure 5.1: The ring example visualization.

5.1.1 Comparison

Rumpsteak first declares a series of types, the global type — `Roles`; each participant endpoint — `struct`s `A`, `B` and `C`; the message being passed around — `Value`; finally, each session type — `struct`s `RingA`, `RingB` and `RingC`. The functionality of each participant is then declared as an `async` function, notice that `ring_b` (lines 32-39 from Listing 5.1) and `ring_c` (lines 41-48 from Listing 5.1) are identical, except for the types used.

Given the session type nature of Rumpsteak, the library is able to enforce communication patterns at the type level, In lines 19-21 of Listing 5.1 we can see through `RingA`'s type that `A` will forcibly send the value to `B` and will then receive the value from `C`; this is not enforceable using `#[typestate]`.

The launch routine Listing 5.2 is standard, using the executor from the `future` crate along with the `try_join` macro as a convenience to launch each one. Rumpsteak ends up being fairly verbose w.r.t. the used types, although the final usage (i.e. the `main` and `ring_` functions) is straightforward.

`#[typestate]` type usage requires less background knowledge than Rumpsteak, a user that understands the DSL should be able to understand what goes on "behind the scenes".

In my example (Listing 5.3, Listing 5.4, Listing 5.5) the protocol is enforced by declaring the state machine of each participant; in this case, while the ring can scale up to N participants, only two protocols are required — one for the participant starting the value propagation, and one for the all other participants, which need to receive the value first. The participants typestates are visualized in Figure 5.2.

Both `RingA` and `RingB` receive two channels (respectively, lines 12 & 23-25 in Listing 5.3, and lines 18 & 38-40 in Listing 5.4), one from which they will receive the value, and one to which they will send the value, these channels, respectively, connect to the previous and next participants in the ring; the channels are received upon instantiation of the respective `Ring` object.

`RingA` (Listing 5.3) starts the process, thus, it is the only participant starting in a *sending* state (lines 10-16, emphasizing line 12, which declares the initial state); in this state the API allows the client to inspect the value with `get_value` (lines 13 & 27); transition to the next state with `send` (lines 14 & 29-36); or end the protocol with `end` (lines 15 & 38).

Inversly to `RingA`, `RingB` will first receive the value, as it would be the participant next to the first sender and thus all other participants; conversely, the protocol (Listing 5.4) will start in a *receiving* state (lines 16-21, emphasizing line 18, which declares the initial state). The main difference from `RingA` to `RingB` is the initial state, the inner logic is the same and besides `new` (lines 18 & 38-40) and `end` (lines 20 & 51), other functions belong to the same states.

While `#[typestate]` is unable to enforce communication patterns, it is able to reuse the participants more effectively (i.e. who sends to whom); given that `B` and `C` perform the same operations, we can simply reuse `RingB` (Listing 5.4) for both `B` and `C` (lines 7-8 of Listing 5.5).

```rust
#[derive(Roles)]
struct Roles(A, B, C);

#[derive(Role)]
#[message(Value)]
struct A(#[route(B)] Sender, #[route(C)] Receiver);

#[derive(Role)]
#[message(Value)]
struct B(#[route(A)] Receiver, #[route(C)] Sender);

#[derive(Role)]
#[message(Value)]
struct C(#[route(A)] Sender, #[route(B)] Receiver);

#[derive(Message)]
struct Value(i32);

#[session] type RingA = Send<B, Value, Receive<C, Value, End>>;
#[session] type RingB = Receive<A, Value, Send<C, Value, End>>;
#[session] type RingC = Receive<B, Value, Send<A, Value, End>>;

async fn ring_a(role: &mut A, input: i32) -> Result<i32> {
    let x = input;
    try_session(role, |s: RingA<'_, _>| async {
        let s = s.send(Value(x)).await?;
        let (Value(y), s) = s.receive().await?;
        Ok((x + y, s))
    }).await
}

async fn ring_b(role: &mut B, input: i32) -> Result<i32> {
    let x = input;
    try_session(role, |s: RingB<'_, _>| async {
        let (Value(y), s) = s.receive().await?;
        let s = s.send(Value(x)).await?;
        Ok((x + y, s))
    }).await
}

async fn ring_c(role: &mut C, input: i32) -> Result<i32> {
    let x = input;
    try_session(role, |s: RingC<'_, _>| async {
        let (Value(y), s) = s.receive().await?;
        let s = s.send(Value(x)).await?;
        Ok((x + y, s))
    }).await
}
```

