Rust — A Systems Programming Language

Blazingly Fast, Prevents Segfaults And Guarantees Thread Safety

Technische Universität Dresden
Fakultät Informatik
Lehrstuhl für Betriebssysteme

Sebastian Humenda, July 30, 2016

Contents

0. License

1. Introduction

Rust is a systems programming language developed at Mozilla Research and aims at being reliable, memory safe, fast and data race free. It provides high-level abstractions, while allowing for tight control over resources as e.g. memory management for systems programming (Matsakis and Klock 2014; The Rust Developers 2016b). The following section will give a rough overview over the most important features of Rust, which are necessary to understand the subsequent sections.

Rust is a statically typed language offering type inference. This means that the compiler can infer the type in most of the situations by the surrounding expressions and statements, therefore taking some of the typing burden introduced by static typing from the programmer.[1]

A novel ownership and borrowing model is introduced as well (see 2.1 on the next page), allowing static safety guarantees and compile-time memory management, eleminating the need for garbage collection (Reed 2015; The Rust Developers 2016b). The system was inspired by cyclones region-based memory management (The Rust Developers 2016b). Rust calls regions *life times* and they are used by the compiler to track an object and issue its deallocation statically (Reed 2015).

Some of the features in Rust were inspired by research done on Sing Sharp, a programming language developed and used in a research project with the name Singularity. In this project, an operating system was built, which isolated software processes solely by the type system, so in the end by the compiler. Some of the ideas can be found in Rusts communication channels and its strict type system (Hunt and Larus 2007). Singularity OS relies for resource management and other security-related tasks on the isolation provided by the compiler. But while Sing Sharp enforced some of its safety guarantees using a runtime and a garbage collector,[2] Rust relies soley on static code analysis within the compiler.

Rust offers zero-cost abstractions using traits. Traits are roughly similar to an (Java) interface, offering a type-generic implementation which gets transformed into a type-concrete implementation during compilation (The Rust Developers 2016b, chap. 4.22). This way, abstract specifications can still be inlined and hence obtain good performance. Furthermore, Rust offers dynamic dispatch to allow for polymorphism, giving the programmer the choice when to trade off performance against flexibility (The Rust Developers 2016b, chap. 4.19, 4.22).

Macros in Rust are substantially different from macros in C. They work on the syntax tree and are hence not prone to substitution issues. They can be used for static code generation, since they integrate nicely with the strict type system. In addition to the built-in macros, the programmer can easily define his/her own macros. Examples for

[1] Some more explanation and examples are given on page 14.

[2] Sing Sharp does not allow for fine-grained control of the hardware and it was not designed primarily for memory safety. All memory safety provided are state-of-the-art runtime features.

built-in macros are `panic` to stop the execution of a thread or `println` to print to standard output[3] (Anderson et al. 2015, p. 4; The Rust Developers 2016b).

2. Memory Safety

Rust enforces an ownership and borrowing model for memory management, which is designed to prevent typical memory safety issues like memory leaks, use after free, dangling pointers and use of uninitialized pointers/references. These properties are all enforced solely through the type system. Since this is enforced statically, there is no execution overhead, because of these safety properties. (The Rust Developers 2016b, chap. 4.8).[4]

To enable the compiler to check statically for memory safety, two invariants have to hold (Reed 2015):

1. There is exactly one unique owner of an object, responsible for deallocation.
2. Memory is never aliased and mutable at the same time.

The implications of these two invariants are explained in the subsequent sections.

2.1. Ownership

Every variable binding[5] has exactly one owner. Therefore, values cannot be aliased. If a binding (or its content) is assigned to a second binding, the value is moved.

```
1 fn main() {
2   let v = vec![0; 42]; // init. vector[6]
3   let w = v; // valid
4   println!("{}", v[0]); // invalid, use of moved value `v`
5 }
```

Figure 1: The contents of the vector (R-Value) can only be owned by one binding in one scope

Initializing an object defines an owner, as can be seen in figure 1, where a vector is initialized. Line 2 however does not reassign the value of v to w, but **moves** the value to

[3] Println is a macro which transform the varargs-alike call to a call to an internal print function with a fixed number of arguments.

