Master of Systems and Computing Engineering

Universidad de los Andes

Master's Thesis

Bounded generics over constants in Rust

Christian Nicanor Poveda Ruiz

Master of Systems and Computing Engineering

Universidad de los Andes

Master's Thesis

Bounded generics over constants in Rust

Author:	Christian Nicanor Poveda Ruiz
Internal examiner:	Silvia Takahashi Rodriguez
External examiner:	Oliver Scherer
Advisor:	Nicolás Cardozo Alvarez
Submission Date:	2019-05-24

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

2019-05-24 Christian Nicanor Poveda Ruiz

To my wife Diva, who never went to sleep until we had an answer.

Acknowledgments

Thanks to the Rust community for providing a welcoming space where everyone can learn about this wonderful language, in particular to Steve Klabnik and the rest of the Rust team for always having ease to learn into their goals. I would also like to thank particularly to Eduard-Mihai Burtescu, Ralf Jung and Oliver Scherer whose explanations and conversations with me were the basis for this work.

Nicolás Cardozo has my full gratitude for welcoming an unknown physics graduate who wanted to learn computer science into his office. Thank you for always having time for my questions and guiding me though this new academic path that I have chosen. Also thanks to the rest of the FLAGLAB group for keeping the C alive in DISC.

Thanks to my parents for all the support and love through my life. Thank you for always putting me first and giving me the best education. Thank you mom for teaching me how to be an exceptional person by example. Thank you dad for answering every single one of my 8-year-old questions.

Before ending these words, I want to thank my wife Diva for being the unconditional partner who always supported me. Thank you for sacrificing a comfortable life with me for our goals and dreams, and for taking care of everything so I could finish my Master. Now we have a new challenge ahead, and I am not scared of taking it by your side. I love you (and Milo does too).

Abstract

Rust is a language aiming to provide a way to write robust and performant code without using garbage collection or manual memory managment. Yet it lacks of a mechanism to abstract constants, forcing users to repeat code in cases where constants are involved. One of the challenges of developing this mechanism, is that it must be compatible with the language's core principles of performance, reliability and productivity, putting it in contrast with well established languages such as C++, where the mechanism for constant abstraction is another source of unsafety. In order to provide such a mechanism, we propose and develop a symbolic execution engine for the Rust compiler. We explore the integration of the engine in the language's trait solving and type inference mechanisms.

Contents

Acknowledgements ix

Abstract xi

1 Introduction 1

2 Preliminaries 3
- 2.1 Mutability 4
- 2.2 Memory management 4
  - 2.2.1 Ownership 5
  - 2.2.2 Borrowing 6
  - 2.2.3 Lifetimes 7
- 2.3 Algebraic data types 7
- 2.4 Control flow 9
- 2.5 Traits 10
- 2.6 Generics 11
- 2.7 Error handling 13
- 2.8 Macros 15
- 2.9 Intermediate representations 16

3 Motivation 17
- 3.1 Trait implementations for arrays 17
- 3.2 Static control flow and optimizations 18

4 Related Work 21
- 4.1 Dependently typed programming languages 21
- 4.2 Template metaprogramming 22
- 4.3 The state of Rust 23
  - 4.3.1 The typenum crate 23
  - 4.3.2 RFC-2000 24

5 Solution **27**
5.1 Symbolic execution of Rust programs 27
5.1.1 Symbolic intermediate representation 28
5.1.2 Equality of symbolic functions 32
5.1.3 Type inference 34
5.2 Bounds for generics over constants 34
5.3 Generic traits over constants 36

6 Validation **39**

7 Conclusion and Future work **43**
7.1 Conclusion 43
7.2 Future work 43

Bibliography **47**

1 Introduction

The current ecosystem for systems programming has been based around having powerful and fast programming languages, mainly C and C++. Such languages make the programmer responsible for checking its code for stack overflow errors, dangling pointers, and data races; forcing them to code inefficiently to avoid such errors.

Modern programming languages try to solve this problem by adding runtime mechanisms to guarantee memory safety, such as garbage collection, making them unsuitable for systems programming, where resource efficiency is vital.

The Rust programming language offers an alternative to C and C++ for systems programming, using compile time mechanisms to guarantee memory safety even for concurrent applications, without sacrificing performance during execution.
Even then, the current state of ergonomics in Rust can be improved and several efforts are being done by the Rust team and the community to improve the language on these regards.

One of the weaknesses of Rust is its lack of expressivenes when constant expressions need to be abstracted from code. This limits the capabilities of the language regarding the implementation of traits and functions for arrays of all sizes and it also limits the optimization capabilities of the compiler regarding constant propagation.

This work proposes a solution to such problems by interfacing the Rust compiler with a symbolic execution engine to allow a basic form of dependent types which will only allow constant values as parameters for generic types. This work also discusses new possible features of the compiler regarding type checking, trait specialization and generic types:

- Chapter 2 offers an introduction to the Rust programming language syntax, features and compiler internals relevant to this work. This chapter does not intend to be a full introduction to the language, just an attempt to set common terminology with the reader.

- Chapter 3 contains a series of code examples of the problems commonly found when working with types depending over constant values. The two problems discussed in this chapter are the implementation of traits over the type of arrays and the optimizations done by the Rust's compiler when handling static control flow. These problems are discussed in terms of their ergonomics, readability, compilation and execution performance.
- Chapter 4 discusses related work from a theoretical and practical perspective. It includes a section about the different approaches to abstracting constants done by two languages: the C++ template system, and the work done on Haskell's type system by Gundry [7] and Eisenberg [6]. Finally, the current state of Rust regarding dependent types is discussed in the last section, which includes the current partial solution using macros, RFC-2000 [1] and the typenum crate. [2]
- Chapter 5 proposes a solution to the problems discussed in chapter 3, based on the idea of adding dependent types over constant values, known in the Rust community as generics over constant values, and a possible extension adding bounds to this new kind of generic parameters.
- Chapter 6 presents the validation of our work. This is done comparing the implementation differences between an small vector library with and without our proposals.
- Chapter 7 presents the conclusion and discusses avenues of future work.

---

[1] `https://github.com/rust-lang/rfcs/blob/master/text/2000-const-generics.md`

[2] `https://crates.io/crates/typenum`

2 Preliminaries

Rust is a systems programming language focused on speed, memory safety and concurrency.[1] It intends to offer both, performance similar to C++, and memory safety similar to Haskell. Rust is a compiled language combining imperative and functional programming features. The focus on performance makes Rust an ideal candidate for writing operating systems, databases, compilers, and other high-performance software without worrying about manual memory allocation. However, Rust's safety and high-level abstractions has encouraged its usage in backend and even frontend development.

Rust started as a personal project of Graydon Hoare during 2006, who wanted to write a memory-safe, suited for a concurrent and compiled language. After three years, Mozilla endorsed the project and Rust was announced to the public in 2010. Since then Rust's development has been completely open to the community. In 2013, Hoare stepped down as the technical leader of the project and a core team for the project was formally established [9]. Version 1.0 of the language was released in May, 2015, following a six weeks release cycle enforcing semantic versioning. The current version of the Rust compiler[2] is completely backwards compatible with version 1.0.

Currently, several companies have written part of their core applications in Rust (including Mozilla, which is working on its next generation web engine: Servo[3]) as can be seen in the Rust Friends website.[4]

In the remaining of this section, we present a quick tour of the Rust programming language main features. Readers more experienced with Rust can skip this explanation.