Listing 5.1: Rumpsteak's Ring implementation.

```
fn main() {
    let Roles(mut a, mut b, mut c) = Roles::default();

    let input = (1, 2, 3);
    println!("input = {:?}", input);

    let output = executor::block_on(async {
        try_join!(
            ring_a(&mut a, input.0),
            ring_b(&mut b, input.1),
            ring_c(&mut c, input.2),
        )
        .unwrap()
    });
    println!("output = {:?}", output);
}
```

Listing 5.2: Rumpsteak's Ring `main` function.

5.1.2 Summary

As previously stated, Rumpsteak allows the developer to enforce communication patterns at compile-time, while `#[typestate]` does not, however, the latter is able to enforce behavior patterns at the participant level; `#[typestate]` is thus more flexible, being usable in synchronous or parallel programming, while Rumpsteak was designed with `async/await` in mind.

It is important to note that in both cases (Listing 5.1 and Listing 5.5) there is opportunity to improve modularity, one can use macros to abstract over the mostly repeated bits of code. For `#[typestate]`, for example, one can define a macro like Listing 5.6 and shorten the lines 17-28 in Listing 5.5 into lines 17 & 18 in Listing 5.7.

5.2 PIN

The PIN example has three principals, a card (Listing 5.9), read by the card reader (Listing 5.8), which is used by the final client.

To use the card, the reader ensures that the card is present by calling `check_for_card` (line 10 of Listing 5.8); in the case the card is not present the reader transitions to the `Error` state, otherwise, the reader transitions to the `CardPresent` state (line 13 of Listing 5.8). The `CardPresent` state not only requires a reference to the card, but also that the card itself is in the `Start` state (line 14 of Listing 5.8 and lines 8-13 of Listing 5.9). From the `CardPresent` state, the reader can issue an authentication operation, which is done by the card by checking the PIN against the on stored inside the card; if the authentication succeeds, the card reference is now required to reflect the reader's state, enforcing that the card is also in the `Authenticated` state (line 21 of Listing 5.8 and lines 15-19 of Listing 5.9); this relationship is visualized in Figure 5.3.

```rust
#[typestate]
mod ring_a {
    use std::sync::mpsc::{Receiver, Sender};

    #[automaton] pub struct RingA {
        pub(crate) send: Sender<i32>,
        pub(crate) receiver: Receiver<i32>,
    }

    #[state] pub struct SendA(pub i32);
    pub trait SendA {
        fn new(value: i32, send: Sender<i32>, receiver: Receiver<i32>) -> SendA;
        fn get_value(&self) -> i32;
        fn send(self) -> RecvA;
        fn end(self);
    }

    #[state] pub struct RecvA;
    pub trait RecvA { fn recv(self) -> SendA; }
}

impl SendAState for RingA<SendA> {
    fn new(value: i32, send: Sender<i32>, receiver: Receiver<i32>) -> Self {
        Self { send, receiver, state: SendA(value) }
    }

    fn get_value(&self) -> i32 { self.state.0 }

    fn send(self) -> RingA<RecvA> {
        self.send.send(self.state.0).unwrap();
        RingA::<RecvA> {
            send: self.send,
            receiver: self.receiver,
            state: RecvA,
        }
    }

    fn end(self) {}
}

impl RecvAState for RingA<RecvA> {
    fn recv(self) -> RingA<SendA> {
        let value = self.receiver.recv().unwrap();
        RingA::<SendA> {
            send: self.send,
            receiver: self.receiver,
            state: SendA(value),
        }
    }
}
```

