José Miguel Gonçalves Duarte

Master of Science

A Very Long and Impressive Thesis Title with a Forced Line Break

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

Master of Science in
Computer Science and Engineering

Adviser: António Ravara, Full Professor, NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

⟨month⟩, ⟨year⟩

Abstract

The dissertation must contain two versions of the abstract, one in the same language as the main text, another in a different language. The package assumes that the two languages under consideration are always Portuguese and English.

The package will sort the abstracts in the appropriate order. This means that the first abstract will be in the same language as the main text, followed by the abstract in the other language, and then followed by the main text. For example, if the dissertation is written in Portuguese, first will come the summary in Portuguese and then in English, followed by the main text in Portuguese. If the dissertation is written in English, first will come the summary in English and then in Portuguese, followed by the main text in English.

The abstract shoul not exceed one page and should answer the following questions:

- What's the problem?
- Why is it interesting?
- What's the solution?
- What follows from the solution?

Keywords: Keyword 1, Keyword 2, Keyword 3, . . .

Resumo

Independentemente da língua em que está escrita a dissertação, é necessário um resumo na língua do texto principal e um resumo noutra língua. Assume-se que as duas línguas em questão serão sempre o Português e o Inglês.

O *template* colocará automaticamente em primeiro lugar o resumo na língua do texto principal e depois o resumo na outra língua. Por exemplo, se a dissertação está escrita em Português, primeiro aparecerá o resumo em Português, depois em Inglês, seguido do texto principal em Português. Se a dissertação está escrita em Inglês, primeiro aparecerá o resumo em Inglês, depois em Português, seguido do texto principal em Inglês.

O resumo não deve exceder uma página e deve responder às seguintes questões:

- Qual é o problema?
- Porque é que ele é interessante?
- Qual é a solução?
- O que resulta (implicações) da solução?

E agora vamos fazer um teste com uma quebra de linha no hífen a ver se a LaTeX duplica o hífen na linha seguinte...

zzzz zzz zzzz zzz zzzz zzz zzzz zzz zzzz zzz zzzz zzz zzzz zzz zzzz zzz zzzz comentar--lhe zzz zzzz zzz zzzz

Sim! Funciona! :)

Palavras-chave: Palavra-chave 1, Palavra-chave 2, Palavra-chave 3, ...

Contents

List of Figures

List of Tables

Listings

Glossary

Acronyms

p4	Pre-Processor-Pretty-Printer	19
PPX	PreProcessor eXtensions	19

Symbols

Chapter 1

Introduction (20/01/2021)

1.1 Context

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

In systems programming, one of the most demanding domains in computer science, bugs and their respective consequences come at a high cost to both service providers and consumers. There are reports from several industries where bugs lead to huge monetary losses and even death. In 2014, the Heartbleed [17] 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 [53]. In 2019 and 2020, after several crashes [5], the Boeing 737 Max was grounded to fix existing problems. While grounded, more software-related issues were found [33, 32], delaying its re-certification. In 2020, as the number of COVID-19 grew, contact tracing apps were deployed as a mitigation strategy. The UK's National Health Service app failed to ask users to self-isolate due to a bug [28].

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

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

Listing 1.1 – Java's null reference example.

1.2 Problem

Languages like C/C++ have dominated the systems programming landscape for years and one of the main problems with both is the lack of memory management. Leaving such responsibility to the developer has proven to be *a less than ideal* solution, with 70% of bugs in projects like Chromium [29] and Microsoft products [30] being due to memory management.

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 its 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, and so we are required to reach out to new mechanisms.

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

1.2.1 The Billion Dollar Mistake

> *This led me to suggest that the null value is a member of every type, and a null check is required on every use of that reference variable, and it may be perhaps a billion dollar mistake. — [19]*

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

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

```
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.

```
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.

1.2.2 API Misuse

Consider Java's Scanner, its 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 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.

If we consider a typestated language, as in Listing 1.3, 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*.