---

[1] `https://www.rust-lang.org/`

[2] The current version is version 1.34.1

[3] `https://servo.org/`

[4] `https://www.rust-lang.org/en-US/friends.html`

2.1 Mutability

Variable declarations are done using the **`let`** keyword; all variables are immutable by default as shown in Listing 2.1.

```rust
fn main() {
    let x = 5;
    x += 1; // stderr: cannot assign twice to immutable
            // variable x
    println!("{}", x);
}
```

Listing 2.1: Trying to modify an immutable value will result in a compilation error

Mutability is allowed using the **`mut`** keyword as shown in the declaration of the variable x in Listing 2.2. The explicitness of mutability not only allows the compiler to do optimizations, it is also useful to the programmer, if a variable is not declared explicitly as mutable, its value will not change during its whole lifetime.

```rust
fn main() {
    let mut x = 5;
    x += 1;
    println!("{}", x); // stdout: 6
}
```

Listing 2.2: Mutability is allowed but it must be explicit

2.2 Memory management

In Rust, memory safety checks are done at compile time, avoiding the need for a garbage collector or manual memory management. The Rust compiler can reason about memory usage via three concepts: Ownership, borrowing, and lifetimes.

2.2.1 Ownership

The semantics of value assignment in Rust differs from the semantics of C/C++ or Java. When assigning a value to a variable the state of the program changes as usual, but the variable becomes the new *owner* of such value [4]. This means that the value will be dropped when its owner goes out of scope. Each value can only have a single owner at the same time. Meaning that for each value stored in memory there is exactly a single variable owning it.

When a value is reassigned to another variable, ownership is transferred i.e, the former owner variable loses the ownership and it cannot be used again unless a new value is assigned to it. As a consequence, when a variable is used as a function argument, the variable loses ownership of its value, and the variable representing the argument of the function becomes the new owner. This behavior can be seen in Listing 2.3.

```
fn exclamate(z: String) {
    println!("{}!", z);
}

fn main() {
    let x = String::from("Hello, world");
    let mut y = x; // y is the new owner, x is invalid.
    exclamate(y); // z is the new owner, y is invalid.
    println!("{}", x); // stderr: use of moved value: x
    println!("{}", y); // stderr: use of moved value: y
}
```

Listing 2.3: Ownership transfer

The ownership restriction has two advantages: First, it is impossible to have a value stored in memory without a variable in the current scope assigned to it, avoiding some memory leaks. Second, is impossible to modify a single value from several different threads, avoiding data races. Nevertheless, ownership forces the programmer to write code akin to continuation-passing style, which is error-prone and difficult to read. The *borrowing* concept (described in the next subsection) solves this issue.

2.2.2 Borrowing

Rust is a language with references. When a reference to a value is created, such value is being borrowed by the reference variable (but ownership is not transferred). There are two kind of references in Rust: immutable references denoted by `&T` and mutable references `&mut T`. Immutable references allow "read-only" access, independently of the referenced variable mutability. Mutable references allow "read and write" access, but they only can reference mutable variables. Examples of both kinds of references can be found in Listings 2.4 and 2.5 respectively.

```
fn exclamate(z: &String) {
    println!("{}!", z);
}

fn main() {
    let x = String::from("Hello, world");
    exclamate(&x); // z is borrowing the value owned by x.
                   // stdout: Hello, world!
    println!("{}", x); // stdout: Hello, world
}
```

Listing 2.4: References avoid the need for ownership transfer

There are three rules about borrowing enforced by the compiler:

- Several immutable references to a value can exist at a given time.
- There can be at most one mutable reference to a value at a given time.
- The first two scenarios are exclusive, only one of them can happen at the same time.

In other words, is possible to have several readers or, have a single writer. This prevents the mutation of shared state and, as a consequence, prevents data races in concurrent applications. There are certain scenarios where the borrowing rules are not flexible enough to allow certain kind of behaviors, such as locks for example, in those cases is possible to use types with internal mutability, the `Mutex` type is an example of this.

```
fn exclamate(z: &mut String) {
    z += &"!";
}

fn main() {
    let mut x = String::from("Hello, world");
    exclamate(&mut x); // z is borrowing the value owned
                       // by x.
    println!("{}", x); // stdout: Hello, world!
}
```

Listing 2.5: Mutable references allow mutation of the borrowed value

2.2.3 Lifetimes

Adding references to a language makes possible to have dangling references. When a value is dropped, all its references become dangling references, no longer pointing to a valid memory location. To solve this memory issue, Rust introduces the concept of lifetimes, each value has a lifetime which starts when the value is allocated and ends when the value is dropped. As a consequence, a value lifetime ends when its owner goes out of scope. The Rust compiler has a "borrow checker", which compares scopes to check that no reference outlives the value being referenced. If this is not the case, a compilation error occurs.

In principle, every reference needs a lifetime annotation. However, this is avoided by a process known as lifetime elision, where lifetimes are inferred automatically by the compiler. Even then, in certain cases the programmer might need to add such annotations. However, this will be discussed in the generics section.

2.3 Algebraic data types

Rust has both sum and product algebraic data types in the form of structures and enumerations respectively.

```
struct Pixel {
    red: u8,
    green: u8,
    blue: u8,
}

fn main() {
    let yel = Pixel {
        red: 255,
        green: 255,
        blue: 0
    };

    println!("({}, {} ,{})", yel.red, yel.green, yel.blue);
}
```

Listing 2.6: A structure representing the color of a pixel

Structures are named sum types, where an struct type has a fixed set of named fields. When declaring a new value of an struct type, all its fields must be given as shown in Listing 2.6. Enumerations are named product types, where an enum type has a fixed set of named variants. When declaring a new value of an enum type, one single variant must be chosen as shown in Listing 2.7.

```
enum Color {
    RGB(u8, u8, u8),
    CMYK(u8, u8, u8, u8),
}

fn main() {
    let yel = Color::RGB(255,255,0);
    let other_yel = Color::CMYK(0, 0, 90, 0);
    ...
}
```

Listing 2.7: An enumeration representing colors in different color systems

It is also possible to add associated functions to any type using the **`impl`** keyword in an object oriented programming style. Each associated function may or may not use an instance of a given type. This is similar, for example, to the way instance and static methods are defined in Java.

2.4 Control flow

Control flow in Rust is realized using the common **`if`**/**`else`** conditional and **`while`** loop statements, as in other programming languages. **`for`** loops are iterator based, where the variable to be iterated must implement the `Iterator` trait.