[4] An exception are out-of-bounds checks, which are inserted before vector accesses. These are however optimized away if possible and can be even bypassed with unsafe functions (The Rust Developers 2016a).

[5] Variable bindings bind a name to a value. There are subtle differences to "Variables", not discussed here (The Rust Developers 2016b, chap. 4.1).

[6] As a shorthand, the standard library provides a macro to initialize vectors more easily. The shown syntax initializes the vector with 42 0's.

```
1 fn length(w: Vec<i32>) -> i32 {
2   vec.len()
3 }
4 fn main {
5   let v = vec![6, 5, 4, 3, 2, 1]; // v ownes values
6   let len = length(v) // w in length is owner
7 }  ↑ value of v lost
```

Figure 2: example of moved ownership and freed object

w, transfering the ownership. That leaves v as "empty". Consequently, the compilation failes on line 3, because of the invariant *only one owner*, exactly one binding can own and access a value.

Figure 2 sketches the transferal of ownership to another scope (function). The vector v is initialized with a few integers and then passed to the `length()` function. `length()` function has a parameter w which now owns the value of v. The function signature shows that a vector of i32[7] is taken as input argument. The argument declaration is followed by a type and the return value.
When `length()` returns, the binding w falls out of scope. Since w was owning the values and `length()` was owning w, this will trigger a free operation on the value of w.

All free operations in Rust are automatically inserted by the compiler and applied according to the ownership/borrowing model. The user can implement custom deallocation code to e. g. free allocated resources[8] (Matsakis and Klock 2014; The Rust Developers 2016b, chap. 4.1).

2.2. Borrowing

Even though the ownership model is powerful and allows for tracking of each and every binding and hence allows deterministic and guaranteed frees of resources, it is not convenient when passing ownership into and out of a scope again. To ease this, a value of a binding can be borrowed, so that only a a reference to the object is passed. While a borrow takes place, the owner of the value is not able to access the value, but control to it will be handed back when the borrow goes out of the foreign scope. The borrow checker enables safe aliasing (Reed 2015) and can be imagined as a static reference count within the compiler. Borrowed references have to be temporary, otherwise memory safety could not be guaranteed (Reed 2015, p. 3).

[7] All primitive types (float, int, unsized, int, …) have an explicit size in Rust which does not vary across platforms (The Rust Developers 2016b, chap. 4.3).
[8] To implement custom deallocation / free operations for types, the programmer needs to implement the trait drop (The Rust Developers 2016b, chap. , 4.19). Traits are not discussed within this work.

A borrow can be either mutable or immutable. To guarantee that no dangling pointers can occur, there may be either only one *mutable reference* or *one or more immutable references* (The Rust Developers 2016b, chap. 4.9; Matsakis and Klock 2014; Reed 2015).[9]

Figure 3 shows a better implementation of `length()`, which takes the vector `v` as a reference and therefore allows the owner to use the object after the function has returned.

```
1 fn length(w: &Vec<i32>) -> i32 {
2   vec.len()
3 }
4 fn main {
5   let v = vec![6, 5, 4, 3, 2, 1]; // v ownes values
6   let len = length(v) // v is borrowed
7   println!("length: {}, first element: {}", len, v[0]);
8 }
```

Figure 3: example for an immutable reference

2.3. Further Measures

The ownership and borrowing model of Rust is an extended version of RAII as known from C++. Objects with just an owner are comparable to an `unique_ptr` in C++ and immutable references to a `shared_ptr`. While it is possible to achieve memory safety within C++, Rust enforces these properties with its type system (Lippman, Lajoie, and Moo 2013, chap. 12.1; The Rust Developers 2016b, chap. 4.8–4.10; Reed 2015).

With the introduced measures, bugs as use after free, dangling pointers, null pointers and memory leaks cannot occur. But there is still the possibility of buffer overflows. To prevent these, Rust inserts bound checks. Whenever the compiler can detect that the bounds are not going to be violated, the bounds checks are optimized away. If performance is critical and random access is required, the bound checks can also be circumvented in an unsafe way (see 2.4 on the next page). Last but not least, Rust has extensive and optimized support for iterators, which can also help to get around bound checks (The Rust Developers 2016b, chap. 4.5, 4.7).