Listing 5.3: #[typestate]'s implementation of participant A.

```rust
#[typestate]
mod ring_b {
    use std::sync::mpsc::{Receiver, Sender};
    #[automaton]
    pub struct RingB {
        pub(crate) send: Sender<i32>,
        pub(crate) receiver: Receiver<i32>,
    }

    #[state] pub struct SendB(pub i32);
    pub trait SendB {
        fn get_value(&self) -> i32;
        fn send(self) -> RecvB;
    }

    #[state] pub struct RecvB;
    pub trait RecvB {
        fn new(send: Sender<i32>, receiver: Receiver<i32>) -> RecvB;
        fn recv(self) -> SendB;
        fn end(self);
    }
}

impl SendBState for RingB<SendB> {
    fn get_value(&self) -> i32 { self.state.0 }

    fn send(self) -> RingB<RecvB> {
        self.send.send(self.state.0).unwrap();
        RingB::<RecvB> {
            send: self.send,
            receiver: self.receiver,
            state: RecvB,
        }
    }
}

impl RecvBState for RingB<RecvB> {
    fn new(send: Sender<i32>, receiver: Receiver<i32>) -> Self {
        Self { send, receiver, state: RecvB }
    }

    fn recv(self) -> RingB<SendB> {
        let value = self.receiver.recv().unwrap();
        RingB::<SendB> {
            send: self.send,
            receiver: self.receiver,
            state: SendB(value),
        }
    }

    fn end(self) {}
}
```

Listing 5.4: #[typestate]'s implementation of participant B.

```rust
fn main() {
    let (a_sender, b_receiver) = channel::<i32>();
    let (b_sender, c_receiver) = channel::<i32>();
    let (c_sender, a_receiver) = channel::<i32>();

    let a = RingA::<SendA>::new(0, a_sender, a_receiver);
    let b = RingB::<RecvB>::new(b_sender, b_receiver);
    let c = RingB::<RecvB>::new(c_sender, c_receiver);

    vec![
        thread::spawn(move || {
            println!("a: {}", a.get_value());
            let a = a.send();
            let a = a.recv();
            a.end();
        }),
        thread::spawn(move || {
            let b = b.recv();
            println!("b: {}", b.get_value());
            let b = b.send();
            b.end();
        }),
        thread::spawn(move || {
            let c = c.recv();
            println!("c: {}", c.get_value());
            let c = c.send();
            c.end();
        }),
    ]
    .into_iter()
    .map(|handle| handle.join())
    .collect::<Result<_, _>>()
    .unwrap()
}
```

Listing 5.5: The `main` implementing for `#[typestate]`'s ring.

```rust
macro_rules! spawn_ring_b {
    ($ring:ident) => {
        thread::spawn(move || {
            let $ring = $ring.recv();
            println!("{}: {}", stringify!($ring), $ring.get_value());
            let $ring = $ring.send();
            $ring.end();
        })
    };
}
```

Listing 5.6: `macro_rules!` to abstract over the thread launching routing for `RingB`.

