Implementación Básica de Typestates en Rust

Marcelo Arroyo, Simón Gutiérrez Brida, Pablo Ponzio

Dpto de Computación, FCEFQyN, Universidad Nacional de Río Cuarto

Resumen Generalmente la API de un módulo describe las operaciones disponibles, aunque el orden lícito de aplicación de las mismas queda implícito o documentado externamente debido a que los lenguajes de programación generalmente no proveen mecanismos de especificación del *protocolo* de uso.
Typestates permite especificar *estados* de objetos de un determinado tipo. Cada estado habilita ciertas operaciones y prohíbe otras, permitiendo especificar el *protocolo* de uso de una API determinada.
En este trabajo se presenta un patrón de implementación de *typestates* en el lenguaje de programación `Rust` y se analiza su sistema de tipos y mecanismos que permiten la verificación del *typestate* en tiempo de compilación, mostrando que cumple con las propiedades requeridas por las propuestas de verificación modular descriptas en la bibliografía especializada.

1. Introducción

Cuando se desarrolla un componente de software, principalmente en el paradigma imperativo u orientado a objetos, se debe definir su interface (API) pero también es importante tener en cuenta el estado del objeto que representa al componente.

A modo de ejemplo, a una variable de tipo `File` se pueden aplicar las operaciones `read`, `write` y `close` luego de un `open` exitoso, o en un iterador (como en Java), la operación `next()` debería ser precedida por `hasNext()`.

En ingeniería de software, generalmente se utilizan máquinas de estados para describir estas clases de objetos y a nivel de diseño se utilizan varios formalismos y notaciones para su descripción y análisis.

Sin embargo, casi la totalidad de los lenguajes de programación carecen de mecanismos directos para describir el estado y sus transiciones. La programación basada en *typestates* [4] permite definir objetos no sólo en términos de clases (o tipos) sino también en los cambios de estado de manera explícita, describiendo los cambios de estado en base a la aplicación de las operaciones o métodos. Esto permite definir el *protocolo* que se debe cumplir en cuanto al uso de la componente, imponiendo la aplicación de las operaciones en los estados correspondientes. Sería deseable que esto pudiese ser verificado estáticamente.

Si bien muchos componentes de software se diseñan de esa manera, en la implementación se deben incluir verificaciones en tiempo de ejecución en las

operaciones para garantizar el uso correcto y en el código el *protocolo* queda implícito.

La idea de *typestates* [1] extiende la idea de *tipo* con el de *estado interno* y su incorporación en un lenguaje de programación permitiría la verificación en tiempo de compilación, evitando tener que considerar los estados en ejecución y el uso de mecanismos como devolución de códigos de error o disparo de excepciones.

Si bien la gran mayoría de los lenguajes de programación de amplia adopción no incluyen soporte para *typestates*, algunos lenguajes más recientes, en particular `Rust` [2], incluyen mecanismos en su sistema de tipos que permiten implementarlo adecuadamente y permiten su verificación durante la compilación.

La figura 1 muestra un simple ejemplo de una API sobre archivos.

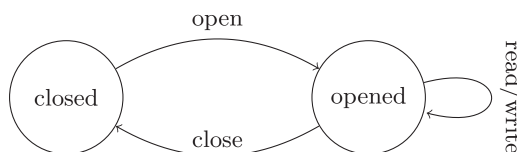

Figura 1. Sistema de transición de estados de una api `File`

En la próxima sección se describen las principales características de `Rust`, principalmente su sistema de tipos. Luego se presenta una propuesta de un patrón de implementación de *typestates*. En la sección 4 se analiza el soporte del sistema de tipos y mecanismos de `Rust` para implementar las ideas propuestas en [4] y [5]. También se consideran herramientas de verificación estática en otros lenguajes como Java y C++. En esta sección se encuentra la principal contribución de este trabajo. Finalmente se describen las conclusiones y trabajo futuro.

2. Características de Rust

`Rust` [2] es un lenguaje de programación reciente (Julio 2010), originalmente desarrollado por Graydon Hoare en *Mozilla Research*[1].

Entre sus principales características se pueden destacar:

- Sintácticamente similar a C++, aunque con construcciones más concisas.