Rust has pattern matching capabilities thanks to its functional heritage, pattern matching in Rust can be used to match specific values or to destructure tuples, arrays, enumerations or structures. There are three statements for pattern matching: **`match`**, **`if let`** and **`while let`**. However, the last two are just syntactic sugar for the first. An example of pattern matching can be found on Listing 2.8.

```rust
enum List {
    Empty,
    Cons(i32, Box<List>)
}

impl List {
    fn length (&self) -> usize {
        match self {
            List::Empty => 0,
            List::Cons(_, cdr) => 1 + cdr.length()
        }
    }
}
```

Listing 2.8: Computing the length of a list using the **`match`** statement

2.5 Traits

Traits are Rust's mechanism to allow ad-hoc polymorphism; they allow to extend the behavior of a type requiring that the type implements a set of methods defined by the trait. The main difference between using traits instead of generic functions consist in the possibility to use concrete properties of an specific type when implementing the trait [5].

```rust
trait Volatile {
    fn explode(&self);
}

impl Volatile for i32 {
    fn explode(&self) {
        for _ in 0..*self {
            println!("Boom!");
        }
    }
}
```

Listing 2.9: Implementation of an user defined trait for a foreign type

User defined traits can be implemented for any type, in contrast to Java interfaces where the implementations are restricted to the types declared in the same package as the interface, as in Listing 2.9. On the other hand, the user can implement a foreign trait for its own types, as in Listing 2.10. However, the user can not implement foreign traits for foreign types as shown in Listing 2.11.

Traits can have associated types, such types can be used in the signature of the trait associated functions to allow more expressiveness, as an example, in Listing 2.9, `Output` is an associated type of `Add` and it can be used as the return type of the `add` method, allowing the addition of two variables of the same type, return a different type. It is also possible to parametrize traits using types and lifetimes, i.e., generic traits.

Traits are used for operator overloading, e.g., the types that can be operated with `+` must implement the `Add` trait of the standard library, Listing 2.10 shows this.

```
use std::ops::Add;

struct Rational {
    a: i32,
    b: i32,
}

impl Add for Rational {
    type Output = Rational;

    fn add(self, other: Rational) -> Rational {
        Rational {
            a: self.a * other.b + other.a * self.b,
            b: self.b * other.b
        }
    }
}
```

Listing 2.10: Implementation of a foreign trait for an user defined type

Thread safety is also handled using traits. Variables which are thread-safe to send must have a type implementing the `Send` trait and variables which are thread-safe to share (using references) must have a type implementing the `Sync` trait. Both traits are empty , i.e., they do not request any function to be implemented, but are marked as unsafe because the compiler can not guarantee the safety of sending or sharing values of a certain type. Such guarantees would require an analysis of lifetimes and memory access deeper than the currently avalable in the compiler.

2.6 Generics

From an external perspective, generic types in Rust are quite similar to the generic types in Java –that is, every type can have generic type parameters in order to reduce code duplication and such parameters are specified between angled brackets. However, there are three main differences between the two implementations:

- Rust allows lifetimes as generic parameters. These are used when the borrow

```
use std::ops::Add;

impl Add for bool { // stderr: only traits defined in the
                    // current crate can be implemented
                    // for arbitrary types.
    type Output = bool;

    fn add(self, other: bool) -> bool {
        self || other
    }
}
```

Listing 2.11: Implementation a foreign trait for a foreign type results in a compilation error

checker cannot infer a proper lifetime for a variable, and thus explicit lifetime annotations are required. An example of this can be found in Listing 2.12.

```
fn longest<'a>(x: &'a List, y: &'a List) -> &'a List {
    if x.length() > y.length() {
        x
    } else {
        y
    }
}
```

Listing 2.12: Returning the longest list, lifetime annotations are required because the Rust compiler cannot decide if the return value will outlive `x` and `y`.

- In Java, generic parameters can be bounded by forcing them to be instances of a class or interface. In Rust, which is not an object oriented language, bounds are done over traits instead, as in Listing 2.13.

- Internally, generics in Java are implemented doing type erasure, where each generic parameter bounds are checked, and then the Java compiler forgets about the type of such paremeter and uses dynamic dispatch over the type

```
fn inc<T: AddAssign + Copy>(vec: &mut Vec<T>, val: T) {
    for elem in vec {
        *elem += val;
    }
}
```

Listing 2.13: A function which increments the elements of a vector by a fixed value, this can only be done if the type of the elements `T` implements both the `AddAssign` and `Copy` traits.

`Object`. On the other hand, Rust uses monomorphization, where the compiler builds a new type specialized for each instance of a generic parameter making all the dispatch completely static.[5]

2.7 Error handling

It is common to use exceptions in imperative languages to represent errors. However, having a functional influence, Rust has two kind of errors:

- Recoverable errors, where the program execution is not interrupted and is possible to handle the error.
- Unrecoverable errors, where the program execution stops and memory is cleaned.

Recoverable errors are written by returning a variable of type `Result<T, E>`, which is an enumeration with two variants: the `Ok<T>` variant, containing the result of a successful operation, and the `Err<E>` variant, containing the failure reason of a failed operation. If a function returns a value of type `Result<T, E>` the programmer must explicitly handle both variants, making error handling explicit every time. Error propagation can be done using the `?` operator, which does an early return if the expression returns an `Err<E>`. An example of this can be seen in Listing 2.14.

---

[5] Rust allows type erasure via trait objects. However, this goes is beyond the scope of this work.

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_file(path: &str) -> Result<String, io::Error> {
    let mut string = String::new();
    // if opening the file fails, the error is returned
    let mut file = File::open(path)?;
    // if reading the file fails, the error is returned
    file.read_to_string(&mut string)?;
    // if nothing fails, the string is returned
    Ok(string)
}
```

Listing 2.14: A function returning a recoverable error, doing error propagation

Unrecoverable errors are written using the macro `panic!`, having as argument a string; the panic reason. Unrecoverable errors are reserved for cases when execution would cause undefined behavior or when it does not make sense to keep executing the program after the error was reached. Because of this, unrecoverable errors are most commonly used in code handling memory or other low-level operations, such as vector insertion[6], or when the logic of the application dictates the application must crash, as seen in Listing 2.15.

```
fn main() {
    match read_file("./config") {
        Ok(string) => { ... }
        Err(_) => panic!("Cannot run without configuration")
    }
}
```

Listing 2.15: A function panicking after a critical error

---

[6] https://doc.rust-lang.org/std/vec/struct.Vec.html#method.push

2.8 Macros

Macros are the mechanism used in Rust to allow metaprogramming. Rust's capabilities on this matter are limited, but macros are widely used given the static nature of the language. Currently Rust has two kind of macros: procedural and declarative. We present declarative macros in the following, as procedural macros are not relevant for the development or application of our work.

Declarative macros, which are called using the name of the macro, followed by the `!` symbol, allow to manipulate Rust code in a pattern matched way. These macros are declared using the macro `macro_rules!`, which take the name of the macro and a series of code patterns like a **`match`** statement.

Most of the time, declarative macros are used as a replacement for variadic functions (functions with a variable number of arguments), given that Rust only admits functions with a fixed number of arguments. For example, `vec!` and `println!` are macros because they may receive a variable number of elements: `vec![1]`, `vec![1, 2]`, `vec![1, 2, 3]` and so on.