To avoid dereferencing of null pointers or access to invalid data, Rust offers a type called `Option<T>`, which contains either a value or `None`. This way, the programmer is forced to explicitly check each value and avoids null values being propagated. Using the value later on is called ”unwrapping”. Through this explicit and enforced check for null, no uninitialized data can be read or used. There is also a `Result<T>` which is similar to an option, but wraps either a value or an error.

[9] In the referenced paper, the authors give an explanation while aliasing and mutability doesn't work at the same time.

```
1 let x = Some(20); // init Option<i32>
2 let temperature = x.expect("temperature is mandatory");
3
4 let pointer = Some(Box::new(9000));
5 let money = match pointer {
6   Some(ref a) => *a // derefence a
7   None => 0 // default
8 }
9
```

Figure 4: usage of option to handle errors and prevent access of null values

In figure 4 an `Option` is unwrapped using both pattern matching and functions provided by the type `Option<T>` (The Rust Developers 2016b, chap. 4.18). In line 1, an Option is initialized[10] and its value is assigned. The type `Option` provides several functions to use the contained value, one being `expect`, which panics with the given message if no value is contained in the option. There are other functions to use default values or execute a closure instead (The Rust Developers 2016a).

The second example shows a pointer represented by the `Box` type. Pointers are implemented as an optional (`Option<T>`) type in Rust. Therefore null pointer do not exist, since `Box<T>` has to be checked for a value and cannot be simply dereferenced.
In line 5, a match is used to determine whether the pointer is valid or not. Match is an expression similar to a switch statement, but more powerful, since each case is evaluated against an expression. It is also checked for exhaustiveness.[11] Each case can have a nested structure, it just needs to return a value to be used in the right-hand side of the case block. The first case therefore is executed, if the `Option` contains something and that something is a reference, which we call 'a'. 'a' is dereferenced on the right-hand side, which is safe. The second case is a simplistic error-handling case (The Rust Developers 2016b, chap. 4.14).

2.4. Unsafe Regions

Some data types and algorithms cannot be expressed by the constraints enforced by the type system. With `unsafe`, the programmer can violate some of the rules temporarily. The programmer can hide these few unsafe regions behind a safe facade.

In unsafe code, the following things are possible:

1. Access or update a static mutable variable.[12]

[10] Options are enum typed and consist of the inner types Some and None.
[11] The compiler will try to figure out whether all possible cases have been covered, i.e. None and Some, all numbers from a range, etc.
[12] `static` variables are global variables.

2. Dereference a raw pointer.
3. Call unsafe functions.

The 3rd item is the most powerful ability, because it allows to call arbitrary code, including foreign code from other programming languages, which cannot be checked for the strong safety guarantees (The Rust Developers 2016b, chap. 4.36). It also allows the implementation of higher-level, generic and safe data types out of partly unsafe components (The Rust Developers 2016b, chap. 4.36).

To make a certain region of code unsafe, the `unsafe` keyword can be used. If a function or trait is prefixed with unsafe, this function/trait can only be used within an unsafe context.
If a block, i.e. `unsafe { ... }` is annotated with `unsafe`, the programmer asserts that after leaving the block, the compiler will find everything in a consistent state and that all the strict properties of Rust hold.

Examples:

```
unsafe function foo() { ... }
unsafe { // init "raw" pointer
  let x = 5;
  let raw = &x as *const i32;
  *raw; // not guaranteed to be valid, although it is here
}
```

Rust also provides a Foreign Function Interface, see A.2 on page 15 which makes it possible to call code from other languages. This is considered unsafe from the compiler's perspective, because the compiler cannot guarantee its safety properties for these calls.

3. Thread Safety

Concurrency has gained a lot of attention in hard- and software design and it remains a challenging task to develop scalable, maintainable and (memory) safe concurrent software. Rust brings the same memory safety guarantees into concurrent programs, while preserving the freedom of the programmer to write highly scalable applications. Rusts type system is able to reason about concurrent code at compile time. It is able to prevent data raises, a common and difficult source of errors. All concurrency features are implemented in the standard library, so that they can be swapped by the programmer, if need be.

Certain algorithms cannot be expressed with Rusts safety guarantees and are yet safe. These can be wrapped with virtually no overhead in a safe layer, only requiring a few lines of unsafe code (Anderson et al. 2015).