[1] El navegador Mozilla Firefox, inicialmente escrito en C++, ya contiene varios subsistemas reescritos en `Rust`.

- Multiparadigma: Si bien se puede considerar un lenguaje imperativo, tiene mecanismos de soporte para programación orientada a objetos, genericidad y concurrencia.
- Adecuado para la *programación de sistemas* (software de base, embebido, etc) y aplicaciones que requieran alto rendimiento, como por ejemplo un navegador web moderno.
- Comparte la filosofía de C++ de brindar *abstracciones de costo cero*.
- Gestión manual de la memoria pero garantizando el desarrollo de programas *seguros* en cuanto a que previene tener *memory leaks* y *dangling references*.
- Es un lenguaje fuertemente tipado, con verificación estática de control de aliasing y tiempo de vida de variables.

2.1. Tipos de datos

Los tipos primitivos de `Rust` incluyen

- *Básicos*: lógicos (`bool`), caracteres Unicode (`char`), numéricos (enteros `i8, i16, i32, i64` y sus versiones sin signo `u8,...` y reales en punto flotante `f32, f64`).
- *Estructurados*: registros (`structs`), tuplas, arreglos de tamaño fijo (de tipo `[T; N]`, con `T` como tipo base y `N` como tamaño).
- *Uniones disjuntas (enums)*: Las enumeraciones pueden usarse como patrones en sentencias `match` para realizar *pattern-matching*.

```
1 struct Point { x: i32, y: i32} // struct
2 struct Color {i32, i32}        // struct tuple
3 enum RGB {Red, Green, Blue}    // disjoint union
4 let mut a = [1,2,3];           // mutable array
5 let first_two = &a[0..2];      // inmutable slice
6                                //(first 2 elements)
7 let pair = (i32, i32);         // tuple literal
8 let my_ref = &pair;            // inmutable reference
```

Las variables y referencias son inmutables por omisión, excepto que se use el modificador `mut`.

La biblioteca estándar define muchos tipos útiles como strings (`str`), vectores (arreglos de longitud variable), y otros.

2.2. Polimorfismo

`Rust` provee *polimorfismo paramétrico acotado*, similar a la generecidad de Java y polimorfismo *adhoc* (*type classes* o *traits*).

Es posible definir tipos y funciones con parámetros de tipos, como `vec<T>` y `fn max<T>(v: vec[T]) -> T { ...}`.

La manera de emular la herencia se logra mediante el uso de *traits*, las cuales podríamos decir que son similares a las *interfaces* al estilo de Java, aunque técnicamente son más del estilo de los *conceptos* de C++20 [10] o las *type classes* de Haskell. En las comunidades de usuarios de C++, Rust, Scala y otros, los traits representan *requisitos* (u operaciones) que determinados tipos deberán implementar.

Los *traits* son extensibles, permitiendo definir jerarquías y cualquier tipo puede implementar múltiples *traits*. Este mecanismo es menos restricto que la *herencia* y facilita la *extensibilidad y adaptabilidad de componentes*.

Es posible implementar un *trait* para cualquier tipo de dato. La biblioteca estándar define algunos *traits* de utilidad y algunos de ellos está integrados en el compilador, como por ejemplo, los traits `Copy`, el cual provee la operación `copy()` para implementar *copy semantics* y *Drop*, que provee la operación `drop()` que indica la liberación de recursos adueñados y la destrucción del objeto invocante. Estos *traits* son conocidos por el compilador y tenidos en cuenta por el *borrow chequer*.

Esta forma de estructurar tipos de datos y operaciones, permite extensibilidad multidimensional sin tener que definir una jerarquía de clases con las restricciones que ello implica. Estos mecanismos están presentes en otros lenguajes de programación modernos como Scala.

Para más detalles sobre `Rust`, se recomienda el libro oficial disponible en la página web: *https://doc.rust-lang.org/book/*.

2.3. Gestión de memoria

`Rust` al contrario de otros lenguajes como Go y Java, entre otros, no utiliza *recolección automática de basura*[2]. En su lugar se basa en el concepto (ampliamente usado en C++) de *RAII (Resource Adquisition is Initialization)*, es decir que los recursos de un objeto son adquiridos en su construcción y liberados en su destrucción siguiendo la idea de *ownership* en la cual el objeto es responsable de la liberación de los recursos que él creó.