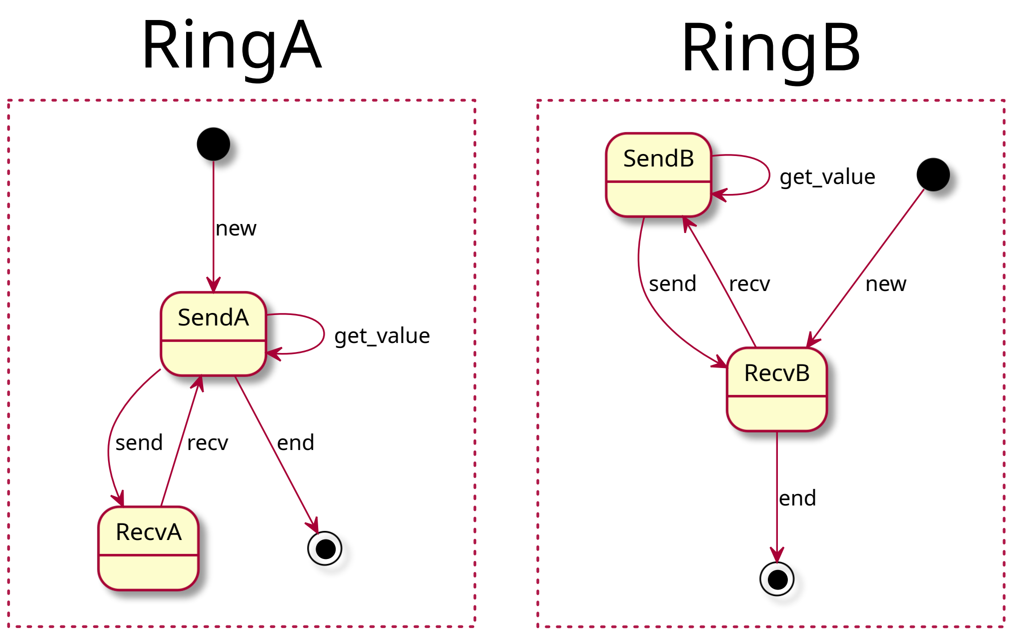


Figure 5.2: The ring participants' typestates.

```rust
fn main() {
    let (a_sender, b_receiver) = channel::<i32>();
    let (b_sender, c_receiver) = channel::<i32>();
    let (c_sender, a_receiver) = channel::<i32>();

    let a = RingA::<SendA>::new(0, a_sender, a_receiver);
    let b = RingB::<RecvB>::new(b_sender, b_receiver);
    let c = RingB::<RecvB>::new(c_sender, c_receiver);

    vec![
        thread::spawn(move || {
            println!("a: {}", a.get_value());
            let a = a.send();
            let a = a.recv();
            a.end();
        }),
        spawn_ring_b!(b),
        spawn_ring_b!(c),
    ]
    .into_iter()
    .map(|handle| handle.join())
    .collect::<Result<_, _>>()
    .unwrap()
}
```

Listing 5.7: The main implementing for #[typestate]'s ring using Listing 5.6.

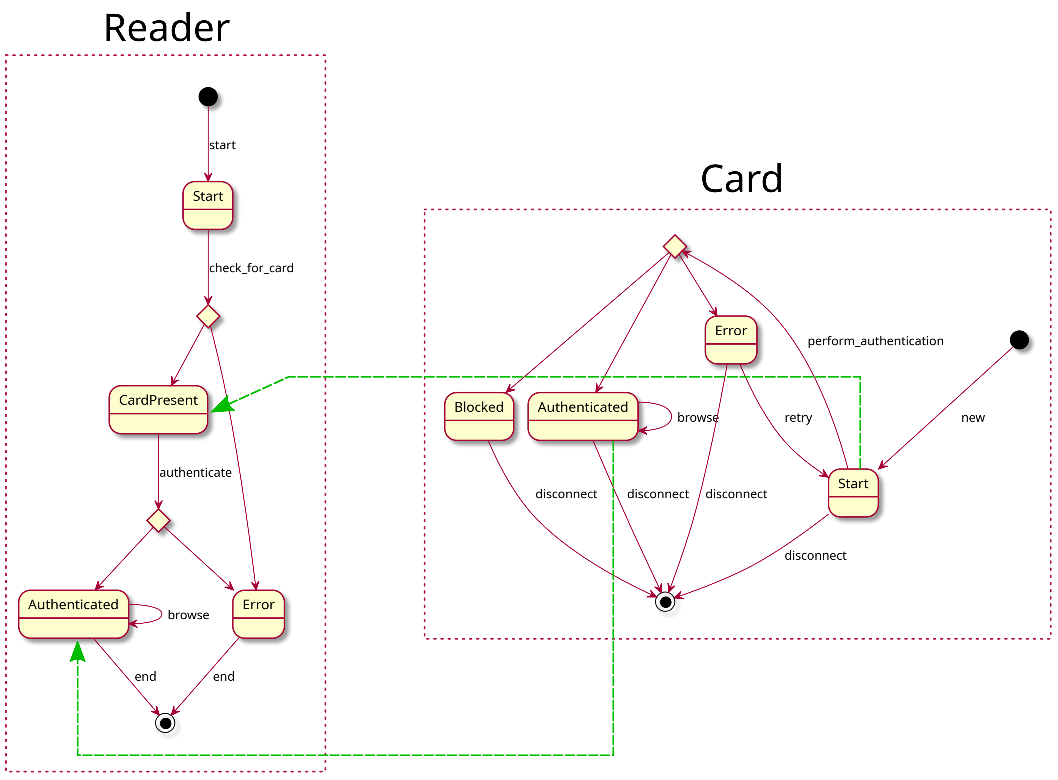


Figure 5.3: The `Reader` and `Card` typestates, the green arrows indicate the dependency relationship between states.

Any client of the reader API is thus required to check all steps before proceeding with the card. The state embedding also guarantees that the `Reader` is a well-behaved client of the `Card`, similar to Fugue [9].

5.3 Auction Client

This example showcases how one can build a typestated API on top of an existing *non-typestated* API, enhancing the guarantees provided by latter. Its goal is to ensure that the user does not perform non-optimal bids (i.e. only bids higher than the existing one) in the case the user is outbid by another, the client is then required to withdraw its bid before submitting another.

