Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Typesafe Parallel Serialization in SKilL/Rust

Roland Jäger

Course of Study:	Softwareengineering
Examiner:	Prof. Dr. Erhard Plödereder
Supervisor:	Dr. Timm Felden
Commenced:	April 11, 2018
Completed:	October 11, 2018

Abstract

The goal of this thesis is to provide a generator for SKilL that targets Rust. Since Rusts type system enforces strict rules to offer guaranteed thread safety the generated binding has to accommodate these rules, which makes it difficult to create a graph with cycles. A lot of the effort goes into the research and creation of solutions for the missing inheritance between `structs` and the resulting missing casting functionality in the language. In conjunction an option has to be found that allows Rusts rules to be enforced at runtime that works with the enhanced inheritance support. Peculiarly, the resulting implementation is not able to benefit from Rusts rules in regard to safe parallelism, as the runtime enforcement of these rules inhibits parallelism. To validate that the implementation works as expected pre-existing functional tests are used. Additionally performance tests are executed to see how Rust bindings fare against a reference implementation in C++.

Kurzfassung

Das Ziel dieser Arbeit ist es einen Generator für SKilL zu erstellen, welcher Rust Code erzeugt. Da Rust strenge Regeln anwendet, um Threadsicherheit garantieren zu können, muss der generierte Code diesen folgen, was Probleme bei der Darstellung von Graphen mit Zyklen bereitet. Viel Zeit wird für die Findung von Alternativen und Implimentierungen für Vererbung zwischen `structs` und deren fehlenden Konvertierungen aufgewendet. Dazu passend muss auch eine Möglichkeit gefunden werden, die Rusts Regeln zur Laufzeit anwendet und die erweiterte Vererbungsfunktionalität unterstützt. Auffallend ist, dass die Implementierung die Threadsicherheitsgarantien von Rusts Regeln nicht nutzen kann, weil die Einhaltung dieser, zur Laufzeit, die Parallelisierbarkeit einschränkten. Um zu validieren, dass die Implimentierung wie erwartet funktioniert, werden bereits existierende funktionale Tests verwendet. Außerdem wird die Durchsatzrate, des generierten Rust Codes, mit der des generiertem C++ Codes verglichen.

Contents

1 Introduction

This thesis is about the design and implementation of a Serialization Killer Language (SKilL) generator that targets Rust. SKilL is a collection of specifications and tools that enables users to write their applications in their preferred language while offering the ability to serialize the applications state platform and language independently [17]. Language independence is achieved by offering multiple language targets for bindings that are generated from a user provided specification. Rust was chosen as it aims to be a new, safer, systems programming language that features a type system that offers thread safety guarantees [6]. These should allow for safe parallelism in the generated bindings. As reference implementation C++ was used, but C, Java and Scala were available too. This section will present the scope of this thesis and then introduce SKilL and Rust. In the following section the design and implementation will be presented, as well as issues and solutions to these. Finally the results will be evaluated through functional and performance tests.

1.1 Scope

The goal is to provide a generator that supports the core language of SKilL[17]. Further has this generator to support the *documented*, *escaped* and *interfaces* features, a description can be seen in Table 1.1. It has to be researched how the Rust type system can be used to safely parallelize the reading and writing of SKilL binary files. Finally interoperability between existing tools has to be shown through tests. In addition to these mandatory requirements the features *auto* and *custom* are optionally implemented.

Documented	Documentation that the user provides in the specification will be present at appropriate places in the generated code.[17]
Escaped	The user may choose to use illegal characters or names for identifiers for the target language, these have to be escaped.[17]
Interfaces	The specification language allows for interfaces, these can be realized in some languages.[17]
Auto	Auto fields are fields that are generated by the generator but are not to be serialized.[17]
Custom	Custom fields are fields that the user provides the type information for and can be thought of as extension of auto.[17]

Table 1.1: Feature descriptions that the generator implements

1.2 SKilL

SKilL encompasses a specification, binary format and tooling for them, that are used to generate code which enables users to serialise data of their software. For this thesis the term *serialization* is used for both reading data from files and writing data to files. The next sections will give a brief overview of features, tooling and the intended use of SKilL. For more detailed information and rational about design decisions it is advised to refer to *The SKilL Language V1.0*[17] and 'Änderungstolerante Serialisierung großer Datensätze für mehrsprachige Programmanalysen'[14].