3.1. Channel-based Communication

Using channels between processes and threads is the easiest and most scalable way of writing concurrent programs. The key benefit is that no data races can occur, because either one thread has a copy of a particular datum or it does not. This also can be a clear advantage for scalability, because it prevents dependencies.

Rusts channels are inspired by Singularity (Hunt and Larus 2007). In Singularity, processes are isolated by software mechanisms, not by hardware. Singularity comes with a programming language Sing Sharp, which is strongly typed and enforces security with its type system. Channels are contract based and are also strongly typed (Matsakis and Klock 2014; The Rust Developers 2016b).

Channels in Rust transfer ownership. Consequently, only objects can be send which can transfer ownership, which is indicated through implementing a special trait called `Send`. It is appropriate to implement this trait for a custom data type with self-contained state. It is however not appropriate to call it for an object holding a resource, where the ownership cannot be transfered. Another case is an object exposed through the FFI. The compiler will enforce this policy.

Another special trait is the `Sync` trait. It tells the compiler that this type can be safely used by multiple threads (read-only) without introducing memory unsafety. Simple data types are automatically `Sync` and every data type **only** made up of primitive types has this property as well.

3.2. Shared Sections

It can be necessary to share a larger data structure or to have concurrent access, in which case a shared section is more appropriate than channel-based communication.

The standard library offers wrapper types implementing `Send` and `Sync`, which require the same for their wrapped values. For read-only data sharing, the type `Arc<T>`[13] allows sharing across threads. It provides a `clone()` function to get a copy of another reference, so that each thread holds its own borrowed copy to the value owned by the instance of `Arc<T>`.

If data has to be shared in a writable manner, it has to be wrapped in a `Mutex` type. This type will enforce a lock when the data is going to be accessed and will unlock the data again when the critical section is left. This way, no data races can occur, because data can only be shared with a mutex in a writable manner and **all** other access is prevented, because the Mutex **owns** the wrapped value (it is moved into it when the mutex is initialized).

[13] *Arc* stands for atomically reference counted.

```
1  use std::sync::{Arc, Mutex};
2  use std::thread;
3
4  fn main() {
5    let numbers: Vec<_> = (0..100).collect();
6    let shared = Arc::new(Mutex::new(numbers));
7    let mut children = Vec::new();
8
9    for i in 0.100 {
10     let child_numbers = shared.clone();
11     children.push(thread::spawn(move || {
12       let local = &mut child_numbers.lock().unwrap();
13       local[i] = i * i;
14     }));  ↑ unlocked when scope left
15   }
16 }
```

Figure 5: parallel writes on a variable synchronized by a mutex

In figure 5, a vector is initialized and wrapped in a `Arc<Mutex<T>>`. The vector on line 5 is initialized using a range[14] and converted into a vector using `collect`. Afterwards, the numbers are wrapped in a shareable mutex type. Please note that the binding defined on line 5 is not useable afterwards anymore, since the values have been moved along with their ownership.
The mutable vector initialized on line 7 is used for the thread handles and its type doesn't need to be specified, since the compiler can infer it statically from the call to `push` further down.

On line 11, a new thread is started with a closure as argument. This closure contains the code to be executed in the thread.[15] The copy of the reference made on line 10 is used on line 12 to acquire the lock. The result is then unwrapped and assigned to a local binding.[16] The `&mut` in front of the mutex.unlock call tells that we want a mutable reference, it would be immutable by default.
When the closure is left, the mutex is automatically unlocked. Since another scope can be introduced solely using `{...}`, it is possible to have automatically-managed, short and precise critical sections.

[14] Rusts ranges are iterators under the hood and lazily evaluated. They are not a container data structure.
[15] Closures have access to the surrounding state, but the compiler will watch which variables are actually used.
[16] Lock returns an option. Doing an unwrap on it will discard all possibility of error handling and directly returns the value. If the mutex was poisoned, the thread would panic and hence stop execution.

4. Performance

Rust is designed to deliver performance comparable to C++ (Anderson et al. 2015). There are no official benchmarks yet for Rust, but several non-scientific measurements have been taken. Anderson et al. compared their very slim browser rendering engine against classical engines against others. This rendering engine is not feature complete yet, but it is able to outperform Gecko by far which gives a rough idea of Rusts optimizability.