```
macro_rules! list {
    ($head:tt $(, $tail:tt)*) => {
        List::Cons(
            $head,
            Box::new(list!($($tail),*))
        )
    };
    () => {List::Empty};
}

fn main() {
    let x = list!(1,2,3);
    println!("{}", x.length()); // stdout: 3
}
```

Listing 2.16: A macro based constructor for lists

2.9 Intermediate representations

Rust code is not compiled directly into machine code, instead is compiled into a series of intermediate representations. In early versions, Rust code passed through a desugaring process (this representation is known as the high-level intermediate representation or HIR) before being compiled into the LLVM intermediate representation, and finally it is compiled to machine code.

During 2016, Rust added the mid-level intermediate representation or MIR to its compilation pipeline. This new representation improved the borrow checking and optimization processes, allowing for both faster and more readable code.[7]

On this same year, miri, an intepreter for the MIR was written. This interpreter can execute code written in the MIR directly instead of compiling it to machine code.[8] However, Rust is still a compiled language, miri is not used currently by the Rust project as a "virtual machine". Instead, some components of miri are used to evaluate constant expressions during compilation.

---

[7] https://rust-lang.github.io/rustc-guide/mir/index.html

[8] https://solson.me/miri-report.pdf

3 Motivation

One of the main objectives of Rust is to reduce the number of trade-offs for the programmer. For example, fast code should not be unsafe, and language abstractions should not have significant performance costs. Rust accomplishes this by having a compiler that is able to reason more deeply about the code than usual. Rust can reason about the lifetime of each variable to avoid dangling pointers, or about mutability to avoid data races.

However, some improvements could be done in the interaction of the language with constant values, not necessarily from a performance perspective, but also from an ergonomic perspective. This chapter will explain some aspects of the compilation and reasoning processes that could be improved. Chapter 5 addresses the solution for the problems posit hereinafter.

3.1 Trait implementations for arrays

In Rust, the programmer has the capability to store data in the heap and the stack. On the one hand, stack allocated values must have an static size (it must be known at compile time). On the other hand, heap allocated values can have dynamic size, but they can only be accessed using references. The differences between vectors and arrays are a perfect example of this trade-off: Vectors, being heap allocated, can increase or decrease in size during execution, but having nested vectors causes performance issues, because access must be done using nested references. Arrays, being stack allocated, have no performance issues when nesting them, but their size must be fixed during execution.

These limitations create some expresiveness problems with arrays: Two arrays with different sizes have different types, even if both store values of the same type. Thus, trait implementations for array types must be done manually for each possible size. Because of this, the standard library only implements traits for array types up to a size of 32, even though arrays are a primitive type of the language.

```
const N: usize = 3;

trait Volatile {
    fn explode(&self);
}

impl<T> Volatile for [T; N] {
    fn explode(&self) {
        for _ in self {
            println!("Boom!")
        }
    }
}

fn main() {
    [0i32, 1, 2].explode();
    [0i32, 1].explode(); // stderr: no method named explode
                         // found for type [i32; 2] in the
                         // current scope
}
```

Listing 3.1: Even though `Volatile` is implemented for `[T; 3]`, it is not for `[T; 2]`.

In certain cases, vectors are used instead of arrays. Making the code unsuitable for applications running on embedded devices (which may not allow heap allocation) or applications needing multidimensional vectors.

An example of this limitation is shown in Listing 3.1, where the implementation of the trait `Volatile` cannot be easily extended to other array sizes, even when such implementation does not use the size of the array.

3.2 Static control flow and optimizations

One of the recent efforts to improve Rust's performance is constant propagation. If a constant can be resolved during compilation, the compiler will propagate the constant through the code. For example, if the condition of an if-else statement can

```
const N: usize = 0;

fn head<T>(array: &[T; N]) -> Option<&T> {
    if N > 0 {
        Some(&array[0])
    } else {
        None
    }
}
```

Listing 3.2: This function must be optimized to improve performance.

```
fn head<T>(array: &[T; 0]) -> Option<&T> {
    None
}
```

Listing 3.3: This function should be equivalent to the one in Listing 3.2 after constant propagation.

be resolved during compilation, the compiler will replace the statement with one just including the matching branch of the statement.

An small illustration of this can be seen in Listing 3.2, where the array size `N` is a constant value known during compilation. Given that the expression `N > 0` can be evaluated during compilation, the compiler will optimize the function, resulting in code similar to the code shown in Listing 3.3.

However, if a mechanism to abstract code over constant were added to Rust, the constant propagation mechanism would not be able to optimize code with unresolved constants. For example, if the value of `N` were not known in Listing 3.2, it would not be possible to optimize the `head` function even if always were used in empty arrays.

4 Related Work

Abstracting over constants is a similar problem to abstracting over types as with generic types. Thus, it is no surprise that the solutions for the problems in Chapter 3 are often related to parametric polymorphism. This chapter discusses mechanisms to parametrize code over constants, both from a formal and an implementation point of view.

4.1 Dependently typed programming languages

There are several formalism carrying the name of dependent types. However, most of the time dependent types are regarded as functions mapping values to types. For example, the collection of types `[N; i32]` in Rust can be thought as a function which takes an integer `N` and returns a type `[N; i32]`. This particular formalism is known as Π-types or dependent product types. Π-types solve most of the problems discussed in Chapter 3, because they provide a mechanism to write types from unresolved constants, and to do typechecking over them [11].

Under the Curry-Howard isomorphism, Π-types and are equivalent to a predicate logic theory with universal quantifiers. Thus, is not surprising that early programming languages with dependent types were proof assistants, such as Coq and Agda [10], both of which can be used to proof theorems automatically. However, such languages are not used outside the academic community nor to write user or system oriented applications.

Haskell, being both a general purpose programming language and the de facto language to explore and study type systems in academia, has provided the foundation to study several formulations of dependent types in the form of new languages, such as Agda, Idris [3] and Cayenne [2], or by adding dependent types directly to the language. This last part is particularly relevant to our work, given that neither Rust's nor Haskell's type system and compiler were written to have dependent types.

The Glasgow Haskell compiler has certain features which makes Haskell almost a dependently typed language. For example, datatype promotion allows algebraic data types to be promoted as kinds, and each term of such types to be promoted as a type itself. With these promotions, traditional polymorphism and type families can be seen as a form of dependent types with some restrictions. This is because type families are functions between types and, if every term of a datatype can be promoted to its own type, type families are a construct mapping terms to types.

The addition of fully dependent types to Haskell's type system has been studied deeply by Gundry [7] and subsequently by Eisenberg [6]. Both authors add dependent types to Haskell by writing an intermediate language, the *evidence* language for Gundry's work, and PICO for Eisenberg's work. These intermediate languages are compilation targets for the dependent version of Haskell, and then the intermediate languages are compiled to standard Haskell.

4.2 Template metaprogramming

C++ is the language that is most often compared to Rust because both languages were designed to write performance oriented applications. Templates, which are C++ mechanism to achieve parametric polymorphism, allow not only types as parameters but also constant values [12]. This idea is pretty similar to the one being implemented for Rust in RFC 2000. Even then, manual memory allocation makes C++ a language far from being memory safe. Where as Rust is a completely memory safe language [8].

C++ templates are widely used in the same way as generic types: They avoid code repetition by parametrizing types and functions over other types. However, templates are by themselves a powerful metaprogramming mechanism and not an organic part of the language itself [1]. This extends the capability of templates beyond traditional generic types. For example, templates allow non-type parameters in high contrast to generics in languages such as Java.

There are some restriction with non-type parameters, they only can be constant integer values, null pointers, or pointers to objects and functions [12]. Being limited to types parametrized over constant values, C++ is not a dependently typed language, neither it has the same capabilities as a dependently typed language. But being able to parametrize code over constant values is a viable solution to some of the problems aforementioned in Chapter 3. In particular, having a similar system in Rust would permit implementing traits for arrays of any size. Even then, memory safety guarantees must be preserved.

4.3 The state of Rust

Rust currently provides implementations of some traits for arrays [1]. This is done using macros to expand the implementation of every trait for each possible size between 0 and 32. Even if this approach is better than writing a separate implementation for each array size, it would not scale to all array sizes, because it produces code machine with all possible implementations even if they are not used.

Given these limitations, the Rust community has taken a different approach to abstract constants, these efforts are discussed in the rest of the section.

4.3.1 The typenum crate

The typenum [2] crate uses Rust's types and traits to encode integers and operations between them. Each type-level integer has associated functions to produce the corresponding value. For example, the type `P1` represents the possitive integer `1` and the function call `P1::as_i32()` returns an `i32` value containing `1`.

Operations are represented by generic types whose parameters are the operation's arguments. As an example, the type `Sum<A, B>` represents the output of the addition between `A` and `B`. Thus, the type `Sum<P1,P2>` is the same type as `P3`, the type-level integer representing `3`. Currently there exist an implementation of arrays generic over its size using typenum's types.[3]

Even though those crates allow to implement traits for arbitrary array sizes, there are some expressiveness drawbacks. First, manipulating these generic arrays requires a different syntax than the used by Rust's own arrays. Second, is impossible to produce a type-level integer from an integer value, making type-level integers hard to manipulate in certain situations.

---

[1] https://doc.rust-lang.org/std/primitive.array.html

[2] https://crates.io/crates/typenum/

[3] https://crates.io/crates/generic-array/

4.3.2 RFC-2000

Substantial changes to Rust must follow the RFC or request for comments process. On September 14, 2017, an RFC regarding the addition of generics over constant values was merged into the RFC repository [4]. The main motivation behind this RFC is to allow implementations of traits for all array sizes [5].

