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

Examination Committee:

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	5
PPX	PreProcessor eXtensions	5

Symbols

Chapter 1

Introduction

1.1 Typestates

Typestates are a subset of behavioral types, the core concept behind them is the elevation of state to the type system. They were first introduced by [NLL/Hermes], recently typestates have been further studied by [Aldrich].

Add references to NLL, Hermes and Plaid

We now present a shallow review over their usefulness, possible approaches to typestates and challenges faced when adding typestates to a language.

1.1.1 Why are Typestates useful?

Typestates allow the developer to provide more expressive APIs, as well as securer ones. Currently, in *typestateless* programming, programmers are required to track object state in their heads, with the help of comments, the code itself and possibly the debugger, running the code and checking their assumptions by hand.

As a simple example, consider the Java application in Figura 1.1 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*.

Figura 1.2 shows the `Mult` program written in a typestated fashion, notice that the `Scanner` type is now augmented with its state.

```java
public class Mult {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int lhs = Integer.parseInt(s.nextLine());
        s.close();
        int rhs = Integer.parseInt(s.nextLine());
        System.out.println(lhs * rhs);
    }
}
```

Figure 1.1: 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);
        int lhs = Integer.parseInt(s.nextLine());
        Scanner[Closed] s = s.close();
        int rhs = Integer.parseInt(s.nextLine());
        System.out.println(lhs * rhs);
    }
}
```

Figure 1.2: The `Mult` program, written in a typestated fashion.

1.1.2 Approaches

As previously discussed, simply put, typestates are the elevation of state to the type system. Currently, there are three main approaches to typestates, some can be retrofitted into existing languages.

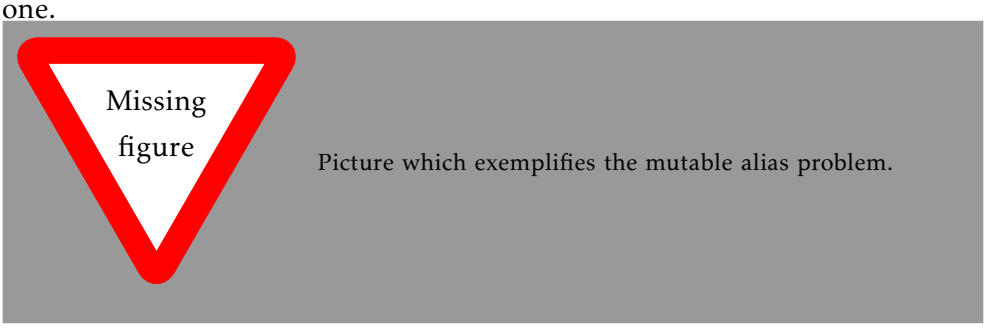
State-Based Design This is the most common approach, able to be used in any language, essentially, all responsibility is on the developer, which is required to track state at runtime by adding state checking barriers (e.g. `if CurrentState == DesiredState`). In such case, state modelling will be done using enumerations or the language equivalent.

Typestate Checkers Typestate checkers are external tools to the language, they may take the form of an independent tool or a compiler plugin. Their goal is to run as part of the compilation stage and enforce the desired state transitions in the code. The biggest problem with tools like these, is the added complexity of extra tools over the build process.

Language-Based The language-based approach is the most solid approach to the problem. At a conceptual level it is also the simplest approach, instead of adding features to existing languages just to enable typestate support, it prioritizes typestates and makes them a first-class language construct. This approach is the one taken by languages such as [Plaid]. Add ref

1.1.3 Challenges

Typestates are not without their challenges, attentive readers might notice that if objects are mutably aliased, typestate guarantees cannot hold since the object can be mutated, changing state while a client depends on the previous one.



Picture which exemplifies the mutable alias problem.

Solving this problem requires objects to be owned by a single client, this is due to the fact that even if an immutable alias exists, its type may be mutated by the original owner. Verify this with the advisor

Such solution requires the type system to be at least affine, being able to enforce a *use at most once* policy over the language objects.

1.2 The Rust Language

1.2.1 Why Rust?

Chapter 2

Related Work

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[1] 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[2].

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.

[1] https://caml.inria.fr/pub/docs/manual-camlp4/manual007.html
[2] https://whitequark.org/blog/2014/04/16/a-guide-to-extension-points-in-ocaml/

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 (JSR 269), 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.

ExtendJ & JastAdd

2.1.3 Kotlin

Kotlin Compiler Plugins

2.1.4 Rust