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	7
PPX	PreProcessor eXtensions	7

Symbols

CHAPTER 1

Introduction (20/01/2021)

1.1 Context

Users face bugs on a daily basis, whether in their instant messaging application or a game, bugs currently permeate our lives, they are mostly harmless as such applications are not critical, resulting in some unsent messages or texture glitches.

However, in areas as 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 [8] 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 [25]. In 2019 and 2020, after several crashes [2], the Boeing 737 Max was grounded to fix existing problems, while grounded, more software-related issues were found [13, 12], 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 [9].

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.

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 [10] and Microsoft products [11] 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.

1.2 The Rust Language

Rust is a fairly recent systems programming language, its main focus revolves around memory safety, effectively removing classes of such bugs (e.g. *use-after-free* and *double-free*). Another one of Rust's focus, is on productivity, aiming to provide the safety mechanisms necessary to remove the previous class of bugs, while trying to provide a pleasant and productive development experience.

To achieve its goals, Rust makes use of a borrow checker and an ownership system, in conjunction, they're responsible for guaranteeing correct memory usage. At its core, the borrow checker is a lightweight theorem prover, it tries to prove that the code does not break safety rules.

Its rules can be distilled down to the following intuition, only one entity can hold a reference to mutable data at a time, several entities may hold references to immutable data. This also enables Rust to also provide mechanisms which help deal with concurrency, allowing for developers to write data-race free code [22].

1.3 Typestates

Typestates were introduced in [20], they are a subset of behavioral types and according to the same article, typestates are a *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*.

The first language to make use of typestates was NIL [20], afterwards languages like Hermes [19] and Plaid [1] extended the concept with new techniques.

1.3.1 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 [23], Reason [16] and PureScript [14] 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 [24, 17].

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 [26], 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 [6].

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.

1.3.2 Typestates in action

As a simple example, consider the Java application in Listing 1 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 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.

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

Listing 1 – The `Mult` program, which reads two integer and multiplies them together.

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

Listing 2 – The `Mult` program, written in a typestated fashion.

Plaid

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 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 1.2, 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 [6]), 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 [21] 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 [18] one finds several repositories (suffixed with `-hal`) which implement typestates (e.g. `gpio.rs` from `stm32h7xx-hal`).

Obsidian

Obsidian is a language targeting Hyperledger Fabric [7], among other features it makes use of typestates to reduce the amount of bugs when dealing with assets.

In [3] 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 2

Related Work (03/02/2021)

2.1 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.

2.1.1 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 [15] 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 [27].

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.

PPX

2.1.2 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 [5], 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 [4].

ExtendJ & JastAdd

2.1.3 Kotlin

Kotlin Compiler Plugins

2.1.4 Rust

2.2 Typestate-oriented approaches

2.2.1 Plaid

2.2.2 Obsidian

Maybe more

Chapter 3

Designing a DSL in Rust (10/02/2021)

3.1 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, they generate code before compilation. In Rust, macros refer to a family of features (see Figure 3.1), *declarative* macros and *procedural* macros.

3.1.1 Declarative Macros

Declarative macros are also known as *macros-by-example*, they can be declared with `macro_rules!` and are called with function syntax (see Listing 3). 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.

Transcribing. When a macro is invoked, the macro expander looks up macro invocations by name, and tries each macro rule in turn. It transcribes the first successful match;

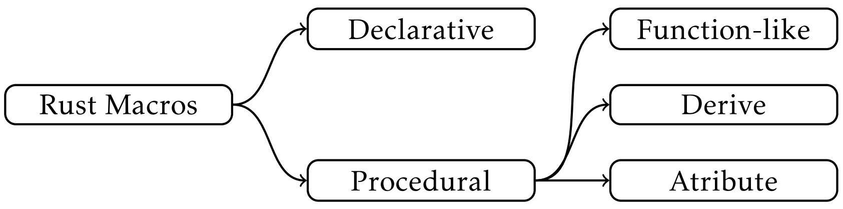

Figure 3.1 – Rust macro's family tree

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

Listing 3 – Example `macro_rules!` usage. When executed, the code above will print “`Hello, world!`” five times.

if this results in an error, then future matches are not tried. When matching, no lookahead is performed; if the compiler cannot unambiguously determine how to parse the macro invocation one token at a time, then it is an error.

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. In Listing 3, the metavariable `n` is of kind `literal` which will match literals such as `'c'`, `"String"` and `1337`.

Repetitions. 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. Declarative macros are *partially* hygienic, meaning they are hygienic when it comes to most identifiers, but not when it comes to generic type parameters or lifetimes. Hygiene works by attaching an invisible "syntax context"value to all identifiers. When two identifiers are compared, both the identifiers’ textural names and syntax contexts must be identical for the two to be considered equal.

3.1.2 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.

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

(a) using_a macro definition.

```rust
let four = {
    let a = 42;
    a / 10
};
```

(b) using_a macro expansion. Declarations with a blue background will be placed in a different *scope* than the others.

Listing 4 – Hygienic macro expansion.

Function-like

Derive

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 5). 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. Derive macros are defined as public functions 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 5 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 6).

Hygiene

In contrast with `macro_rules!`, procedural macros are *unhygienic*, meaning their output can interfere with the surrounding code and vice-versa, like C macros, they act as if the output code was simply written in the original source file. This leads developers to be required to add a series of extra measures when outputting code such as using absolute paths and function names which are unlikely of clashing with user code.

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

Listing 5 – 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 6 – Example usage of a derive macro with helper attributes, in this case the `error(...)` defines an error message with a `Coordinates` parameter.

Attribute

3.2 The DSL

3.2.1 Objectives

What the DSL should achieve

3.2.2 Architecture

How it achieves it

CHAPTER 4

PLANNING (17/02/2021)

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/$%5Csim$aldrich/papers/onward2009-state.pdf%20http://dl.acm.org/citation.cfm?doid=1639950.1640073.

[2] 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.

[3] 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.

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

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

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

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

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

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

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

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

[20] 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.

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

[22] A. Turon. *Fearless Concurrency with Rust - Rust Blog*. 2015. URL: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html.

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

[24] *typing — Support for type hints — Python 3.9.1 documentation*. URL: https://docs.python.org/3/library/typing.html.

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

[26] 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.

[27] 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).