```
fn zeros<const N: usize>() -> [i32; N] {
    [0; N]
}
```

Listing 4.1: A generic function having a constant value as parameter

The proposed solution in this RFC is to add constant values to the possible kinds of generic parameters (which currently are types and lifetimes as stated in Chapter 2), the proposed syntax to declare constant parameters can be seen on Listing 4.1. This RFC imposes certain restrictions about when a constant parameter can be used, specifically constant parameters cannot be used to initialize constant or static values, functions and other types.

```
fn push_zero<const N: usize>(a: [i32, N]) -> [i32; N + 1] {
    let res: [i32; N + 1] = [0; N + 1];
    for i in 0..N {
        res[i] = a[i];
    }
    res
}
```

Listing 4.2: After implementing RFC 2000, Rust's compiler will not be able to type-check the function `push_zero`

This RFC takes structural equality as its notion of term equivalence, i. e. two values are equivalent if all their inner fields are equal. It is also mentioned that projections over constant parameters will not be considered equivalent even if they represent the same value. As an example, the compiler will not be able to typecheck the code in Listing 4.2, because the projection `N + 1` appears on different places of this code's AST and is treated as a different expression on each appearance.

---

[4] https://github.com/rust-lang/rfcs/pull/2000

[5] https://github.com/rust-lang/rfcs/blob/master/text/2000-const-generics.md

Finally, this RFC states some unresolved questions about unification and well formedness of constant expressions. In particular, it is stated that the absence of a proper unification mechanism for projections over constant parameters will result in poor user experience and that such a mechanism should be provided before stabilizing this feature.

5 Solution

Even though the changes introduced to Rust on subsection 4.3.2 add the necessary syntax to abstract code over constant expressions, Rust lacks a mechanism to reason deeply about constant expressions. As a consequence is not possible to type-check or provide any compile time guarantees over the safety of using generics over constant values. In this chapter, we introduce SIRE, a symbolic evaluator for Rust, this provides a foundation to handle function equality via an SMT solver.

Symbolic execution evaluates a program in an abstract manner, taking symbols representing each program's input, instead of concrete values, and then executing the program propagating each symbol. With a symbolic representation of a function, it is possible to reason deeply about its behaviour. In particular it is possible to decide if two functions evaluate to the same values for every possible input or which input for a function would produce a particular result or failure.

With this taken into account, Rust's compiler can be extended to reason about generic types over constant values in a similar manner to its generics over types counterpart, providing trait resolution and type inference for this new kind of generics.

The remaining part of this section explains the inner processes done by SIRE and proposes a solution to the problems stated in Chapter 3.

5.1 Symbolic execution of Rust programs

SIRE is a symbolic executor for the MIR of Rust, which is able to interact with Rust's compiler. After the compiler generates the MIR for a function, SIRE takes this representation and evaluates it into an small symbolic intermediate representation or SIR for short.

Currently SIRE can only evaluate an small subset of all possible MIR functions. Specifically, it can only evaluate functions with neither mutable arguments nor mutable references as arguments. Additionaly only the following subset of MIR can be evaluated:

- Places: `Base`.
- Statements: `Assign`, `StorageLive` and `StorageDead`.
- Terminators: `Goto`, `Return`, `Call` and `SwitchInt`.
- Rvalues: `BinaryOp`, `Ref` and `Use`.
- Operands: `Move`, `Copy` and `Constant`.

Loops between blocks are not supported, given that their execution would run indefinitely. In addition, SIRE only supports integer and boolean types. Support for structures, tuples, enumerations and arrays is planned for future work. Floating point types are not supported given that they lack of a total equality relation. [1]

SIRE has a store to read and write symbolic expressions created during evaluation. The symbolic execution of a function starts by allocating each of the function's arguments as symbols into its store, then each MIR expression is evaluated into a SIR expression. When the return statement of the MIR of the function is reached, SIRE takes the symbolic expression corresponding to the return value of the function from its store and returns it, providing a symbolic representation of the function.

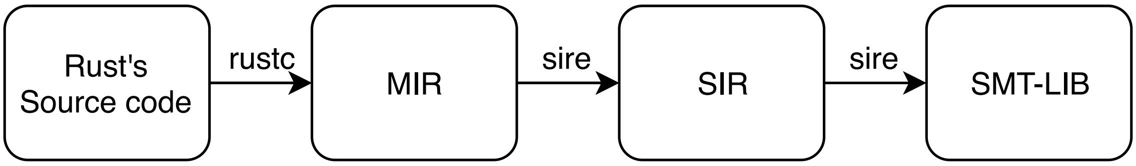


Figure 5.1: The different transformation stages of a constant function

5.1.1 Symbolic intermediate representation

SIR is a relatively simple language. Functions are the main construct of SIR, and function arguments are numbered following the same convention as MIR where the zeroth argument is the return place. The grammar for SIR can be found in Listing 5.1.

---

[1] A deeper discussion about the different notions of equality can be found on RFC-1445.