1.3 State of the Art

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 Rust's ecosystem a crate providing typestated futures exists, the state_machine_future crate [14]. It provides 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 all state transitions are invalid. Furthermore, these guarantees are provided at compile-time, invalid state transitions, for example, fail to compile. The crate, however, revolves around futures, requiring a 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 Rust, current mainstream languages do not provide first-class support for behavioral types, leaving developers to write their own abstractions over the language type-system. These abstractions will usually be built on top of existing meta-programming capabilities offered by the language, or with the help of external tools in an ad-hoc fashion. The latter is the most common approach as it allows the built tool to take advantage of other existing tools. This is Mungo's case [24, 54], a language which generates a Java API skeleton along with a typestate specification from a Scribble [58] protocol.

Amidst the previous topics, Scribble and Rust's crates, there exists a third alternative, rooted in session types. The work done by [22, 31] introduces bi-directional session types to Rust, since then, this line of work has been expanded by [27], extending it to multiparty session types.

Regarding typestates, languages like Plaid [1] and Obsidian [6, 7] put typestates to use. Plaid in an object-oriented context while Obsidian makes use of typestates and linear types to provide more safety when writing blockchain smart contracts. While the Plaid project is considered to be done, Obsidian is relatively new and possibly able to provide key insights into the way we write code [6].

1.4 Objectives & Contributions

In this thesis I try to bridge the gap between typestates and Rust, aiming for an elegant and usable solution, allowing for effective usage of typestates in Rust. To achieve such solution I expect to develop an embedded Rust DSL, enabling the flexibility of a dedicated language without leaving the Rust ecosystem. To this effect I expect the contributions of this thesis to be a typestate specification DSL to be embedded in Rust, this topic is further developed in Chapter 4.

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

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

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

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

1.5 Report Organization

This document is organized as follows:

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

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

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

Chapter 2

Background (03/02/2021)

2.1 Systems Programming Languages

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

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

2.1.1 C

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

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

Its 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 mimic its performance.

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

2.1.2 C++

Introduced in 1985 as an extension to C; its 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.*
>
> — *[45, 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++ has its problems. The language is enormous, with very complicated parts (e.g. templates) and compilation for big projects is very slow, the author acknowledges this in [50]. Furthermore, as the language provides a high level of control over the system, it has manual memory management, suffering from the same problems as C. Even with smart pointers (e.g. `unique_ptr`) the problem is not considered solved, as they introduce overhead in the most demanding applications.

2.1.3 Ada

Ada was developed in 1980, during a standardization effort in the USA's Department of Defense, with the goal of unifying projects spanning over 450 programming languages [46]. Its main focus was the development of embedded applications, currently the Ada language is mostly used in the critical domain due to its strong emphasis on safety, some Ada success stories are the London Metro Victoria Line and the Paris Metro Line [40]. Ada is also used in several other domains, such as aviation, space vehicles, financial systems and more [13].

In comparison with the other languages in this section, Ada is eclipsed, barely showing in the GitHub rankings [34]. However, given that Ada's compiler is mostly a product, it makes sense that most Ada code is not open-source. Regardless, when one views the list of features Ada has, the first arising question is "*why is Ada not popular?*".

An old article in AdaPower [16] provides some possible insight over the question, referring to the compiler's price and the Hoare's harsh critics. From my point of view, the critics to the compiler and ecosystem pricing still make sense, as access to the full tooling is limited. The lack of programmers goes on to deepen the lack of adoption in the industry and this cycle ends up limiting Ada's reach in the market.

2.1.4 Go

The Go programming language (or `golang`) is a Google project, according to its tale, it was designed by the authors while they waited for their C++ code to compile. Go tried to address several of the criticisms to C, namely memory management, which it solved by using a garbage collector. While it has made a name for itself in the network and distributed systems sector, being the main language behind projects like Docker [10] and Kubernetes [26], its position as a systems programming language can be discussed.

Sometimes, however, the performance might not be enough, as was the case for Discord, the popular internet voice server company, as demand increased, Go was not able to meet the expected performance requirements and the company replaced it with Rust [20]. In [50], one of Go's authors, Rob Pike, says that he regrets categorizing Go as a systems programming language, being rather a server programming language that evolved into a cloud infrastructure language. Regardless of discussion, Go has proven to be a viable alternative to its counterparts, compromising extreme performance in name of safety and simplicity.

