

# Bounded generics over constants in Rust

---

Author: Christian Poveda

Advisor: Nicolás Cardozo

2018-09-18

Systems and Computing Engineering Department  
Universidad de los Andes



## Context: What is a type?

On this snippet

```
String myString = "Hello, World!";
```

String is the type of myString and as such, it determines:

- The operations on which myString could be used.
- The values myString could take.
- The type of "Hello, World!" (it should be String).

To be more general:

*A type system is a syntatic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute*

## Context: The Rust programming language

Rust is a systems programming language focused on safety, speed and concurrency. It is sponsored by Mozilla.

However, Rust is not your typical language:

- It does not have garbage collection nor pointer arithmetic, but it is memory safe.
- It is blazingly fast (as fast as C++), but it has high-level features:
  - Pattern matching
  - Traits
  - Higher order functions
- It features concurrency without data races.

## Context: Rust's type system

Rust's type system is based on the ML type system. As such it has:

- Static type checking (i.e., programs are checked for safety during compilation).
- Type inference (i.e., type annotations are not always needed).
- Polymorphism via traits and generics.
- Enums and Structs.

It also takes care of memory safety

- Rust encodes the lifetime of each variable in its type.
- For each variable, Rust only allows one of the following:
  - One mutable reference.
  - Several immutable references.

## The problem: Functions over arrays

In Rust, Arrays (being stack allocated) must be statically sized.

As a consequence, this code compiles

```
fn add_arr(a: &[f64; 3], b: &[f64; 3]) -> [f64; 3] {  
    let mut result = [0.0; 3];  
    for i in 0..3 {  
        result[i] = a[i] + b[i];  
    }  
    result  
}
```

## The problem: Functions over arrays

In Rust, Arrays (being stack allocated) must be statically sized.

But this code does not

```
fn add_arr(a: &[f64; N], b: &[f64; N]) -> [f64; N] {  
    let mut result = [0.0; N];  
    for i in 0..N {  
        result[i] = a[i] + b[i];  
    }  
    result  
}
```

## The solution: Generics over constants

Generics are a way of archieving polymorphism based on parametrizing types. For example

- `Vec<T>` is the type of vectors of elements of type `T`
- `Vec<bool>` is the type of vectors of booleans
- `Vec<i32>` is the type of vectors of integers

We could allow the same pattern letting constant values be parameters

- `[bool; N]` is the type of arrays of booleans and size `N`
- `[bool; 0]` is the type of arrays of booleans and size `0`
- `[bool; 1]` is the type of arrays of booleans and size `1`

We only use constants because Rust's type checking is static.

## The solution: Generics over constants

In general, a type `Foo` parametrized by a constant `C` of type `T` would be written as

```
Foo<const C: T>
```

In a similar fashion, a function `bar` parametrized by a constant `C` of type `T` would be written as

```
fn bar<const C: T> (args) -> ReturnType { ... }
```

Is important to say that the dispatch mechanism in Rust is static.

On compilation, generic functions and generic types are specialized on demand.



## The solution: Arrays as const-generic types

With generics over constant values, we can write traits and functions for any array size

```
fn add_arr<const N: usize> (a: &[f64; N], b: &[f64; N])  
-> [f64; N] {  
    let mut result = [0.0; N];  
    for i in 0..N {  
        result[i] = a[i] + b[i];  
    }  
    result  
}
```

However, this is only a partial solution...

# The problem: Dynamical check of static bounds

Consider the following generic function

```
fn <const N: usize> head(a: [f64; N]) -> Option<f64> {  
    if N > 0 {  
        Some(a[0])  
    } else {  
        None  
    }  
}
```

The restriction  $N > 0$  is checked at runtime even though  $N$  will not change after compilation

## The solution: Bounded generics over constant values

Given that now constants are valid parameters. Rust's type checker could take care of checking properties over them.

If `P: bool` is a boolean expression depending on a constant `C`, we can restrict `C` as a parameter

```
// A generic type over a bounded constant  
Foo<const C: T> with P  
// A generic function over a bounded constant  
fn bar<const C: T> (args) -> ReturnType with P { ... }
```

Now, the type checker enforces that any instance of `Foo` and `bar` over a constant satisfies `P`

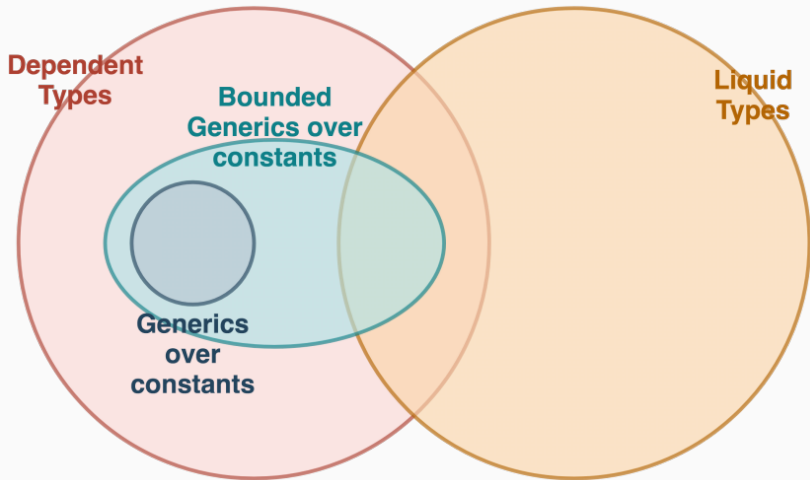
## The solution: Statical check of static bounds

Now, we can write static bounds over constant parameters.

```
fn <const N: usize> head(a: [f64; N]) with {N > 0} -> f64 {  
    a[0]  
}
```

Given that the bound  $N > 0$  is checked during compilation and not in runtime, this code is not just shorter, it is also faster.

## Related work: Generics over values in theory



## Related work: Languages with dependent types

There are already languages with dependent types:

- Haskell: A general purpose pure functional programming language.
  - It supports dependent types using extensions.
  - It has a liquid version.
- Idris: A general purpose pure functional programming language with dependent types.
- Coq: An interactive theorem prover written in OCaml.
- Agda: An interactive theorem prover written in Haskell.

On 2018, there were around 15 papers about those languages and dependent types in POPL, OOPSLA, PLDI and ICFP.

The Rust community is already working on an implementation of generics over constants. However, this implementation still needs:

- A mechanism to determine when two constant expressions are equal.
- A mechanism to add bounds over constant parameters.
- A mechanism to determine when two bounds are equal.

## Road Ahead: What needs to be done

We want to:

- Provide an unification algorithm for constant expressions to determine type equality on generic types over constants.
- Modify the parser and the subsequent compiler stages to add the `with` syntax.
- Provide unification for boolean arithmetic expressions to determine type equality on bounded generic types over constants.
- Integrate these changes into Chalk, the PROLOG interpreter written in Rust, as it will be integrated into the type checking process.



# Road Ahead: Unification

Rust compilation has several stages

Rust  $\rightarrow$  AST  $\rightarrow$  HIR  $\rightarrow$  MIR  $\rightarrow$  miri  $\rightarrow$  LLVM IR  $\rightarrow$  Machine Language

The unification algorithm for constant expressions can be implemented in

- HIR or AST: Some constants could be in external libraries.
- MIR: Control flow would make unification too complex.
- miri: A simple unification algorithm could be done.

We are still deciding which alternative is better.

We will

- Provide formal verification of the unification algorithm either by hand or using a proof assistant (Coq).
- Compare the capabilities of Rust with this new set of features against other dependently typed languages.
- Seek to integrate this work into Rust during 2019.