`Rust` ofrece referencias mutables (`mut&`) e inmutables (`&`) con verificación estática (en tiempo de compilación) de su uso seguro:

> *Una referencia no puede tener un tiempo de vida mayor que el valor referenciado.*

`Rust` almacena los valores típicamente en el *stack*, aunque permite almacenar valores en el *heap* mediante el uso de constructores como `Box<T>` (el equivalente a `new` en otros lenguajes de programación, como Java o C++).

Además ofrece opcionalmente en su biblioteca estándar *smart pointers*, como `Rc<T>` (contador de referencias, similar a `std::shared_ptr<T>` de C++0x).

[2] Esto va en consonancia con el objetivo de ofrecer el máximo rendimiento posible, al estilo de C.

A pesar de que no dispone de un recolector de basura, el compilador fuerza a que la utilización de referencias sea de manera segura, es decir que si un programa compila, éste no contiene errores como *lagunas de memoria*, acceso a *referencias nulas*, o *condiciones de carrera (race conditions)* en presencia de concurrencia usando memoria compartida (*threads*.

Esta seguridad en el manejo de la memoria se basa en las siguientes restricciones o invariantes que un programa debe cumplir:

1. *Ownership*:
   - Cada valor tiene una única variable como *dueño*.
   - Cuando la variable se destruye (finaliza su alcance), el valor asociado se destruye.
   - Es posible tener múltiples referencias inmutables.

   La asignación (y el pasaje de parámetros) sigue *move semantics*, es decir que la variable original cede su *ownership* al destino, lo que invalida el uso futuro de la primera. Este sistema es una implementación de *affine types*.
2. *Borrowing*: Mecanismo estático que lleva la pista de *ownership* y tiempo de vida de valores y referencias.

   Además de verificar las condiciones de *ownership* debe contemplar la transferencia temporal de variables (*owners*) a otras variables o referencias, como es el caso de pasaje de parámetros por referencia y en alcances (scopes) interiores.

El siguiente ejemplo muestra un uso indebido de variables:

```
1 let mut x = 5;
2 let y = &mut x;
3 *y += 1;
4 println!("{}", x);
```

El compilador detecta un uso de `x` en la línea 4 (ha cedido su valor a `y`).

En cambio el siguiente código es legal.

```
1 let mut x = 5;
2 {
3         let y = &mut x;
4         *y += 1;
5 }
6 println!("{}", x);
```

ya que dentro del bloque (líneas 2-5) se hace referencia al valor adueñado por `y` (únicamente) y a la salida del bloque, `x` recupera su condición de (único) *dueño*.

Notar que en este ejemplo el compilador puede determinar que el tiempo de vida de y es menor al de x.

Este mismo escenario es válido para el pasaje de referencias como parámetros a funciones.

3. Implementación de typestates

El objetivo de *typestates* es garantizar, en tiempo de compilación, el cumplimiento del *protocolo* de uso asociado a una API. La idea más simple de implementación es asociar a cada estado un tipo que provea las operaciones aplicables en ese estado, un enfoque comúnmente encontrado en implementaciones de lenguajes de dominio específico *DSL*. Cada operación retorna un objeto del tipo correspondiente a las operaciones disponibles, de acuerdo al protocolo de uso de la API al que pertenece el objeto[3]. En la Figura 3 se muestra un ejemplo de la API `StFile` (typestate File).

Es importante notar que para las operaciones que cambian el estado, como `open` o `close`, el objeto (`self`) se pasa por valor, lo que tiene como consecuencia que el objeto se *consuma (borrow)*, es decir que no pueda usarse en operaciones subsiguientes. Recordar que `Rust` usa *move semantics* por omisión. Por eso al compilar el código del siguiente ejemplo:

```
1 fn main() {
2     let f = file();
3     let mut f = f.open(); // we define a "new" f here
4     let mut g = f;
5     f.read();
6     let f = f.close();
7 }
```