```
defun   = '(defun' name {ty} expr ')';
expr    = value | '('expr {expr}')' | '('op expr expr')' |
         '(switch' expr {'('expr '->' expr')'} defcase);
value   = '_'num | '(const' ty num')' | name;
defcase = '(else ->' expr')';
ty      = '(int' num')' | '(uint' num')' | 'bool';
num     = ? a positive integer ?;
name    = ? an string denoting the name of a function ?;
op      = ? a binary operator ?;
```

Listing 5.1: SIR's grammar in EBNF

The body of each function is composed of expressions which can be function applications, binary operations, switch statements or pure values.

There are only three kinds of values: Function arguments, constants and function names. Constants are stored as raw bits with its corresponding type.

Finally, switch statements are composed by the value to be compared, the possible values that it can take, and the result for each possible value (there must be always a default result).

`StatementKind` variant	Effect
`Assign(place, rvalue)`	Evaluate `rvalue` and store it into `place`
`StorageLive(local)`	Add the key `local` into the store
`StorageDead(local)`	Remove the key `local` from the store

Table 5.1: Evaluation of each MIR statement and the effect it has into SIRE's store.

`Operand` variant	Result
`Move(place)`	Return the value stored in `place`
`Copy(place)`	Return the value stored in `place`
`Constant(constant)`	Extract the bits of `constant` and returns a SIR constant with the corresponding type

Table 5.2: Result of the evaluation done by SIRE for each `Operand` variant.

`Rvalue` variant	Result
`BinaryOp(bin_op, op1, op2)`	Evaluate `op1` and `op2`, and build the corresponding SIR binary operation using `bin_op`
`Rvalue::Ref(_, Shared, place)`	Return the value stored in `place`
`Use(operand)`	Evaluate `operand`

Table 5.3: Result of the evaluation done by SIRE for each possible `Rvalue` variant.

`TerminatorKind` variant	Effect
`Return`	Stop execution
`Goto{target}`	Continue execution into the `target` block
`Call{func, args, destination}`	Evaluate `func` and `args`, store them in the place stated in `destination`. Continue the execution as stated in `destination`
`SwitchInt{discr, values, targets}`	Fork execution for each block in `targets` replacing `discr` by the corresponding value in `values`

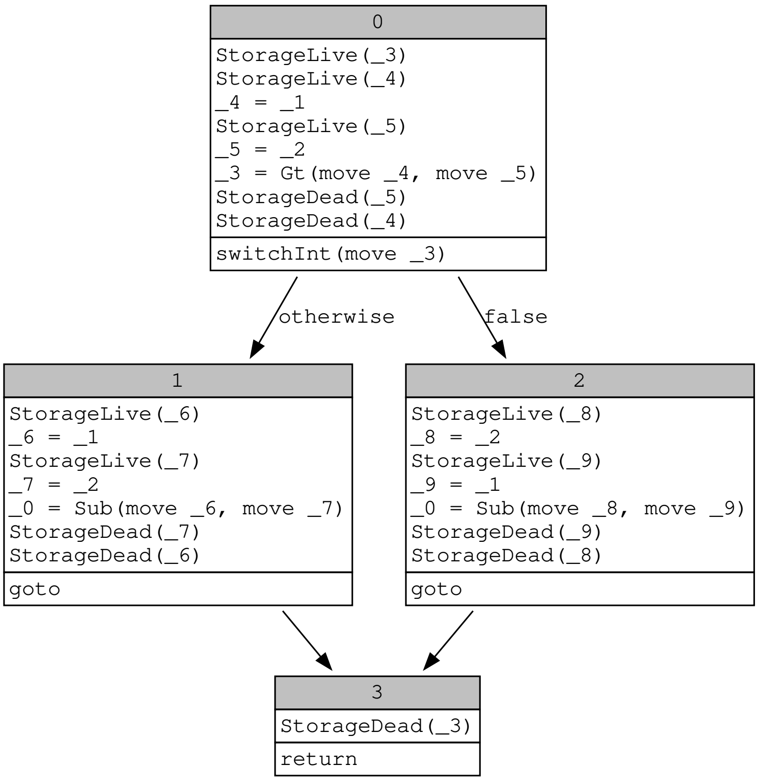
Table 5.4: Evaluation of each MIR terminator and the effect it has into SIRE's execution flow.

The evaluation done by SIRE for each one of the MIR expressions mentioned on section section 5.1 is explained on Tables 5.1-5.4.

As an example, when executing the code on Listing 5.2, the MIR and SIR generated can be found on Figure 5.2 and Listing 5.3 respectively. Even though the code on Listing 5.2 is similar to the one found on Listing 5.3, SIRE is executing each instruction of Figure 5.2 to generate this representation.

```
fn distance(x: i32, y: i32) -> i32 {
    if x > y {
        x - y
    } else {
        y - x
    }
}
```

Listing 5.2: A simple Rust function to be evaluated using SIRE



```
StorageLive(_3)
StorageLive(_4)
_4 = _1
StorageLive(_5)
_5 = _2
_3 = Gt(move _4, move _5)
StorageDead(_5)
StorageDead(_4)
switchInt(move _3)
```

```
StorageLive(_6)
_6 = _1
StorageLive(_7)
_7 = _2
_0 = Sub(move _6, move _7)
StorageDead(_7)
StorageDead(_6)
goto
```

```
StorageLive(_8)
_8 = _2
StorageLive(_9)
_9 = _1
_0 = Sub(move _8, move _9)
StorageDead(_9)
StorageDead(_8)
goto
```

```
StorageDead(_3)
return
```

Figure 5.2: The MIR of the `distance` function on Listing 5.2

```
(defun distance (int 32) (int 32) (int 32)
    (switch (> _1 _2)
        ((const bool 0) -> (- _2 _1))
         (else -> (- _1 _2))))
```

Listing 5.3: The SIR of the `distance` function on Listing 5.2

```
(define-fun distance
 ((x1 (_ BitVec 32)) (x2 (_ BitVec 32))) (_ BitVec 32)
 (ite (bvsgt x1 x2) (bvsub x1 x2) (bvsub x2 x1)))
```

Listing 5.4: The SMT-LIB snippet for the SIR of the `distance` function on Listing 5.2

5.1.2 Equality of symbolic functions

Two SIR functions are considered equal if they have the same type and evaluate to the same expression for every possible argument. For simple arithmetic expressions, this could be achieved via E-unification. However, SIR functions contain control flow operations and recursive calls, in this case a theorem prover such as Z3 is up to the task. SIRE can transform every SIR function into an small snippet in the SMT-LIB format to use it in an SMT solver.

```
fn alt_dist(x: i32, y: i32) -> i32 {
    let sign: i32;
    if x > y {
       sign = 1;
    } else {
       sign = -1;
    }
    sign * (x - y)
}
```

Listing 5.5: An alternative implementation of the `distance` function on Listing 5.2

Integer types on SIR are transformed into SMT-LIB bit vectors of the corresponding length and the signed or unsigned arithmetic operations are transformed according to the type of the operands. Switch expressions are transformed into nested conditionals, preserving the order of the switch expression's branches. An example of such snippets can be found on Listing 5.4.

To decide if two functions are equal, is enough to write an small SMT-LIB snippet checking if the two functions are equal in their whole range. In Listing 5.6, the `distance` function is compared with an alternative implementation found on Listing 5.5.

It is important to consider that the SMT solver might find no answer to one of this queries under time constrains. This is because in the worst case scenario, some SMT solvers will try to check every possible output to decide if two functions are equal.