2.1.5 Summary

I reviewed four system programming languages, suited for different kinds of environments, C, C++ and Ada can be considered the traditional system languages kind, with a strong emphasis on efficiency and support for embedded devices. Go on the other hand, could be considered a new generation systems programming language, a language for cloud infrastructure. Among the four, only Ada places strong emphasis on safety, with several features allowing for more guarantees at compile time, such as contract based programming and non-nullable types by default, being the only one which does not suffer from the "*billion dollar mistake*" [19].

2.2 The Rust Language

Rust is a fairly recent systems programming language, it started as a side project of the author Graydon Hoare, its public history dates back to 2010 [18]. 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 [23].

2.2.1 What makes Rust different?

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

The key to all the promises Rust makes lies in its ownership system and the 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 problem, the learning experience. While Rust has become more accessible over the years, ownership and the borrow checker still require some effort on the part of the developer to learn. I provide a small overview of ownership, the borrow checker and their part in Rust's promise of "*fearless concurrency*".

2.2.2 Ownership

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

> - *Each value in Rust has a variable that's called its owner.*
> - *There can only be one owner at a time.*
> - *When the owner goes out of scope, the value will be dropped.*
>
> — *[48, Section 4.1]*

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

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

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

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

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

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

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

2.2.3 Borrowing

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

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

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

```rust
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 using borrowing to allow for more than one reader on the same variable.

```rust
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 immutable borrowed in the previous lines
```

Listing 2.4 – Example using borrowing to allow for more than one reader on the same variable.

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

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

2.2.4 Concurrency

> *Initially, the Rust team thought that ensuring memory safety and preventing concurrency problems were two separate challenges to be solved with different methods. Over time, the team discovered that the ownership and type systems are a powerful set of tools to help manage memory safety and concurrency problems! By leveraging ownership and type checking, many concurrency errors are compile-time errors in Rust rather than runtime errors. — [48, Section 16]*

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

Message-passing

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

> *Do not communicate by sharing memory; instead, share memory by communicating. — [12, Concurrency]*

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

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

Shared-state

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

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

2.2.5 Why Rust instead of Language X?

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

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.

2.3 Behavioral Types

As previously discussed, with the growth in software complexity, developers are required to develop better tools to tame such complexity. While the tools are "physical"in the sense that they are programs, they will most likely have a theoretical background. Behavioral types are part of such background. The theory behind them encompasses several domains, they can be applied over a wide range of entities, from an object to a web service.

> *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.

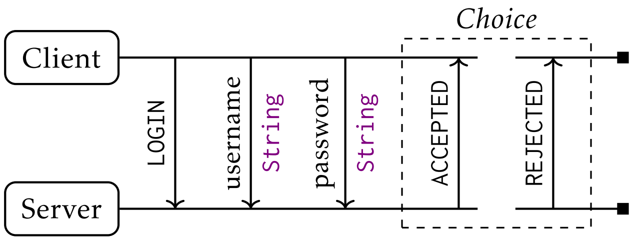

Figure 2.1 – Communication protocol example. The communication establishment step is omitted for simplicity. In this protocol the client tries to login to a service by sending a message `LOGIN` followed by the username and password, both of type `String`. The server then replies with either an `ACCEPTED`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

A developer can use it as a specification (for simplicity consider the uppercase messages to be simple strings), using behavioral types the developer could be able to specify the described interactions and all boilerplate could be generated for him. While using strings is not very interesting "*interesting*", if we consider the object in transit to be an encrypted payload, the boilerplate will take care of decryption and deserialization. Furthermore, consider the constraint that *all interactions end with a message from the server*. If the specification has an interaction that is not compliant with such rule, the code should not compile, raising an error.

2.3.1 Session Types

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

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

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

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

No message matching required, the compiler does it for the developer. Using session

```
enum Request {
    Login(String, String),       // login with: {username, password}
    SendMessage(String, String),// send message to an user: {user, message}
}
enum Reply {
    AuthOk,      // successful login
    AuthErr,     // failed login
    MessageOk,  // successful message
    MessageErr, // failed message
}
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.

types it is possible to write it in a simpler form, where each endpoint has its type:

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

Consider ! to be *sends*, ? to be *receives*, . the *sequence* operator finally, & the *choice offering* and ⊕ the *choice making*. Using session types effectively offloads complexity to the type system, resulting in more complex types, but simpler implementations, since protocol compliance can be checked during compilation and boilerplate can be added by the compiler.