The Benchmark Game

The Benchmarkgame is a project to provide a set of implementations for a list of algorithms. Each algorithm is implemented in all languages and execution times, memory usage and a few other parameters are logged. Execution times are averaged and the benchmarks are re-run repeatedly with the latest compiler updates. This gives a rough idea about Rusts speed, but the quality of the implementations is uncertain.[17] Below is an excerpt comparing the performance of Rust against C, taken from the benchmark game. It was taken on a quad-core 2.4 GHz Intel Q6600 with 4 GB of RAM, running Ubuntu GNU/Linux x64.

	Rust		**C (GCC)**	
Algorithm	time s	mem KiB	time s	mem KiB
pidigits	1.74	8,104	1.73	1,992
binary-trees	3.78	128,060	3.26	156,840
fannkuch-redux	16.65	20,284	8.97	1,588
spectral-norm	4.01	14,284	1.98	1,868

Project home: `http://benchmarksgame.alioth.debian.org`

5. Applications

5.1. GPU Programming

Holk et al. experimented with Rust for GPU computations. They claim that nowadays languages used on GPUs are to low-level to provide programmer-friendly abstractions offered by modern programming languages.[18] This has been adressed by using DSLs

[17] Although the majority of the benchmarks attest Rust very good performance, a few implementations are so slow, that even the Java implementation (which includes JVM start up and shut down) outperforms Rust.

[18] The whole section is taken from (Holk et al. 2013).

compiling to, e. g. Cuda, but these are not as expressive as Cuda and are another layer of indirection.

On the other hand, Rust offers zero-cost abstractions and through its usage of LLVM, it is in theory portable across architectures. LLVM has support for the PTX architecture[19] and Holk et al. try to use this backend, disabled by default, to compile code for the GPU. This way algebraic data types and other abstractions can be used on the GPU.

To execute the code on the GPU, a runtime is required. The OpenCl runtime uses the PTX format as well and therefore the authors used the OpenCl runtime to execute their code. Some parts of the compiler were modified, for instance a `#[kernel]` directive was introduced to mark kernels and functions and special variables, for instance to access the global thread id.

The authors conclude that the performance of their generated code is similar to those of hand-written OpenCl kernels. They give some insights into the performance of some higher-level features, for instance that most closures are simply inlined by the compiler and are hence no performance penalty.

The paper shows that Rusts memory safety model can be applied to the GPU and demands for more research into zero-cost abstractions for highly parallel computation.

5.2. Servo

Servo was started as a long-term replacement technology for the browser rendering engine Gecko. Gecko is implemented in C++ and hence offers tight control of resources and offers hence good sequential speed. On mobile devices, processors are less powerful and the trend is to do more in parallel. Because Rust has a focus on thread-safety and low-level control, the project was started to bring better performance, better power usage and more safety to the browser. An informal analysis done by Anderson et al. showed that roughly 50 % of the errors were use after free, out of range access and integer overflow.[20]

According to the authors, memory safety will however not be the key to its success, it must be at least as fast if not faster as existing rendering engines. The authors highlight several features of the language, enabling for more speed as static dispatch by polymorphism by default, pattern matching for static dispatch (no specialized optimization necessary), polymorphism gets applied to one specific case, hence monomorphised and code is generated for the concrete type (as also done in C++) (Anderson et al. 2015, p. 2 ff.).

[19] This is an intermediate ISA used by NVIDIA graphics cards.

[20] Integer overflow checks are only prevented through dynamic checks inserted by the compiler. Release builds turn off this feature to gain performance, developers however catch this type of error (The Rust Developers 2016b).

The implementation of Servo also faces some difficulties. For instance can scripts modify the DOM while it is build.[21]

Some data structures from the Rust standard library are not suitable for Servo's needs. That doesn't mean that they are poorly implemented, but rather that Servo's needs are very specific. For example, doubly-linked lists offer in certain situations best performance, but since multiple mutable pointers to one object are necessary, this is impossible to implement in Rust by default.[22] There is a work-stealing implementation written in partly unsafe Rust for Servo as well, an algorithm which cannot be implemented with the tight safety guarantees.