The auction API is listed in Listing 5.10, it allows bidding to be done and to emulate the closure of the auction with each bid a random boolean is generated, if the auction closes, bidding is closed; all checks are performed at runtime.

The client is required to check if the auction is running before any action can be performed. This is done through the `has_ended` function (line 13 of Listing 5.11), which returns the `AuctionState` state (line 46 of Listing 5.11), an enumeration representing the two possible outcomes of the transition; in case the auction has ended, the only thing

```rust
#[typestate]
pub mod reader_api {
    use crate::card::card_api;

    #[automaton] pub struct Reader;

    #[state] pub struct Start;
    pub trait Start {
        fn start() -> Start;
        fn check_for_card(self) -> CheckCardResult;
    }

    #[state] pub struct CardPresent {
        pub card: card_api::Card<card_api::Start>,
    }
    pub trait CardPresent {
        fn authenticate(self, pin: [u8; 4]) -> AuthResult;
    }

    #[state] pub struct Authenticated {
        pub card: card_api::Card<card_api::Authenticated>,
    }
    pub trait Authenticated {
        fn browse(&self);
        fn end(self);
    }

    #[state] pub struct Error {
        pub message: String,
    }
    pub trait Error {
        fn end(self);
    }

    pub enum CheckCardResult { CardPresent, Error }
    pub enum AuthResult { Authenticated, Error }
}
```

Listing 5.8: The Reader typestate specification.

```rust
#[typestate]
pub mod card_api {
    #[automaton] pub struct Card {
        pub valid_pin: [u8; 4],
        pub attempts_left: u8,
    }

    #[state] pub struct Start;
    pub trait Start {
        fn new() -> Start;
        fn perform_authentication(self, pin: [u8; 4]) -> AuthResult;
        fn disconnect(self);
    }

    #[state] pub struct Authenticated;
    pub trait Authenticated {
        fn browse(&self);
        fn disconnect(self);
    }

    #[state] pub struct Error;
    pub trait Error {
        fn retry(self) -> Start;
        fn disconnect(self);
    }

    #[state] pub struct Blocked;
    pub trait Blocked {
        fn disconnect(self);
    }

    pub enum AuthResult { Authenticated, Blocked, Error }
}
```

Listing 5.9: The `Card` typestate specification.

the user can do is end its "session"; otherwise the user will be placed in the `NoBids` state, as they have not submitted any bids. From the `NoBids` state, the user can perform a bid; in the case that the bid is not the highest, the user is forced to withdraw its bid; if the bid is the highest the user can then check if their bid is still the highest or if the auction has ended. In the case that the user's bid is no longer the highest, the user transitions into the `Withdraw` state again; in the case that the auction has ended, the user will then check if they have won the auction or not, and take the according action before ending the "session".

5.4 Summary

In this chapter I have presented three examples which highlight different capabilities of the `#[typestate]` macro. The first example (Section 5.1) shows how `#[typestate]` is

```rust
pub struct Auction {
    owner: String,
    bids: HashMap<String, u64>,
    highest_bid: u64,
    ended: bool,
}

impl Auction {
    pub fn new(owner: String) -> Self {
        Self {
            owner,
            bids: HashMap::new(),
            highest_bid: 0,
            ended: false,
        }
    }

    pub fn bid(&mut self, client: String, bid: u64) -> Option<()> {
        if self.owner != client && !self.has_ended() {
            self.ended = rand::random(); // simulate uncertainty
            self.bids.insert(client, bid);
            if self.highest_bid < bid {
                self.highest_bid = bid;
            }
            Some(())
        } else {
            None
        }
    }

    pub fn is_highest_bid(&self, client: &String) -> bool {
        match self.bids.get(client).map(|bid| self.highest_bid == *bid) {
            Some(is_highest) => is_highest,
            None => false,
        }
    }

    pub fn get_bid(&self, client: String) -> Option<&u64> {
        self.bids.get(&client)
    }

    pub fn has_ended(&self) -> bool { self.ended }
}
```