1.2.1 Motivation

SKilL was created with the goal to provide small serialised file sizes, fully reflective type encoding, type-safe storage of references, a rich type system, tools that work with partial specifications, coding that is platform and language independent, upwards and downwards compatibility and lazy reading of unused data. The specification language was kept simple by providing fundamental types that are building blocks of most, commonly used, languages as well as single inheritance. These properties allow users to describe the data structures for a software system in a natural way. This specification is then used to generate a binding in a language chosen by the users e.g. C, C++, Java. With the generated binding users are able to write their implementation in a language that they are most comfortable with, by using the offered API. [17]

1.2.2 Architecture

Figure 1.1 illustrates the fundamental architecture of a SKilL binding. The `SkillFile` is the API root that the user accesses to create and open SKilL files. Through `SkillFile` instances the user can access a management object, from here on called `StringPool`, that is responsible for creating and managing `Strings` in such a way that they can be serialised. Likewise there are management objects that are responsible for instances of a user defined type, from here on called `UserTypePools`. These management objects are necessary as the runtime system has to know which objects have to be serialized. For this reason deletion of instances is also handled through these management objects. [14, pp. 107–112]

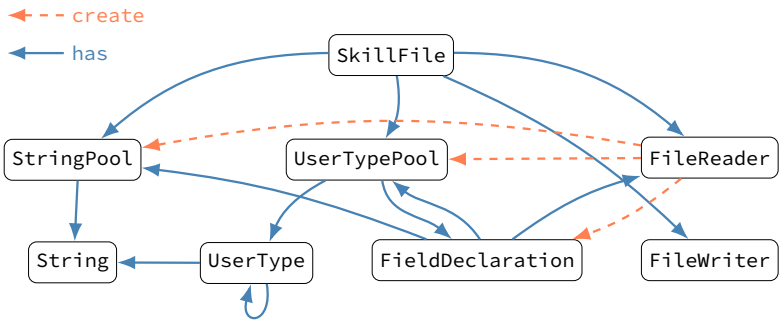


Figure 1.1: Abstract SKilL Binding Architecture

UserTypePools contain FieldDeclarations that describe fields of UserTypes. They can't be encoded in the UserTypePools at generation time, as there can be fields at runtime that have not been known at generation time. Further, fields have to be validated against those in the specification. For serialization FieldDeclarations have to know the UserTypePools as UserType fields can have UserTypes as fields. Similarly, they have to know the StringPool in case a field is of type String. [14, pp. 144–148]

The FileReader is responsible for the creation of the StringPool, UserTypePools and FieldDeclarations. While the creation of UserType instances is handled by UserTypePools, before the field data is read, FieldDeclarations are responsible for reading field data. This split is done as only FieldDeclarations can know how to read the data and if the reading can be deferred until the first access or writing of the instance. Furthermore, this two step process potentially allows for parallelization as all UserType instances are guaranteed to exist when FieldDeclarations require them. [14, 148f.]

1.2.3 Serialization

As seen in Figure 1.2 a SKilL binary file can consist of multiple blocks. Each of these blocks consists of a string section and a type section. String sections first list the amount of strings, followed by the lengths of the strings, followed by the strings themselves. This layout allows for easy reading and writing of the strings with mmap[1] and potentially for parallelization. Type sections start with the amount of types followed by the type definitions. These definitions are about the types only and do not contain information about the fields of the types. In that section the type name, number of instances, restrictions, number of fields and super type are listed. The next section contains the field definitions for the types. Similar to inheritance in most other languages, super field definitions are not redefined for the sub types. The definition consists of the field name, type of the field, as well as the restrictions. A field definition ends with an offset into the next section where the data of the fields resides. [17]

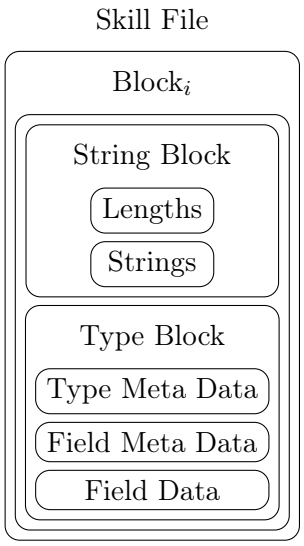


Figure 1.2: SKilL File Layout

All data touched so far is small and fast to read as its size is not influenced by the amount of instances. It is mostly relevant to the SKilL runtime system as this information allows it to serialise data. The last section contains field data of serialised type instances where data of one field is written contiguously for all instances that have that field. The known offsets from the previous sections field declarations and the specific layout of the field data can then be used to skip data with mmap to allow for parallelized serialization. This is possible as there is no overlapping write and read access on the instances memory as all instances are created before serialization starts. [17]

[1] mmap allows to map files and file sections to a memory location that can be accessed like an array.

The reading and writing operations themselves are not too different from the other implementations. It will not be explained in detail. Reading of multi-block files is more sophisticated than this section may make them out to be, but it is not crucial for the rest of the thesis. The involved data structures and data flow however will be discussed as well as their consequences.

1.2.4 Generators

The generators are written in Scala and rely on mixins to be easily extensible, and string interpolation for readability. Mixin classes are a kind of inheritance that allows linearisation of subclasses with different super types instead of an inheritance tree [19, 117f.]. Each generator has a `Main class` that has various `traits` mixed in. These traits are then responsible for creating a certain set of files.