Language interoperability is an issue for two cases as well: Rust cannot call into C varargs (which can be mitigated using a C wrapper) and cannot call into C++ code natively. The latter is a performance issue, because all the C++ code needs to be accessed using the FFI as well. However it would be possible to bring Rust's and C++'s code together, because both can be compiled to LLVM's intermediate language (Anderson et al. 2015, p. 3ff.)).

Servo uses channels for nearly all of its communication. This way, each thread has its own copy of the data, no synchronization or locking is required. Only for the flow tree data structure,[23] multiple readers need to have access in parallel and for this scenario a shared region is faster.

5.3. Redox

Redox OS is a modern unix-alike operating system written entirely in Rust. It is based on a microkernel and is already able to boot a graphical window manager. There are no publications yet, but a lot of material and information, as well as an ISO image to try out. The project can be found at:

`http://www.redox-os.org`

A. Further Reading

A.1. Type Inference

Type inference is a modern capability of compilers and means that the compiler can infer the type of an expression out of the types used within. This does not mean that

[21] Modern browsers even do speculative token scanning in scripts to increase performance (Anderson et al. 2015, p. 2).

[22] For a doubly-linked list, all pointer construction and modification has to be wrapped inside an unsafe block. A thin unsafe layer can therefore make the operations on the list safe.

[23] The flow tree contains styling and display information from the page being displayed and is hence a frequently used, tab-globally accessed data structure.

the types are dynamic, they are known at compile time, but the compiler is able to reason about the type automatically. It will abort as soon as a type cannot be inferred, because of an ambiguous expression. Consider the following, where the left and right cell compile to the same code:

Explicite Type	**Inferred Type**
`let x: i32 = 9;`	`let x = 9;`
`let y: Bar = Bar::new();`	`let y = Bar::new();`

The compiler can even infer the type of i.e. a generic (here `Vec<T>`) by examining the later usage of the binding:

```
// type infered through the push statements:
let z = Vec::new();
z.push(9); // is Vec<i32>
z.push(10);
// invalid, does not compile:
// vec.push("z");
```

More information: The Rust Developers 2016b, chap. 4.1.

A.2. Foreign Function Interface

The Foreign Function Interface *FFI* allows Rust to call into binary code from other languages, specifically C. Foreign code is considered unsafe and hence are all calls to foreign code unsafe. When a library is going to be used within Rust, the library is normally wrapped within a thin layer of Rust code to abstract from the underlying unsafety.

More can be found in The Rust Developers 2016b, chapter 5.9. There is also a book dedicated to writing unsafe Rust code, located at:

`https://doc.rust-lang.org/nomicon/`

References

Anderson, Brian et al.: "Experience Report: Developing the Servo Web Browser Engine using Rust". In: *CoRR* abs/1505.07383 (2015).
Web: http://arxiv.org/abs/1505.07383.

Holk, Eric et al.: "GPU Programming in Rust. Implementing High-Level Abstractions in a Systems-Level Language". In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 315–324. DOI: 10.1109/IPDPSW.2013.173.
Web: http://dx.doi.org/10.1109/IPDPSW.2013.173.

Hunt, Galen C. and James R. Larus: "Singularity: Rethinking the Software Stack". In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424.
Web: http://doi.acm.org/10.1145/1243418.1243424.

Köster, Johannes: "Rust-Bio - a fast and safe bioinformatics library". In: *CoRR* abs/1509.02796 (2015).
Web: http://arxiv.org/abs/1509.02796.

Lippman, Stanley B., Josée Lajoie, and Barbara E. Moo: *C++ Primer (4th Edition)*. Addison-Wesley Professional, 2013.

Matsakis, Nicholas D. and Felix S. Klock II: "The Rust Language". In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641.
Web: http://doi.acm.org/10.1145/2692956.2663188.

Reed, Eric: "Patina. A Formalization of the Rust Programming Language". In: (Feb. 2015).

Rust by Example. Apr. 9, 2016.
Web: http://rustbyexample.com.

The Rust Developers: *Standard Library API Reference*. Aug. 2016.
Web: https://doc.rust-lang.org/1.10.0/std.

– *The Rust Language*. No Starch (unpublished), Aug. 2016.
Web: https://doc.rust-lang.org/1.10.0/book.