Obtenemos el siguiente error de compilación:

```
error[E0382]: borrow of moved value: ‘f‘
   --> typestate.rs:51:2
    |
3   |    let mut f = f.open();
    |        ----- move occurs because ‘f‘ has type ‘StFile<FileOpened>‘,
    |              which does not implement the ‘Copy‘ trait
4   |    let mut g = f;
    |                - value moved here
5   |    f.read();
    |    ^ value borrowed here after move
```

Este ejemplo muestra que el *borrow checker* lleva el control de *ownership*.
Si eliminamos del ejemplo de uso anterior las líneas 3 y 4,

```
1 fn main() {
2     let f = file();
3     f.read();
4     f.close();
5 }
```

obtenemos el siguiente error:

[3] El uso de tipos parametrizados permite simplificar la implementación.

```
1  // States (phantom types)
2  struct FileClosed;
3  struct FileOpened;
4
5  struct File; // actual file i/o impl.
6
7  struct StFile<State> {
8    // real implementation of File i/o
9    _impl: File,
10   // zero size data, not represented in runtime
11   _state: State
12 }
13
14 // Implementation for closed state.
15 //User only can call open()
16 impl StFile<FileClosed> {
17   // open() consumes self and return a opened file
18   fn open(self) -> StFile<FileOpened> {
19     File {
20       _impl: self._impl,
21       _state: FileOpened
22     }
23   }
24 }
25
26 // Implementation for open state.
27 // User can use read(), write() and close()
28 impl StFile<FileOpened> {
29   // This operations lead object in the same state
30   fn read(&mut self)  {}
31   fn write(&mut self) {}
32
33   // consume self and return a new closed state object
34   fn close(self) -> StFile<FileClosed> {
35     File {
36       _impl: self._impl,
37       _state: FileClosed
38     }
39   }
40 }
41
42 // simple factory for file in initial state (closed)
43 fn file() -> StFile<FileClosed> {
44   File {
45     _impl: File,
46     _state: FileClosed
47   }
48 }
```

Figura 2. `File` con typestates.

```
 error[E0599]: no method named ‘read‘ found for type ‘StFile<FileClosed>‘
               in the current scope
   --> typestate.rs:3:4
3  |      f.read();
   |        ^^^^
```

porque hemos violado el protocolo al no haber invocado previamente a `open` y se aplica la operación `read()` sobre un objeto en estado `Closed`.

El siguiente listado muestra un uso legítimo del protocolo y la compilación es exitosa.

```
fn main() {
     let f = file();
     let mut f = f.open();
     f.read();
     f.close();
}
```

Como se puede apreciar, el único inconveniente que esto genera es que ante cada operación que produce un cambio de estado se deberá usar una nueva variable, ya que esta operación retornará un valor de un nuevo tipo. Esto no le quita simplicidad al uso de estos tipos ya que `Rust` permite el rehúso de identificadores y además omitir el tipo (inferencia automática).

4. Verificación estática

En esta sección se analizan y justifican porqué el sistema de tipos y el *borrow checker* de `Rust` permiten realizar la verificación de *typestates* cuando se implementan en un estilo como el que se muestra en la sección anterior.

En [5] y [4] se describe una propuesta de análisis estático modular de *typestates*.

Cualquier algoritmo de verificación de *typestates* deberá llevar la pista de cada estado de un valor en un punto de programa determinado, teniendo en cuenta las transiciones de estados.

Con lo anterior no alcanza si el lenguaje permite *aliasing*, por ejemplo, mediante el uso de *referencias*. En cuyo caso, un verificador estático deberá verificar que un valor puede ser accedido desde varias referencias lo que hace la verificación mucho más compleja.

En los artículos referenciados arriba, Bierhoff, Aldrich y sus colegas han propuesto lenguajes con construcciones o anotaciones que permiten al sistema de tipos tener control y así poder *razonar* en presencia de aliasing.

Particularmente, en [5], se propone un algoritmo de verificación de typestates que requiere que las referencias tengan las siguientes características:

1. *Lógica lineal*: Cada variable puede ser asignada sólo una vez.

2. *Permisos de acceso*
   - *full*: Permite acceso de lectura/escritura al valor referenciado. En este caso, sólo deberá haber una única referencia al valor (en cada scope).
   - *pure*: Acceso de sólo lectura. En este caso, puede haber múltiples referencias *puras*.