Listing 5.10: The auction *non-typestated* API.

```
#[typestate]
mod auction_client_api {
    use crate::auction::Auction;

    #[automaton] pub struct Client {
        pub(crate) name: String,
        pub(crate) auction: Auction,
    }

    #[state] pub struct AuctionRunning;
    pub trait AuctionRunning {
        fn start(name: String, auction: Auction) -> AuctionRunning;
        fn has_ended(self) -> AuctionState;
    }

    #[state] pub struct NoBids;
    pub trait NoBids {
        fn bid(self, bid: u64) -> BidStatus;
    }

    #[state] pub struct HasBidded;
    pub trait HasBidded {
        fn check_bid(self) -> BidStatus;
        fn has_ended(self) -> AuctionEnded;
    }

    #[state] pub struct CheckWinner;
    pub trait CheckWinner { fn is_highest_bid(self) -> WinnerStatus; }

    #[state] pub struct Withdraw;
    pub trait Withdraw { fn withdraw(self) -> AuctionRunning; }

    #[state] pub struct Lost;
    pub trait Lost { fn withdraw(self) -> End; }

    #[state] pub struct Winner;
    pub trait Winner { fn win(self) -> End; }

    #[state] pub struct End;
    pub trait End { fn end(self); }

    pub enum WinnerStatus { Lost, Winner }

    pub enum AuctionEnded { HasBidded, CheckWinner }

    pub enum AuctionState { NoBids, End }

    pub enum BidStatus { HasBidded, Withdraw }
}
```

Listing 5.11: The auction client's typestate declaration.

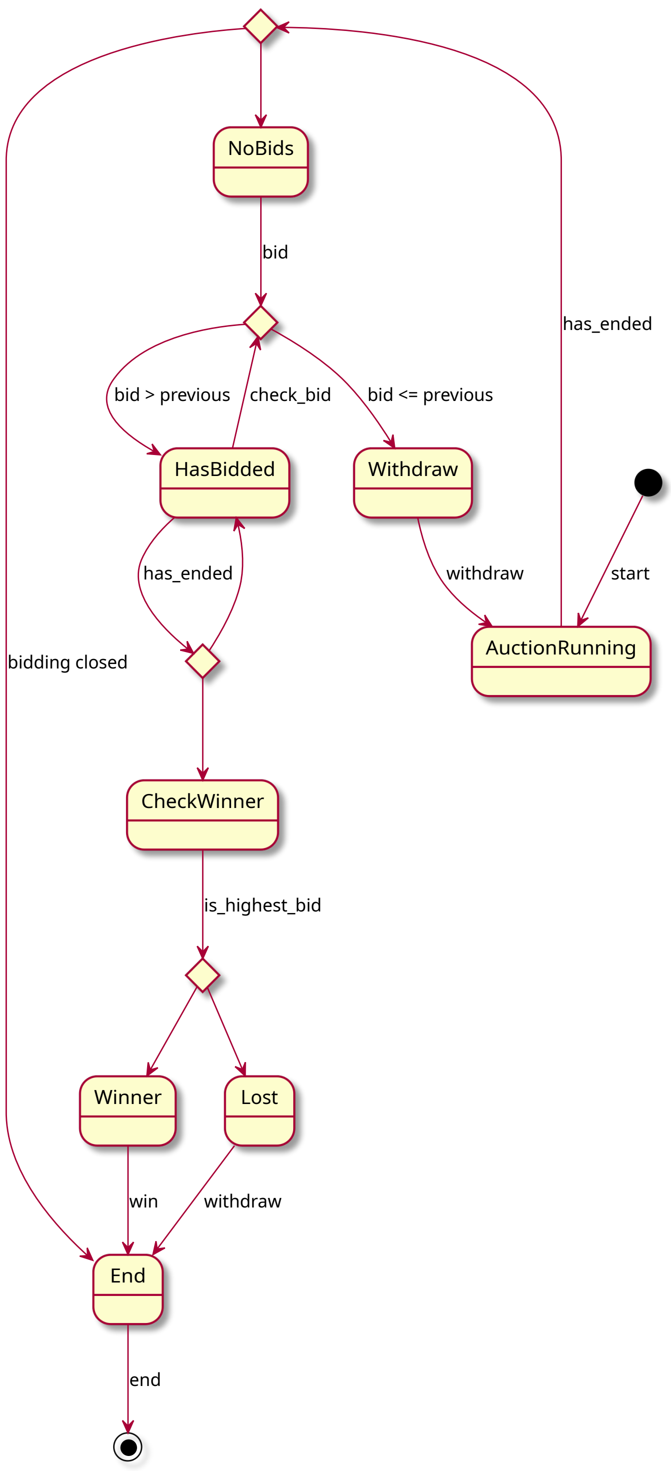


Figure 5.4: The auction client's typestate.

able to constrain the behavior of objects in multi threaded environments; the second example (Section 5.2) highlights how using multiple typestates in conjunction allows the developer to provide extra guarantees, especially when compared with non-typestated APIs; finally, the final example (Section 5.3) illustrates how one can wrap an existing API using `#[typestate]` to provide a safe-to-use API.