```
;; Definition of distance provided by sire
(define-fun distance
 ((x1 (_ BitVec 32)) (x2 (_ BitVec 32))) (_ BitVec 32)
 (ite (bvsgt x1 x2) (bvsub x1 x2) (bvsub x2 x1)))

;; Definition of alt_dist provided by sire
(define-fun alt_dist
 ((x1 (_ BitVec 32)) (x2 (_ BitVec 32))) (_ BitVec 32)
 (ite (bvsgt x1 x2)
  (bvmul (_ bv1 32) (bvsub x1 x2))
  (bvmul (_ bv4294967295 32) (bvsub x1 x2))))

;; Assert that the functions are equal
(assert (forall ((x1 (_ BitVec 32)) (x2 (_ BitVec 32)))
 (= (distance x1 x2) (alt_dist x1 x2))))

;; Check if the assertions can be satisfied
(check-sat) ; sat
```

Listing 5.6: Equality check between the `distance` and `alt_dist` functions

5.1.3 Type inference

Type inference is the process of inferring the type of an expression. This is what makes it possible to write Rust code with almost no type annotations. The Rust compiler does type inference by adding constrains over inference variables and then trying to bind each variable to a concrete type.

The relevant constrains for this work are equality constrains, meaning that a type variable must be equal to another. Subtyping constrains are not discussed here, given that the subtyping relation is only relevant when dealing with lifetimes.

When an equality constrain is added, and both the type variables correspond to types with constants as generics parameters, this equality constrain over types can be reduced over an equality constrain over its constant parameters, i.e., as a constrain of equality between functions, which can be represented and solved using SIRE.

Note that it is possible that an SMT solver is unable to decide if a clause is satisfiable or not. For example, is possible to encode a root finding problem as a function equality problem, these kind of problems do not have an analytic solution for high degree polynomials.

As a consequence, is not possible to perform type checking nor type inference for every program with generics over constant values using symbolic execution.

5.2 Bounds for generics over constants

As explained in Section 2.5, Rust's generic type parameters can be constrained indicating that the type parameters must implement an specific trait. Extending this behavior to constant parameters would allow the user to constrain constant expressions using predicates over them, an example of this can be seen on Listing 5.7, where the `head` function's constant parameter `N` is bounded to be strictly positive, guaranteeing that the first element of the array always exists.

The difference between this implementation and a traditional one as in Listing 5.8 is the stage in which the condition `N > 0` is checked. When using the function provided in Listing 5.7, the condition is verified in compilation, during the monomorphization stage. On the other case, the condition is verified also in compilation, during the constant propagation stage, or in a case where the compiler is doing few

```
fn head<T, const N: usize>(array: &[T; N]) -> &T
where {N > 0} {
    &array[0]
}
```

Listing 5.7: Type-safe access to the first element of a non-empty array using bounded generics

optimizations, the condition would be verified every time the function is called during execution.

Using the `Option` type also introduces an small memory overhead, given that enumerations require additional memory space to indicate which variant is being used. This being said, using bounded generics over constants would provide a more consistent and performant abstraction for providing type-safe access to arrays.

```
fn head<T, const N: usize>(array: &[T; N]) -> Option<&T> {
    if N > 0 {
        Some(&array[0])
    } else {
        None
    }
}
```

Listing 5.8: Type-safe access to the first element of a non-empty array using the `Option` type

In order to check conditionals during monomorphization, is necessary to evaluate the conditionals in the desired constants. For example, if the user calls the `head` function in the `[1, 2, 3]` array, the compiler must decide if `N > 0` is true when `N = 3` is true. Both MIRI and SIRE would be able to perform such task. In particular, SIRE can generate the SIR of `N > 0` and use an SMT solver to check the conditional.

5.3 Generic traits over constants

The RFC-2000 allows the user to write implementations for traits which are generic over constant values, solving the problem of implementing traits for arrays of arbitrary size as shown in Listing 5.9. However, there is no mechanism to extend trait specialization for generics over constant values. As a consequence, if a trait has several implementations for a single type as in Listing 5.10 is necessary to decide which implementation will be used on each case.

```
impl<A: Sized, B, const N: usize> PartialEq<[B; N]>
for [A; N] where A: PartialEq<B> {
    fn eq(&self, other: &[B; N]) -> bool {
        self[..] == other[..]
    }
    fn ne(&self, other: &[B; N]) -> bool {
        self[..] != other[..]
    }
}
```

Listing 5.9: Implementing the `PartialEq` trait for all array sizes

```
impl<const N: usize> PartialEq<[i32; N]> for [i32; N]
with {N + 1 > 1} {
    ...
}

impl<const N: usize> PartialEq<[i32; N]> for [i32; N]
with {N > 0} {
    ...
}
```

Listing 5.10: Two implementations of a trait for the same type

The basic strategy to decide which implementation to use is based on the notion of specificity: If there is more than one implementation of a trait for the same type, the compiler will give priority to the one which is more specific. [2]

---

[2] More information about trait specialization can be found at RFC-1210.

When generics are taken into account, it means that an implementation with boundless generics is less specific that one with bounded generics. At the same time, an implementation with bounded generics is more specific than one without bounds. Finally, when two implementations have bounded generics, the specificity degree will be decided by the specificity of its bounds. One bound is more specific than another if the first implies the second.

```
;; Definition of bound_a provided by sire
(define-fun bound_a
 ((x1 (_ BitVec 64))) Bool
 (bvugt (bvadd x1 (_ bv1 64)) (_ bv1 64)))

;; Definition of bound_b provided by sire
(define-fun bound_b
 ((x1 (_ BitVec 64))) Bool
 (bvugt x1 (_ bv0 64)))

;; Assert that the first bound implies the second
(assert (forall ((x1 (_ BitVec 64)))
 (implies (bound_a x1) (bound_b x1))))

;; Check if the assertions can be satisfied
(check-sat) ; sat
```

Listing 5.11: Checking if `N + 1 > 1` is more specific than `N > 0`.

Naturally, an SMT solver is capable of checking if one predicate implies another, meaning that we can take the SIR of the bounds and decide if one is more specific than another. To decide which implementation is more specific in Listing 5.10, is enough to generate an SMT-LIB snippet like the one found in Listing 5.11.

When the SMT solver is not able to decide which bound is more specific, the same posture as in RFC-1210 can be taken and throw an error telling the user that the implementations are overlapping.

6 Validation

Having discussed the design of our solution, this chapter addresses the advantages and disadvantages of using this new kind of generics on an small linear algebra module. This particular topic was chosen, given that numeric computing makes heavy usage of statically sized linear data structures to represent matrices and vectors.

Using Rust's `array` primitive type would be an unfair comparison, given that in its current state, Rust is not able to abstract the size of the arrays in a satisfactory manner. Instead we use the standard library `Vec`, which is dynamically sized but it can be wrapped to restrict its size.

Listings 6.1 and 6.2 contain two implementations of a `Vector` structure with and without using generics over constant values. Each implementation has three methods:

- The `dot` method: Compute the dot product between the current vector and another vector given as parameter, it is required that both vectors have the same number of dimensions.
- The `append` method: Consume the current vector and return a new vector which has an additional dimension with an element given as paramter.
- The `split` method: Consume the current vector and return two vectors which contain the two halves of the current vector.

The `Vector` type on Listing 6.1 has the size of the vector stored inside the `inner` field and is accessed using `self.inner.len()` as shown in the `size` method. Thus, it is necessary to check if the `other` argument has the same size inside the `dot` method and an `Option<f32>` is returned because there is no certainty that the sizes of the vectors will coincide. The `append` method on Listing 6.1 adds the new element to the `inner` field and returns the same object. Finally, the `split` type uses the method `split_at` of the `Vec` type and creates two new vectors from the two halves of the `inner` field.

The first noticeable difference between the two implementations is the lack of a `size` method in Listing 6.2, this is because the size of the vector is encoded in the `N` parameter of the type `Vector<N>`.