En `Rust`, usando variables inmutables y gracias a *move semantics* se logra implementar lógica lineal, mientras que el *borrow checker* garantiza que en presencia de aliasing, existe una única referencia mutable en uso en un bloque (*scope*) dado, o bien pueden existir múltiples refererencias inmutables.

Mediante estos mecanismos, junto con el patrón de implementación propuesto, se logra que la verificación estática realizada por el compilador prácticamente imite el algoritmo propuesto en [5].

Es importante notar que en otros lenguajes de programación, como por ejemplo C++, aunque soporta *move semantics*, la verificación de *lógica lineal* sólo puede ocurrir en tiempo de ejecución (cuando se intente acceder a un valor movido anteriormente se producirá una excepción), ya que a diferencia de `Rust`, no se realiza ningún chequeo en tiempo de compilación.

En otros lenguajes la verificación de *typestate* podría hacerse mediante el uso de herramientas externas, como por ejemplo con el *clang static analyzer* [7,9,3], que contiene soporte para el análisis de *dataflow*, como se propone en [6].

5. Conclusiones y trabajo futuro

El uso de *typestates* es cada vez más importante para el desarrollo de software confiable. Existen pocos lenguajes de programación con soporte para su verificación estática. Algunos de los lenguajes que lo soportan son experimentales y no han sido adoptados ampliamente en producción.

Si bien en las versiones iniciales de `Rust` se incluía soporte para especificar *typestates*, esta característica fue eliminada en su versión 1.0. Sin embargo, en este trabajo se muestra una posible estrategia de implementación sin costo extra en tiempo de ejecución, es decir que no se requiere más memoria ni chequeos dinámicos, aprovechando el sistema de tipos y el verificador (*borrow checker*) para conseguir análisis estático de *typestates* en `Rust`.

Muchos componentes de la biblioteca estándar de `Rust` se están desarrollando siguiendo el concepto de *typestates*.

Como trabajo futuro se está desarrollando un *checker* para el *Clang static analyzer* [3] ya que provee un framework de ejecución simbólica, el cual permitirá al desarrollador especificar el sistema de transición de estados representando el *typestate* y analizar su uso. El checker en desarrollo usa otro checker existente (el *move checker*). El principal desafío es la *emulación* del *borrow checker* de `Rust`.

Referencias

1. Robert E. Strom and Shaula Yemini. *Typestates: A Programming Language Concept for Enhancing Software Releability*. IEEE Transactions on Software Engineering, vol SE-12, nº 1, 1986.
2. Steve Klabnik and Carol Nichols. *The Rust Programming Language (Ed. 2018)*. https://doc.rust-lang.org/book/
3. Clang static analyzer. http://clang-analyzer.llvm.org/
4. Jonathan Aldrich, Joshua Sunshine, Darpan Saini, Zachary Sparks. *Typestate-Oriented Programming*. Onward 09. ACM 78-1-60558-768-4/09/10. 2009/2010.
5. Kevin Bierhoff, Jonathan Aldrich. *Modular Typestate Checking of Aliased Objects*. OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada. ACM 978-1-59593-786-5/07/0010. 2007.
6. Thomas Reps, Susan Harwitz, Mooly Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. POPL '95 Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Pages 49-61. ISBN:0-89791-692-1.
7. Zhongxing Xu, Ted Kremenek, Jian Zhang. *A Memory Model for Static Analysis of C Programs*. International Symposium on Leveraging Applications of Formal Methods, 2010. Lecture Notes In Computer Science. Vol. 6415.
8. J. Aldrich, et al. *Permission Based Programming Languages*. ICSE'11, May 21–28, 2011. ACM 978-1-4503-0445-0/11/05. 2011.
9. Hari Hampapuram, Yue Yang, and Manuvir Das. *Symbolic Path Simulation in Path-Sensitive Dataflow Analysis*. PASTE '05 Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. Pages 52-58. ACM SIGSOFT Software Engineering Notes. Volume 31 Issue 1, January 2006. Pages 52-58.
10. Bjarne Stroustrup. *Concepts: The Future of Generic Programming*. Technical Report P0557r1. Morgan Stanley and Columbia University. 2017.