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.* — [44]

The first language to make use of typestates was NIL [44], afterwards languages like Hermes [43] and Plaid [1] extended the concept with new techniques.

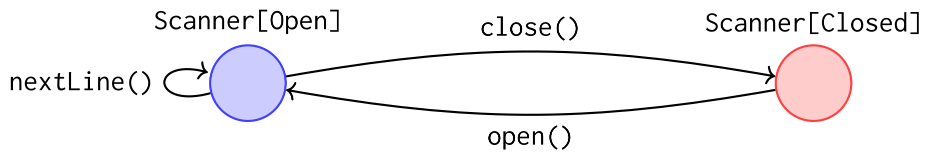

Figure 2.2 – The Scanner typestate automata, based on Listing 1.3.

Automata

A possible question on the reader's mind is *"how are automata and typestates related?"*. This section tries to address that question and exemplify how automata helps prove typestates properties. It is possible to express typestates as automata, as the reader can observe in Figure 2.2. Each state is a possible state the object can be in, transitions are done through methods. Methods can either mutate the object state, such as `open` and `close`, or leave it unchanged, such as `nextLine`.

Real-world scenario. In production applications, the API is not this simple. In fact, the `Scanner` API is not this simple, however it was reduced to the simplest possible 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. Details like those can be costly, imagine for example that a method call reaches a state, which as no outgoing edges, but it is not a final state. This is a problem addressed by existing automata algorithms, thus representing typestates as automata, extracting all necessary information and applying such algorithms provides the API extra safety.