```
pub struct Vector { inner: Vec<f32> }

impl Vector {
    fn size(&self) -> usize {
        self.inner.len()
    }

    pub fn dot(&self, other: &Vector) -> Option<f32> {
        if self.size() == other.size() {
            let mut res = 0.0;
            for i in 0..self.inner.len() {
                res += self.inner[i] * other.inner[i];
            }
            Some(res)
        } else {
            None
        }
    }

    pub fn append(mut self, value: f32) -> Vector {
        self.inner.push(value);
        self
    }

    pub fn split(self) -> (Vector, Vector) {
        let (a, b) = self.inner.split_at(self.size() / 2);
        (Vector { inner: a.to_vec() },
         Vector { inner: b.to_vec() })
    }
}
```

Listing 6.1: A vector implementation without generics over constants

There is also a new method `from_vec_unchecked` which panics during execution if the size of the `vec` argument is not `N`. This is necessary because, there is no way to do this verification during compilation, as a consequence of the dynamic length of the `Vec` type. It is the responsibility of the author of the `Vector<N>` type to use `from_vec_unchecked` in cases when it will not panic. It would be possible to use the `Option` or `Result` type, but this would add an unnecessary overhead in this use case.

```rust
pub struct Vector<const N: usize> { inner: Vec<f32> }

impl<const N: usize> Vector<N> {
    fn from_vec_unchecked(inner: Vec<f32>) -> Vector<N> {
        if inner.len() == N {
            Vector { inner }
        } else {
            panic!("Creating a vector of incorrect size")
        }
    }

    pub fn dot(&self, other: &Vector<N>) -> f32 {
        let mut res = 0.0;
        for i in 0..N {
            res += self.inner[i] * other.inner[i];
        }
        res
    }

    pub fn append(mut self, value: f32) -> Vector<N + 1> {
        self.inner.push(value);
        Vector::from_vec_unchecked(self.inner)
    }

    pub fn split<const M: usize>(self) ->
        (Vector<M>, Vector<N - M>) where M == N / 2 {
            let (a, b) = self.inner.split_at(M);
            (Vector::from_vec_unchecked(a.to_vec()),
             Vector::from_vec_unchecked(b.to_vec()))
        }
}
```

Listing 6.2: A vector implementation using generics over constants

Another difference is the return type in the `dot` method. In Listing 6.1 the method's return type is `Option<f32>`, this is because it is necessary to verify that both vectors have the same size during runtime. In contrast, the Listing 6.2 implementation does not require this verification inside the method because the type compilation. The vector parameter `other` must have size `N` and, as a consequence, calling this method with a vector of a different size would result on a compile time error.

The differences on the `append` method reside in the return type. On Listing 6.2, the size of the return vector is increased by one on its type parameter. However, in order to satisfy this, it is necessary to use the `from_vec_unchecked` method to create a vector of size `N + 1`, this call will not panic given that after adding the new element, the `inner` field will have length `N + 1`.

The `split` method in Listing 6.2 is generic over a constant `M` but is bounded to be equal to `N/2`, then this parameter is used as the size of the return vectors acordingly. Given that the returned vectors have length `M` and `N - M` respectively, using `from_vec_unchecked` will not panic.

This use case provides some valuable insights about the usage of generics over constants. First, being able to use constants as generic type parameters does not guarantee that each parameter coincides semantically with a particular value (in this case, with the size of the vectors) and is the developer's responsibility to keep this semantic match.

As a consequence, developers must be careful when creating variables with generic types over constant from variables without them. Even then, using this new kind of generics allows more expressiveness when constants are involved and simplifies the return type of functions with verifications which can be done during compile time.

7 Conclusion and Future work

7.1 Conclusion

The objective of this thesis was to provide Rust with generics over constants as a means to facilitate the language's capabilities to write generic code over constants.

In this regard, it is certain that adding constant expressions as a kind of parameter over generic types has improved the expressiveness and ergonomy of Rust when dealing with constant abstraction. In particular, these additions make the array types first class entities of the language, allowing to implement traits over all its sizes and providing a safe mechanism to manipulate them with static guarantees. These changes also allow for a more natural definition of user defined types which have information that will not change during execution and thus can be encoded in constant values as generic parameters.

It is expected that following this design would allow for a more performant execution of functions dealing with statically verifiable conditions. However, it is important to remark that this design would increase compilation time given the complexity of verifying if a set of statements is satisfiable.

Some prospects of future work are given in the rest of this chapter.

7.2 Future work

Even though SIRE is able to interpret a subset of the Rust's MIR, it is important to extend the coverage of MIR expressions. In particular, we consider of utmost importance adding support for loops, panicking functions and algebraic data types:

- Loops are the most common way of writing repetitive tasks in Rust even though recursion is supported. Given that loops are represented in MIR as blocks with their terminators looping between them, this would require promoting loops to functions to transform it to a recursive representation suitable for the SMT-LIB format.
- One of the biggest concerns of using any kind of generics is allowing programs with generic types that would produce unexpected behaviour after monomorphization. Checking on which cases a function panics, would allow the compiler to avoid monomorphization of generic types on those cases, or to at least warn the user about them.
- On the other hand, algebraic data types are needed to implement interesting structures such as bounded integers to improve the ergonomy of type safe access of arrays and to allow the usage of constants with types `Option<T>` and `Result<T, E>` as type parameters. To fully support algebraic data types, it is necessary to support MIR projections to represent field access in structures.

Even though using an SMT solver makes our solution simpler in terms of implementation, it adds an additional dependency if these changes were to be added to the compiler. It is also true that an SMT solver is far from the notion of zero-cost abstraction, because it would be more performant to write a logic system to deal only with Rust's characteristics. Thus, we consider necessary to provide an alternative verification mechanism, one option would be to extend the chalk[1] project to deal with these verifications.

As a final prospect, integrating this project with the compiler codebase would simplify maintenance of the new features exposed on this work. This codebase would fit under the constant evaluation modules of the compilers code, given its closeness to MIRI.

---

[1] `https://github.com/rust-lang/chalk`

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004. ISBN 0321227255.

[2] Lennart Augustsson. Cayenne – a language with dependent types. pages 239–250. ACM Press, 1998.

[3] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23, 09 2013. doi: 10.1017/S095679681300018X.

[4] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36946-9. doi: 10.1007/978-3-642-36946-9_3. URL `https://doi.org/10.1007/978-3-642-36946-9_3`.

[5] Stéphane Ducasse, Oscar Nierstras, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, March 2006. ISSN 0164-0925. doi: 10.1145/1119479.1119483. URL `http://doi.acm.org/10.1145/1119479.1119483`.

[6] Richard Eisenberg. Dependent types in haskell: Theory and practice. *CoRR*, abs/1610.07978, 2016. URL `http://arxiv.org/abs/1610.07978`.

[7] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017. ISSN 2475-1421. doi: 10.1145/3158154. URL `http://doi.acm.org/10.1145/3158154`.

[9] Steve Klabnik. The history of rust. New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4464-7. doi: 10.1145/2959689.2960081. URL `http://doi.acm.org/10.1145/2959689.2960081`.

[10] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[11] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. 2005.

[12] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2nd edition, 2017. ISBN 0321714121, 9780321714121.