Generators work with an Intermediate Representation (IR) that provides all types that have been defined by the user as well as types for the built-in types. These type representations are given to the generator in form of a type context which can be queried for different types, e.g. user types or interfaces. Once the needed lists of types are obtained a generator can create the files with the translated type definitions. Generators have to pay special attention to the escaping of reserved keywords in their target language. For that a escaping function can be used that is applied to non literal strings.

1.3 Rust

The official Rust language website describes the language as: 'Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.'[6] It presents itself as a *safe* language that should be considered as a substitute for C, C++ and other systems programming languages. The project especially criticizes the following issues in other languages and intends to solve them [4]:

1. There is too little attention paid to safety.
2. They have poor concurrency support.
3. There is a lack of practical affordances.
4. They offer limited control over resources.

But the project doesn't aim to create a completely safe language and instead intends to make compromises in favor of other goals. [4]

The following sections will provide a short introduction to the Rust programming language. The focus is on the elements and features that are important to this thesis. For a more in-depth and up to date introduction it is strongly advised to refer to the official documentation [6, 7, 11, 9].

1.3.1 Borrows and Lifetimes

At the core of the memory and concurrency goal stands the *borrowing* mechanism. Borrowing can be thought of as *taking a reference* in C++. Like all variables borrows are by default *immutable* or in C++ words `const`. Contrary to C++ mutable access has to be explicitly requested with the `mut` keyword. Where borrows differ from references is that they behave like a reader-writer lock regarding the mutability state of the referenced value. That means that a mutable variable may be referenced as immutable more than once at a time or exclusively mutable. This is referred to as *borrow rules*. Additionally Rust attaches a *lifetime* to the borrow that is the lifetime of the referenced memory allocation. A borrow may not outlive its lifetime, meaning that there can't be a reference to freed memory.

```
1 fn main() {
2   let mut s = String::from("Hello");
3   let rs = &s;
4   println!("{}", rs.capacity());
5   println!("{}", s.capacity());
6   rs.reserve(10); // cannot borrow immutable borrowed content `*rs` as
7                   //   mutable
8   s.reserve(10);  // cannot borrow `s` as mutable because it is also
9 }                 //   borrowed as immutable
```

Listing 1.1: Rust Borrow Mechanics

Listing 1.1 depicts a simple example demonstrating these rules. In line 2 a new *owned* string is created – `"Hello"` is a string literal and can't be modified. Since the string shall be modified an explicit conversion function has to be called. An immutable borrow is taken from `s` in line 3 with the `&` operator. The lifetime can be omitted here as the compiler is able to infer it. It is fine to use both `s` and `rs` to call methods that require an immutable borrow of the object. But, using a method that requires a mutable borrow on `rs` will lead to a compile-time error. It might be surprising, but line 8 also leads to a compile-time error. Since `s` is borrowed immutable until the end of the scope it doesn't matter that it is a mutable variable.

```
1 struct Foo {
2   s: &String, // missing lifetime
3 }             //   specifier
4 fn main() {
5   let foo = {
6     let s = String::from("Maria");
7     Foo { s: &s } // No semicolon
8   };              //   returns
9 }
```

(a) Explicit Lifetime Required

```
1 struct Foo<'a> {
2   rs: &'a String,
3 }
4 fn main() {
5   let foo = {
6     let s = String::from("Wolle");
7     Foo { rs: &s } // `s` does not
8   };           // live long enough
9 }
```

(b) Lifetime Mechanism Benefits

Listing 1.2: Rust Lifetime Mechanics

In Listing 1.2a a `struct Foo` is created that has an attribute. `s` is of type `&String`, or in

words: `s` *is a reference to an immutable* `String`. For `s` an explicit lifetime is needed because the compiler is not able to infer where the lifetime will come from. In Listing 1.2b this error is fixed by explicitly giving the borrow a lifetime with `'a`. The lifetime is given to the attribute through the type signature of `Foo`. On construction of `Foo`, line 7, the lifetime of `s` is used as the lifetime for the reference. Because `s` is freed at the end of the scope in line 9 and the `Foo` instance is not, a compilation error is raised. Since the memory of `s` was freed an access through `foo.s` could lead to runtime failures. It has to be noted that lifetimes can only be applied to borrows and are taken implicitly. In other words they can't be used to chain two instances to each other without using borrows [11, PhantomData].

1.3.2 Structs, Traits and Generics

The fundamental pieces of Rusts OOP model are `structs`, `traits` and `impl` blocks. `structs` are equivalent to C `structs` as they are a collection of fields. `traits` can be thought of as Javas `interfaces`. They define a set of functions that have to be implemented by a type that provides a certain `trait`. Besides `traits`, `impl` blocks can be used to add methods to `structs`.

