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!";
```

we would say `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`).

Context: What is a type?

To be more specific:

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

Usually type systems are described using *typing rules*:

$$\textit{true} : \texttt{Bool}$$

$$\textit{false} : \texttt{Bool}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \ : T}$$

Context: The Rust programming language

Rust is a systems programming language focused on safety, speed and concurrency. However, Rust is not your typical language:

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

Context: Rust's type system

Rust's type system is based on the ML type system and it has

- **Static type checking:** Programs are checked for safety during compilation.
- **Type inference:** Type annotations are not always needed.
- **Polymorphism** via traits and generics.
- **Algebraic data types:** It has 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 inmutable references.

The problem: Functions over arrays

In Rust:

- Arrays (being stack allocated) must be statically sized.
- Writing functions or traits for arrays is cumbersome.

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.
- Writing functions or traits for arrays is cumbersome.

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: Constant values as type parameters

The result: 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 adds a new problem about constant values as parameters

The problem: Checking bounds

Even if we had generics over constant values, we still have to do dynamical checks for constant values

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

Even though the compiler has the value for N (it is a constant), we are doing checks over N at run-time.

The solution: Bounds over value parameters

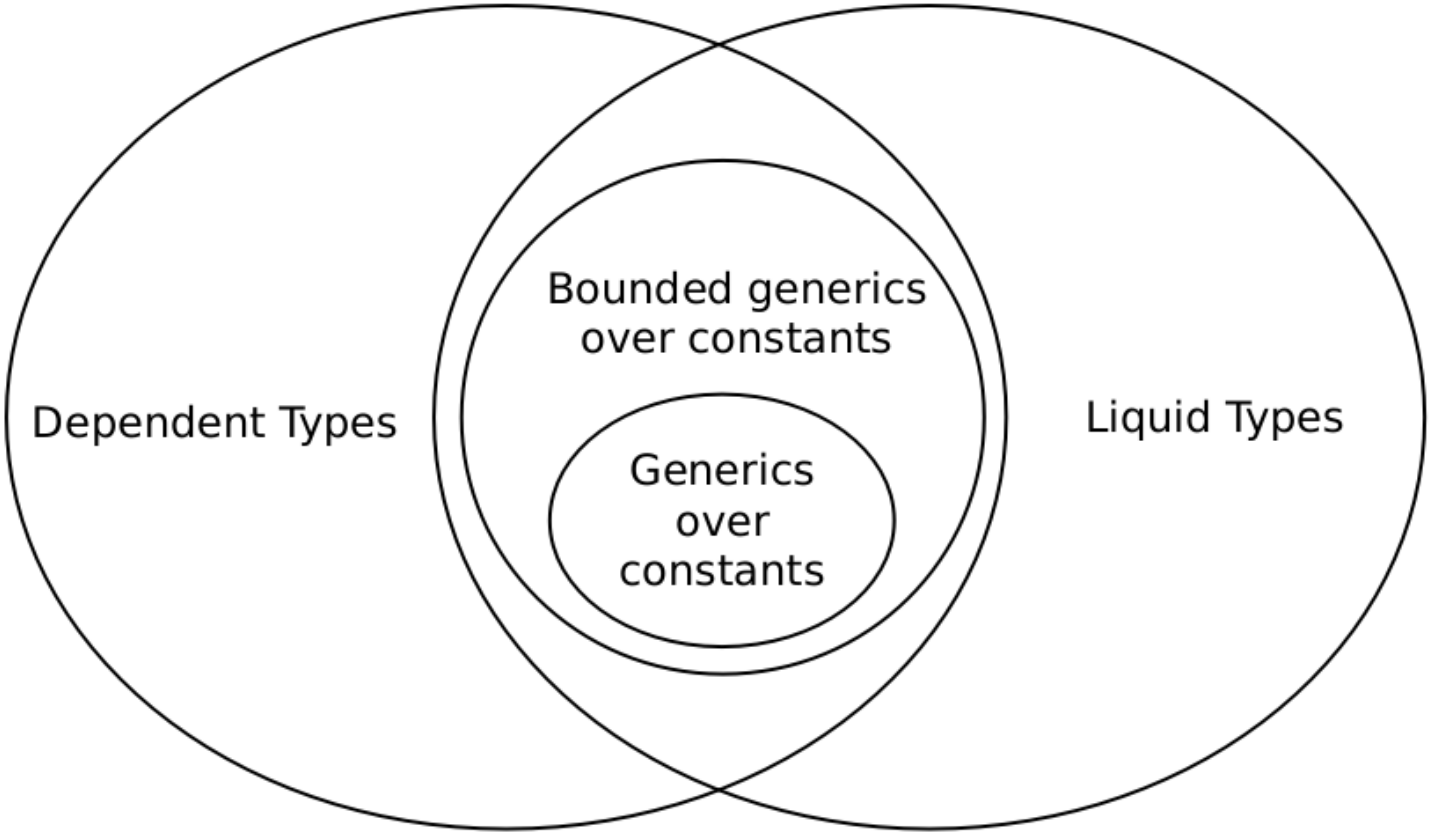
The result: Checking bounds

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 run-time. This code is not just shorter, is faster.

Context: Generics over values in theory



Context: Languages with dependent/liquid types

Haskell, Idris, Agda, Coq, Otros. Mostrar número de papers en el último año para cada uno

Context: Idris, a dependently typed language

Como se ve Idris y que deja hacer

Context: Rust Status Quo

RFC-2000, que hay, que habrá y que quedará faltando

Road Ahead: What needs to be done

Desbaratar los ejemplos para mostrar que pasos hay que seguir, entre ellos unificación

Road Ahead: Unification

Mostrar que alternativas hay para implementarla

Validation: Formal verification

Proveer una prueba formal de que unificación y bounds son bien comportados

Validation: Comparison against other languages

Con estas nuevas features hasta donde puede dar Rust al compararlo con lenguajes como Idris

Validation: Integration with Rust

Integrar este trabajo dentro de Rust como tal, RFCs y PRs al respecto

Schedule

El cronograma... para (no) cumplirlo