While in some cases one can further abstract over the code written using `#[typestate]`, such cases are often trivial to address using Rust's `macro_rules!`. Overall, `#[typestate]` is able to address the problem at hand and allow the developer to write an elegant solution while providing the desired API constraints.

6 Conclusions & Future Work

6.1 Summary

In this thesis a new tool is presented for the development of typestates in Rust, a language without first-class support for such concept. While it is possible to model typestates in plain Rust, it is cumbersome and error-prone; my work takes advantage of Rust's advanced type and macro systems to provide an embedded DSL which simplifies the development of typestates in Rust.

To the best of my knowledge, while similar works have been developed in other languages, the present work is the first one to leverage macros to automate and provide an elegant DSL along with extra features over Rust's existing typestate capabilities.

While the DSL is a single contribution, its parts constitute several points of improvement over the current *status quo*:

Tool independent DSL. The DSL requires no dependencies external to Rust itself, a user is able to start working with my work by simply adding a dependency to their project.

Verification tool. The macro executes several verifications over the declared typestate (discussed in Section 4.3).

Visualization tool. The macro behind the DSL also works as a visualization tool for the developed typestates (discussed in Section 4.4), it provides three formats, one of which can be embedded in the client's API, avoiding documentation rot.

The capabilities provided by the macro are useful for large systems and provide extra assurance during development, any system required to "get it right" can take advantage of them, such as embedded devices, which, with the rise of the Internet-of-Things are becoming ever more present in our lives. Typestates further allow reducing runtime state checks and easing the development process of large stateful systems, another possible use case are smart contracts, this idea is supported by the work on the Obsidian language [6, 5].

This work is open-source and made available on GitHub under the dual license APACHE/MIT, users can use the crate made available on `crates.io`. Following, are the relevant links to the project:

- **Repository** — `https://github.com/rustype/typestate-rs`
- **Documentation** — `https://docs.rs/typestate/0.7.2/typestate/`
- **Library** — `https://crates.io/crates/typestate`

6.2 Future Work

The current version of the macro suffers from some limitations: it does not support generics and by consequence, state machine nesting. Furthermore, the present work lacks formalization, relying on the Rust compiler to provide part of the formal guarantees; formalizing the present work represents an important step towards ensuring more guarantees that can be used in the "real-world".

While the previous points aim to provide extra features, the macro would also benefit from some work in its internals. Everything works as expected, but the architecture and code organization could be made cleaner by further dividing each visitor. Currently, some visitors perform mutations to the tree at the same time they extract relevant information; this process can be split into mutation and information extraction, easing code readability and maintenance.

Along with the visitor changes, the underlying graph structures could also be modified, using a single graph structure as the result of the information extraction process. The graph could then be "reduced" into several specialized use cases, such as state machine verification and visualization.

The two previous items, visitors and graph, can be leveraged to take the macro from a simple tool to an experimentation platform for typestates and automata verification in Rust.

Finally, planning for the development of such work in the future — I believe that starting with the re-architecture of the visitors followed by the graph would yield the best results, allowing changes in the future to be integrated in a simple manner; leaving the formal work and new features for a more stable and robust version of the macro.

Bibliography

[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 (cit. on pp. 5, 17, 19, 37).

[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 (cit. on pp. 4, 15, 18, 36).

[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 (cit. on p. 8).

[4] D. M. R. Brian W. Kernighan. *The ANSI C Programming Language*. 2nd ed. Prentice Hall, 1988. ISBN: 9780131103627,0131103628 (cit. on p. 7).

[5] 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 (cit. on pp. 5, 20, 81).

[6] 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 (cit. on pp. 5, 81).

[7] Z. Cutner and N. Yoshida. “Safe Session-Based Asynchronous Coordination in Rust”. In: *23rd International Conference on Coordination Models and Languages*. Springer, 2021, pp. – (cit. on pp. 36, 65).

[8] 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 (cit. on pp. 35, 38).