While Rust can be thought of as an OOP language it differs in a few features these languages normally offer. Inheritance is one of the features that differ. It is not possible to inherit from a `struct`, `traits` however can inherit from each other. `traits` can also be implemented for a set of types through type parameters. The set of types that a `trait` is generically implemented for can be restricted through a collection of required `traits` and lifetimes. These requirements are also called *bounds* and can be seen in line 16 of Listing 1.3b. As a result of the missing inheritance code-reuse can be more difficult if a bound can't be expressed. Such a situation can appear as a bound has to hold for all types, whether all types actually invoke the function doesn't matter. This is in contrast to C++ where only the function invocation triggers the instantiation for it [33, §17.8.1 10] which allows more code to be valid. As a solution either composition or code duplication can be used, where the latter can be done through macros for convenience. [7, 10. Generic Types, Traits, and Lifetimes, 17. Object Oriented Programming Features of Rust]

Listing 1.3a declares a `trait` and two `structs` that then implement the `trait` respectively. The functions `who` take a parameter `&self` which de-sugars to `self: &Pinky` and `self: &Becci`. In line 1 of Listing 1.3b the function `fun_gen` uses a type parameter `T` with a bound of `Who`. Code duplication also happens here – this time at invocation of the generic function[2]. `T` will be substituted by the `struct` that implements the bound – in this case `Pinky` and `Becci`.

In case a function is not implemented for a set of types, or it is not possible to do so, the alternative is a *trait object*[7, 17. Object Oriented Programming Features of Rust]. This can be seen in line 19 of Listing 1.3b where the function expects a trait object of type `Who`. Trait objects can be compared to virtual abstract classes in C++. This means that method calls can be *dynamically dispatched* – in other words a virtual function call can be executed for `obj.who()`. Rusts approach to virtual function calls is based on fat pointers

[2] No official documentation validates this, but the assembly code does. With optimisations both functions are in-lined.

```rust
struct Pinky;
struct Becci;
trait Who { fn who(&self); }

impl Who for Pinky {
  fn who(&self) {
    println!("Pinky")
  }
}
impl Who for Becci {
  fn who(&self) {
    println!("Becci")
  }
}
```

(a) Type Structure

```rust
fn fun_gen<T: Who>(obj: &T) {
  obj.who()
}
fn fun_obj(obj: &Who) {
  obj.who()
}

fn main() {
  let p = Pinky;
  let b = Becci;

  fun_gen(&p);
  fun_obj(&b);
}
```

(b) Generics and Trait Usages

Listing 1.3: Rust Generics and Trait Objects

`[data_ptr, vtable_ptr]`[8, Trait Objects, 11, std::raw::TraitObject]. This can introduce additional slowdown as more cache is needed per object, access is non atomic [20, 8.2.3.1 ff.] and moves are more expensive.

1.3.3 Errors, Null and Matching

Rust provides two ways to express an error, both can be seen in Listing 1.4a. The first way uses the `Result<T, E>` wrapper as a return type of functions. This type contains either the success return value of type `T` or an error value of type `E`. In case the value is `0` the success value will be returned. In case the value is greater than `7` an error value will be returned instead. This value can be a `struct` and contain detailed information or none at all. Lastly, there is `panic!`[3] which will terminate the program. A panic in Rust can be compared to an exception, as a panic can be caught [11, std::panic::catch_unwind]. This macro is used throughout the standard library e.g. for an access over the limits of an array. But panics are mostly used for 'unrecoverable' or unexpected problems, whereas `Result` is used in case an error can be expected, e.g. `File::open()` returns `Result<File, Error>`. Another wrapper type that is used throughout the standard library is `Option<T>`, seen in the `gun` function. The main difference to `Result` is, that `Option` can only hold a valid value or none at all. Because of that it is also used as a replacement for `NULL`.

Listing 1.4b shows three ways how these wrapped values can be unwrapped through *matching*. In `hun()` it is the `?` operator, which unwraps the `Result`. If it is the success value it will be returned to the *scope*. But if it is the error value the value will be returned from the *function*. The `?` operator is a tool to direct the data flow of possible error results conveniently. What the `?` operator does can be roughly seen in the `nun()` function. In `pun()` the third way can be seen, where the `Option<T>` value is moved and if it is `Some` it will be matched and accessible through `didi`. Matching can be found in many aspects of

[3] The `!` indicates that `panic!` is a macro.

```rust
fn fun(x: i32) -> Result<DiDi, E> {
  if x == 0 {
    Ok(DiDi)
  } else if x > 7 {
    Err(E)
  } else {
    panic!()
  }
}
fn gun(x: i32) -> Option<DiDi> {
  if x == 42 {
    Some(DiDi)
  } else {
    None
  }
}
```

(a) Rust Error Types

```rust
fn hun() -> Result<(), E> {
  let x = fun(13)?;
  // ...
  Ok(())
}
fn nun() {
  match fun(2) {
    Ok(didi) => didi
    Err(e) => return e
  }
}
fn pun() {
  if let Some(didi) = gun(42) {
    // ...
  }
}
```