The case for typestates

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

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

Static typed languages. The current trend is to move from dynamically typed languages, to statically typed ones, or at the very least, add typing support to existing dynamic languages. Typescript [51], Reason [37] and PureScript [35] are all examples of languages

```
1 public class Mult {
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 `Mult` program, which reads two integer and multiplies them together.

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 [52, 39].

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 [54], 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 [11].

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

Typestates in action

As a simple example, consider the Java application in Listing 2.6 which simply takes two numbers and multiples them together. The application will throw an exception on line 6, since the programmer closed the `Scanner` in line 5. In this example, the error is simple to catch, the program is short and the `Scanner` can either be open or closed, however, real-world applications are not that simple.

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

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

Plaid is a typestate-oriented programming language [1], instead of `class`es users write `typestate`s. Each typestate represents a class in its possible states, its methods and

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

Listing 2.7 – The `Mult` program, written in a typestated fashion.

behavior change during runtime as state changes, in contrast with other languages (e.g. Java) where public methods and fields are always available.

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

Rust. As discussed in Section 2.2, Rust takes its 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 its type system (as demonstrated in [11]), this is discussed in further sections of this document.

Embedded Rust. As any systems programming language, Rust penetrated the embedded development space. Its features are most adequate and the community has put great effort into making Rust a viable language for embedded systems.

The Embedded Rust Book's [47] Chapter 4 is dedicated to static guarantees, introducing programmers to the concepts of typestate in Section 4.1, and their usage in embedded systems.

As for real-world usage, typestates are abundantly used in the area (not just discussed in the book), under [41] one finds several repositories (suffixed with `-hal`) which implement typestates (e.g. `gpio.rs` from `stm32h7xx-hal`).

Obsidian is a language targeting Hyperledger Fabric [15], among other features it makes use of typestates to reduce the amount of bugs when dealing with assets.

In [6] 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.

Chapter 3

Related Work (10/02/2021)

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, however, previous to version 4.02, OCaml made use of p4.

We briefly review both p4 and PPX.

Camlp4

Camlp4 is a parsing library which provides extensible grammars, its main goal is to allow users to extend OCaml syntax, Camlp4 is also able to redefine the core syntax, OCaml even introduced a revised syntax [36] to enable Camlp4.

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 [55].

In a nutshell, the Camlp4 library would allow developers to develop an extension syntax, when the compiler would pass the source code as text to the preprocessor, which, in turn would generate valid OCaml source code.

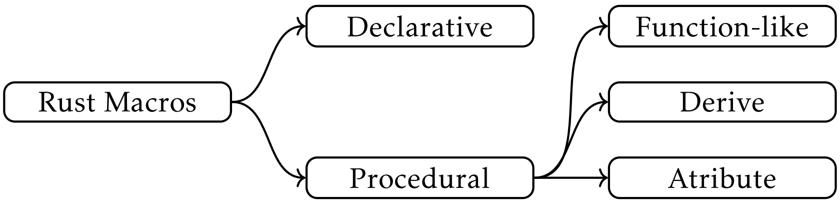

Figure 3.1 – Rust macro's family tree

PPX

3.1.2 Java

As other languages, Java is also capable of source code processing during compile time, we review two existing approaches, annotations and the ExtendJ compiler.

Java Annotation Processor

Java annotations were first introduced in Java 5 [9], 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 [8].

ExtendJ & JastAdd

3.1.3 Kotlin

Kotlin Compiler Plugins

3.2 Rust Macros

Just like its predecessors, C & 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 3.1), *declarative* macros and *procedural* macros.

3.2.1 Declarative Macros

Declarative macros (also known as *macros-by-example*) can be declared with `macro_rules!` and are called with function syntax (see Listing 3.1).

> *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. — [49, 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 [49, 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 [49, Section 3.1 - Metavariables]. In Listing 3.1, the metavariable `n` is of kind `literal` which will match literals such as `'E'`, `"Elite"` and `420` [49, 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 [56]. Identifiers are compared over two pieces of information, the *textual value* and their *syntactic context*. The textual value consists of the variables name (e.g. `four`), the syntactic context is a kind of scope added to variables declared inside the macro. This is done to keep the macro declared variables from interfering with existing ones.

When expanding a declarative macro[1] variables declared inside the macro belong in a different scope, consider the macro declared in Listing 3.2 and the respective expansion in Listing 3.3. As illustrated by Listing 3.3, line 2 is considered to be in a different context than the rest of the expanded code. This will rightfully raise an error (shown in Listing 3.4), since line's 3 `a` will not exist due to not being in the same syntactic context as line 2.

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.

[1] The same mechanism does not apply to procedural macros, which are not hygienic. Their output will interfere with existing code if precautions are not taken [25].

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

Listing 3.1 – Example `macro_rules!` usage. When executed, the code above will print "`Hello, world!`" five times.

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

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

Listing 3.3 – Listing 3.2 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.

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

Listing 3.4 – The expansion in Listing 3.3 will result in an error during compile time since the `a`s 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 3.5 – HTML DSL embedded in Rust. Example taken from [57].

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 [57, 42] (see the example in Listing 3.5) and the possibility to run Python inside Rust[21].

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 3.6). They define new inputs for the `derive` attribute, and can also create new items given the token stream of a `struct`, `enum` or `union`.

Definition. Just like function-like macros, derive macros are defined as a public function with the `proc_macro_derive` attribute and a signature of `(TokenStream) -> TokenStream`. The input is a token stream of the item with the `derive` attribute, the output is a set of items that are appended to the module or block where the input token stream is in. In Listing 3.6 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 3.7).

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

Listing 3.6 – 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 3.7 – Example usage of a derive macro with helper attributes, in this case the error(...) defines an error message with a Coordinates parameter.

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

Listing 3.8 – Attribute macros are commonly used in web frameworks to provide an easy way to declare an endpoint. In this example (taken from [38]) the user declares that GET requests to hello/ have two path parameters (name and age) and should be handled by the hello function.

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

Definition. Like its counterparts, 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 3.8 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).

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

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

3.3.1 Papers do NOW (Ancona)

3.3.2 Mungo

3.3.3 Session Types in Rust

Chapter 4

Planning (17/02/2021)

4.1 The DSL

4.1.1 Objectives

What the DSL should achieve

4.1.2 Architecture

How it achieves it

References

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

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

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

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

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

[6] 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.

[7] 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.

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

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

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

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

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

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

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

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

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

[17] *Heartbleed Bug*. Jan. 2021. URL: https://heartbleed.com/.

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

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

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

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

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

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

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

[34] T. Palmer. *tjpalmer/languish*. URL: https://tjpalmer.github.io/languish/ (visited on).

[35] *PureScript*. Jan. 2021. URL: https://www.purescript.org/.

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

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

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

[39] *ruby/rbs*. Ruby. Jan. 2021. URL: https://github.com/ruby/rbs.

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

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

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

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

[44] 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.

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

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

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

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

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

[50] 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.

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

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

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

[54] 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.

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

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

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

[58] 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.