[9] 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/ (cit. on pp. 4, 36, 37, 73).

[10] T. Ekman and G. Hedin. "The JastAdd Extensible Java Compiler". In: 42.10 (2007), pp. 1–18. DOI: 10.1145/1297105.1297029 (cit. on pp. 23, 25).

[11] 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 (cit. on p. 36).

[12] S. Gay and A. Ravara. *Behavioural types: From theory to tools english*. 2017, pp. 1–375. ISBN: 9788793519817. DOI: 10.13052/rp-9788793519817 (cit. on p. 4).

[13] 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 (cit. on pp. 15, 35).

[14] K. Honda. "Types for dyadic interaction". In: *CONCUR'93*. Ed. by E. Best. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 509–523. ISBN: 978-3-540-47968-0 (cit. on pp. 4, 15).

[15] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language primitives and type discipline for structured communication-based programming". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1381 (1998), pp. 122–138. ISSN: 16113349. DOI: 10.1007/bfb0053567 (cit. on p. 15).

[16] 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 (cit. on p. 4).

[17] 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 (cit. on p. 14).

[18] 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 (cit. on pp. 5, 35, 36).

[19] 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 (cit. on p. 10).

[20] 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 (cit. on pp. 35, 36).

[21] 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 (cit. on pp. 4, 35, 38).

[22] Koxiaet. *How to Write Hygienic Rust Macros*. 2020. URL: https://gist.github.com/Koxiaet/8c05ebd4e0e9347eb05f265dfb7252e1 (visited on 01/18/2021) (cit. on p. 31).

[23] 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 (cit. on pp. 5, 36).

[24] 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) (cit. on p. 24).

[25] R. Macedo, A. M. Dias, and A. Ravara. "Visualização e animação de autómatos em Ocsigen Framework". In: *CoRR* abs/1907.05384 (2019). arXiv: 1907.05384. URL: http://arxiv.org/abs/1907.05384 (cit. on p. 55).

[26] J. Mota. "Coping with the reality : adding crucial features to a typestate-oriented language". PhD thesis. FCT-NOVA, 2020 (cit. on p. 20).

[27] R. Neykova et al. "A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#". In: *27th International Conference on Compiler Construction*. ACM, 2018, pp. 128–138. DOI: 10.1145/3178372.3179495 (cit. on p. 35).

[28] 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 (cit. on p. 35).

[29] R. Reis and N. Moreira. *FAdo: tools for finite automata and regular expressions manipulation*. 2002 (cit. on p. 56).

[30] R. E. Strom et al. *Hermes : A Tutorial and Reference Manual*. 1990. URL: https://researcher.watson.ibm.com/researcher/files/us-bacon/Strom90HermesTutorial.pdf (cit. on p. 17).

[31] 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 (cit. on pp. 4, 17).

[32] B. Stroustrup. *The C ++ Programming*. 1986, p. 1346. ISBN: 978-0321563842 (cit. on p. 8).

[33] *The Rust Programming Language*. 2021. URL: https://doc.rust-lang.org/book/ (visited on 01/18/2021) (cit. on pp. 11, 13).

[34] *The Rust Reference*. 2021. URL: https://doc.rust-lang.org/reference (visited on 01/18/2021) (cit. on pp. 30, 33).

[35] 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 (cit. on pp. 7–9).

[36] A. Trindade, J. Mota, and A. Ravara. "Typestates to automata and back: A tool". In: *Electronic Proceedings in Theoretical Computer Science, EPTCS* 324.Ice (2020), pp. 25–42. ISSN: 20752180. DOI: 10.4204/EPTCS.324.4 (cit. on pp. 42, 52).

[37] 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 (cit. on p. 35).

[38] 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 (cit. on pp. 15, 16).

[39] 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 (cit. on pp. 4, 18, 35, 38).

[40] L. Wirth. *The Little Book of Rust Macros*. 2021. URL: https://veykril.github.io/tlborm (cit. on p. 30).

[41] H. Xi and H. Wu. "Linearly Typed Dyadic Group Sessions for Building Multiparty Sessions". In: (2016). arXiv: 1604.03020 (cit. on p. 4).

[42] H. Xi et al. "Session Types in a Linearly Typed Multi-Threaded Lambda-Calculus". In: (2016). arXiv: 1603.03727 (cit. on p. 4).

[43] 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 (cit. on pp. 4, 35, 36).