(b) Ways to Deal with Errors

Listing 1.4: Rust Error and Matching Mechanics

the language and is mostly a convenience feature as functions can be used to unwrap the values as well. [11, Enums and Pattern Matching]

1.3.4 Cells and Interior Mutability

`UnsafeCell<T>` is a wrapper type from Rusts standard library. This wrapper allows to change contained data without having to be mutable. In cases where an object contains `UnsafeCell<T>` there are two mutabilities, interior and exterior. In other words: an object that is immutable doesn't deny the modification of a value contained in a `UnsafeCell<T>`. The result is, that code that uses `UnsafeCell<T>` is vulnerable to most issues that Rust wants to prevent. It is in the responsibility of the developer to ensure that the borrowing rules are heeded and that thread safety is guaranteed. [11, std::cell]

```rust
use std::cell::Cell;

struct Foo { x: Cell<i8>, y: i8 }

fn main() {
    let f = Foo { x: Cell::new(42), y: 7 };
    f.x.set(13);
    f.y = 13; // cannot mutably borrow field of immutable binding
}
```

Listing 1.5: Example to Interior Mutability

In Listing 1.5 an example to interior mutability can be seen. The `struct Foo` contains a `Cell<T>`, which uses an `UnsafeCell<T>` internally. The provided interface of `Cell<T>` is

safe to use and doesn't need `unsafe`, like `UnsafeCell<T>` does, because there is no way for the user to borrow the contained value. In line 7 it is possible to modify the value of `x`. However, the value of `y` in line 8 results in the compiler raising an error. The mutability of `x` would lead to data races and therefore `Cell<T>` and `RefCell<T>` are not marked as thread safe [11, std::cell, 9, Send and Sync].

1.3.5 Visibility and Grouping

Visibility in Rust is specified with the `pub` keyword. For example, to specify that a field is public to the crate it is prefixed as `pub(crate) foo: String`. Apart from types, fields and functions `pub` is also used for modules. Modules can be compared to Javas packages or C++ namespaces. Files themselves are a module and further modules can be declared in files with the `mod` keyword. This can be used to create a more fine grained grouping and visibility resolution. To flatten the module hierarchy, in case that the file module is not wanted, or to create a more convenient user API the `use` keyword can be used to pull types in another module. [12, Visibility and Privacy, Use declarations]

1.4 Rust Oddities

This section will focus on oddities and questionable design choices that have been encountered while working with Rust. While these are not strictly necessary for the understanding of this thesis, they do provide a deeper understanding of Rust which is helpful to understand decisions made throughout the thesis.

1.4.1 Immutable File Manipulations

A few strange things happen in Listing 1.6. The first thing is, that line 8 and 10 do *not* produce the same output. `file` is immutable, so why and how can an iterator, created by `bytes(self)`, change the position in the file? The how can be explained with interior mutability which was introduced in the previous section. The why however is left unanswered.

Also, how can `bytes(self)` be called by `&file`? `&file` creates a reference to an immutable file but `bytes(self)` requires an object. The deception lies therein, that `trait Read` is not implemented for `File` but for `&'a File`. Which means that `self` is not an object but a reference to one.

But the real question is the following: How can it be that line 12 fails? With the results, that prompt the first question, this should be allowed. In this case there really is no modification going on.

```
use std::io::{Read, Seek, SeekFrom};

fn main() {
  let args = std::env::args().nth(0).unwrap();
  let file = std::fs::File::open(args).unwrap();

  let x = (&file).bytes().next();
  println!("{:?}", x); // Some(Ok(127))
  let x = (&file).bytes().next();
  println!("{:?}", x); // Some(Ok(69))
  // Tell position
  file.seek(SeekFrom::Current(0)); // cannot borrow mutably
}
```

Listing 1.6: File is not Subject to Borrow Rules

1.4.2 Coercion of `traits`

Rust uses *coercion* as the name for implicit casts. Listing 1.7 shows a strange mechanism that is related to coercion. Here, `A` is inherited by `B` and `B` and `A` are implemented for `Foo`. Then `x` is borrowed as `&B` and through `rb` the function `fun()` of `A` is called successfully.[4] But in line 12 the compiler raises an error, informing us that `rb` is not of type `A`. Furthermore, it is not possible to cast `rb` to `A&` with the `as` keyword, as can be seen in line 13.

```
struct Foo {}
trait A { fn fun(&self); }
trait B: A {}

impl B for Foo {}
impl A for Foo { fn fun(&self) {} }

fn main() {
  let x = Foo {};
  let rb: &B = &x;
  rb.fun();
  let ra1: &A = rb; // expected trait `A`, found trait `B`
  let ra2: &A = rb as &A; // an `as` expression can only be used to convert
                          // between primitive types.
  let ra3: &A = &b;
}
```

Listing 1.7: Coercion of Traits

[4] `rb` and `ra` have to be borrows as their size is not known at compile time. [11, E0277 referes to 19.4 Advanced Types]

1.4.3 Immutability Resolution

Rusts immutability is an on/off switch, there is no mechanism to define an attribute as immutable. If an object is passed as immutable nothing can be changed that is not wrapped by an `UnsafeCell<T>`. But on the other hand everything can be changed if the object is mutable. Coming from a C++ background the lack of the `const` feature reduces the expressiveness. To a certain degree it also influences the safety of the program by providing another level of type safety [32, 30ff.]. A workaround is an extra type that wraps the value and only provides immutable accessors.

1.4.4 Static Functions for Traits

Implementing a static function for a `trait` should not be difficult, but it is. In Listing 1.8 two options are shown how to do it.

The first option involves functions that are implemented for `Foo` as well as the two `traits` without a `self` parameter. For the `traits` a bound is required that specifies that the `self` type (`Self`) has to be `Sized`, meaning that the size of `Self` is known at compile time. Because of that bound they can't be called on the `trait` itself, as can be seen in line 28. Valid calls for this approach can be seen in lines 31 to 33.

The other option uses an extra `trait` that is implemented for `Foo` and the two `traits` that should have the static function available. In this example the `trait` is named `IDs` and the expected call syntax works as demonstrated in lines 35 to 37. An alternative for static functions of `traits` are top-level functions that are grouped by name or module. This however is just an option in a context that doesn't use the `trait` in type parameters.

1.4.5 Lifetimes

Lifetimes are needed for the Rust compiler to verify the compliance with the borrow rules, in particular for memory safety. The example from Listing 1.9 is about the lifetimes of borrows. The function `fun()` accepts a `Foo` reference and if `opt` has a value it should be printed and another function should be called. This leads to a compile time error in the current version of Rust. [10, 2094-nll]

1.4.6 Borrowing from `RefCell<T>` Mutable in an Argumentlists

In Listing 1.10 an unfortunate side effect can be seen from using `RefCell<T>` to enforce the borrow rules at runtime. `F` is a `struct` that is wrapped by `RefCell<T>` and has the function `fun()` that needs a mutable `Self`. In line 15 the program will terminate with a panic because the borrow rules were violated. The issue is not solved by introducing a scope around the first borrow and use of `f`, like in `hun()`. Only the introduction of a scope local variable solves the issue, as seen in `nun()`.

```rust
trait Bar {
  fn num() -> usize
    where Self: Sized,
  { 1 }
}
trait Baz {
  fn num() -> usize
    where Self: Sized,
  { 2 }
}
struct Foo;
impl Foo { fn num() -> usize { 3 } }
```

```rust
trait IDs { fn id() -> usize; }
impl IDs for Bar {
  fn id() -> usize { 4 }
}
impl IDs for Baz {
  fn id() -> usize { 5 }
}
impl IDs for Foo {
  fn id() -> usize { 6 }
}
impl Bar for Foo {}
impl Baz for Foo {}
```

```rust
fn main() {
  // println!("{}", Bar::num()); // type annotations required:
                    //   cannot resolve `_: Bar`

  println!("{}", <Foo as Bar>::num());
  println!("{}", <Foo as Baz>::num());
  println!("{}", Foo::num());

  println!("{}", Bar::id());
  println!("{}", Baz::id());
  println!("{}", Foo::id());
}
```

Listing 1.8: Static Functions for Traits

```rust
struct Foo { opt: Option<String>, }
impl Foo {
  fn fun(&mut self) {
    if let Some(ref val) = &self.opt { // immutable borrow occurs here
      println!("{}", val);
      self.gun(); // mutable borrow occurs here
    } // immutable borrow ends here
  }
  fn gun(&mut self) { println!("Norbert"); }
}

fn main() {
  let mut foo = Foo { opt: Some(String::from("Norbi")), };
  foo.fun();
}
```

Listing 1.9: Self in Matches

```rust
use std::cell::RefCell;
use std::collections::HashMap;

struct F {}
impl F {
  fn fun(&mut self) -> i8 { 42 }
}

fn gun(f: &RefCell<F>)
  -> HashMap<i8, i8>
{
  let mut map = HashMap::new();
  map.insert(
    f.borrow_mut().fun(),
    f.borrow_mut().fun() // BOOM
  );
  map
}

fn main() {
  let f = RefCell::new(F {});
  fun(&f);
}
```

```rust
fn hun(f: &RefCell<F>)
  -> HashMap<i8, i8>
{
  let mut map = HashMap::new();
  map.insert(
    { f.borrow_mut().fun() },
    f.borrow_mut().fun() // BOOM
  );
  map
}
fn nun(f: &RefCell<F>)
  -> HashMap<i8, i8>
{
  let mut map = HashMap::new();
  map.insert(
    {
      let mut f = f.borrow_mut();
      f.fun()
    },
    f.borrow_mut().fun()
  );
  map
}
```

Listing 1.10: Borrowing from `RefCell<T>` in argumentlists can lead to panics

2 Design and Implementation

In chapter 1 the basics of SKilL and Rust were explained. This section will feature the design and implementation of the common and generated Rust code. The first sections will present problems that arise from Rusts type system. Afterwards the solution for them will be presented. These sections are the most elaborate ones as the decisions have far reaching consequences and differ the most from the other implementations. Lastly the generated code and the resulting architecture will be covered. The covered parts mostly follow the reference implementations.

2.1 Cyclic Graphs in Rust

SKilL allows the creation of graphs with its references [14, p. 9] as most languages do. This is generally not a problem and sometimes needed to express the required data structure. Rust however prevents the creation of cycles with references, which will be demonstrated in the following sections. In this section different approaches are explored that produce cyclic graphs in Rust. It will roughly follow the recommendation of *Rust - Frequently Asked Questions*[4] and will show the problems that arise in combination with SKilL.

2.1.1 Native Cycles

In Listing 2.1a a naïve implementation of a `Node` containing another `Node` is shown. This looks simple but it is what a cyclic graph boils down to. There has to be an object that doesn't exclusively own memory but references it. Without that detail it would only be possible to create a tree. This example fails as there is no `Node` object before the root `Node x`.

Listing 2.1b uses the `Option` wrapper introduced in subsection 1.3.3. This allows the creation of the initial `Node` object `x`. It is further possible to assign the root object `x` to the child object `y`. But from that point on problems appear. First of all, `y` does not live long enough – this is the direct result of the lifetime specifier. Secondly, `y` has to be assigned to `x.next` and with that `x` needs to be mutable. This should sound familiar to the problems shown in subsection 1.3.1. Since `x` is already borrowed by `y.next`, `x` can't be borrowed mutable again to assign `y` to it.

2.1.2 Pointers in Rust

Coming from a C++ background the next thing that comes to mind would be (raw) pointers. Pointers were not introduced in section 1.3 and the reason for that is, that they are not a

```rust
struct Node<'a> {
  pub next: &'a Node<'a>,
}

fn main() {
  let x = Node {};
  // missing field `next`
  //   in initializer of
  //   `Node<'_>`
}
```

(a) Naïve Implementation

```rust
struct Node<'a> {
  pub next: Option<&'a  Node<'a>>,
}
fn main() {
  let mut x = Node { next: None };
  let mut y = Node { next: Some(&x) };
  x.next = Some(&y);
  // `y` does not live long enough
  // cannot assign to `x.next` because
  //   it is borrowed
}
```

(b) Defered Next

Listing 2.1: Rust, no Cycles Allowed

type commonly used in Rust [8, Raw Pointers]. In Listing 2.2 it can be seen why this is the case.

Listing 2.2a shows the basic usage of pointers, also named 'raw pointers'. Raw pointers differ from references and smart pointers in the following points [7, Unsafe Rust]: Pointers ...

- are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location
- aren't guaranteed to point to valid memory
- are allowed to be null
- don't implement any automatic cleanup

A pointer to the `String` `s` is assigned to `ps` in line 3. In line 6 `s` is accessed through `ps`. To do this `unsafe` code is involved. Unsafe code allows the following [7, Unsafe Rust]:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

```rust
fn main() {
  let s = String::from("Matze");
  let ps = &s as *const String;
  println!(
    "{}",
    unsafe { &*ps }.len()
  );
}
```

(a) Pointers in Rust

```rust
fn main() {
  let mut s = String::from("Denny");
  let ps = &mut s as *mut String;
  let rms1 = unsafe { &mut *ps };
  let rms2 = unsafe { &mut *ps };
  rms1.reserve(42);
  rms2.reserve(42);
}
```

(b) Problems with Pointers in Rust

Listing 2.2: Rust Pointers and Unsafe Code

Why pointers are not the solution for our graph problem and why they are uncommon in Rust code, becomes apparent in Listing 2.2b. A pointer `ps` is created that points to the mutable `String` `s`. Line 4 and 5 indicate that there might be an issue – it is possible to create two *mutable* references to the same value. Line 6 and 7 prove that the safety rules of Rust have been broken, as it is possible to use both `rsm1` and `rsm2` in methods that mutate `s`. With these rules broken also thread safety comes undone and with that most of the incentive to use Rust is lost.

2.1.3 Memory Arenas and Cells

The subsection 1.3.4 introduced `UnsafeCell<T>` and companion types. This section will explore an option to use these to create a cyclic graph. In line 8 of Listing 2.3 a mutable `Vec` is created with two instances of `Node`. In line 12 a new string literal is assigned to the first instance. In line 13 and 14 a reference cycle is created between the two `Foo` instances. This is in contrast to Listing 2.1b, which failed. Apart from `UnsafeCell<T>`, which solves the borrowing error, here a vector is used to store the `Node` instances. This small detail fixes the lifetime issue. In line 15 a compiler error is raised as the second `Node` instance is already borrowed immutable in line 13. That means that fields from the objects can't be modified if they are not also wrapped by a `UnsafeCell<T>`. But that would defeat the purpose of the borrow rules.

```
1  use std::cell::UnsafeCell;
2
3  struct Node<'a> {
4    pub s: &'static str,
5    pub next: UnsafeCell<Vec<&'a Node<'a>>>,
6  }
7  fn main() {
8    let mut arena: Vec<Node> = vec![
9      Node { s: "Nadine", next: UnsafeCell::default() },
10     Node { s: "Andy", next: UnsafeCell::default() },
11   ];
12   arena[0].s = "Hubi";
13   unsafe { &mut *{ &arena[0] }.next.get() }.push(&arena[1]);
14   unsafe { &mut *{ &arena[1] }.next.get() }.push(&arena[0]);
15   arena[1].s = "Quack"; // cannot borrow `arena` as mutable because it is
16                         //   also borrowed as immutable
17 }
```

Listing 2.3: Breaking Mutability Rules

2.1.4 Smart Pointer

The term smart pointer might be familiar from C++. In Rust there is `Rc<T>` which is a reference counting wrapper that acts as pointer and it is comparable to `std::shared_ptr<T>` from C++. This smart pointer uses two integers to count how many *weak* and *strong* references there are to the owned data. If the strong counter reaches zero the destructor

of the contained value is called by `Rc<T>`s destructor. The memory is released once the weak counter reaches zero too. This means there is only one memory allocation which leads to better cache locality and this increases speed. But it also means that the memory can only be freed once the last weak pointer is destructed, which might be unwanted. [11, std::rc::Rc]

The semantics of `Rc<T>` are the following. It is possible to have any number of `Rc<T>`s and `Weak<T>`s to the same `T`. If there is more than one `Rc<T>` or a `Weak<T>` to the same `T` it is only possible to access `T` in an immutable way. Further can `Rc<T>` instances be downgraded to `Weak<T>` and if the strong count has not yet reached zero it is possible to upgrade `Weak<T>` instances to `Rc<T>`. This means that this approach suffers from the same issue that was encountered in subsection 2.1.3. [11, std::rc::Rc]

The solution to this issue is to use an additional wrapper `Rc<RefCell<T>>`. `RefCell<T>` uses internally `UnsafeCell<T>` and provides the means to enforce the borrowing rules at runtime by using an integer counter. When calling `borrow()` or `borrow_mut()` on `RefCell<T>` it will not return a `&T` or `&mut T`, instead `Ref<T>` and `RefMut<T>` are returned. These wrappers will increment the appropriate counters on construction and decrement them on release. If the borrow rules would be violated, by returning a `Ref<T>` or `RefMut<T>`, the afore mentioned functions cause a `panic`. Alternatively there are `try_*` versions of the functions that return `Result<T, E>`. [11, std::cell::RefCell]

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

struct Node {
  pub s: &'static str,
  pub next: Vec<Weak<RefCell<Node>>>,
}
fn main() {
  let x = Rc::new(RefCell::new(Node { s: "Lore", next: Vec::default() }));
  let y = Rc::new(RefCell::new(Node { s:"Steffi", next:Vec::default() }));
  y.borrow_mut().next.push(Rc::downgrade(&x));
  x.borrow_mut().next.push(Rc::downgrade(&y));
  y.borrow_mut().s = "Beni";
}
```

Listing 2.4: Smart Pointer to Cell to Value

Listing 2.4 shows a working example that solves the encountered problems so far. Now `Node` contains a `Vec<T>` that stores `Weak<T>`. It is very important that it is `Weak<T>` and not `Rc<T>` as `Rc<T>` would result in a memory leak. This is the case as `a` and `b` would keep the strong counter above zero for each other.

2.1.5 Indices as Pointers Replacement

Another solution for graphs is something that is similar to Entity Component Systems (ECSs). In an ECS an 'object' is represented through an entity that is an identifier, e.g. an integer. Components contain the data of entities and are stored usually in an indexable,

continuous structure. Systems are responsible to use and modify components, based on logic that uses other components and entities. [27, p. 209] The ECS pattern provides composition over inheritance as the entity has no information about the systems using it and the assigned components. ECSs are most commonly used by game engines as they result in fast code and it is easier to write reusable code with them [27, p. 209, 18, 22].

In the rust community ECS like structures are in discussion as possible solutions to cycles and graphs [21, 23, 25, 30]. An prominent example for an implementation of an ECS is the *specs*[31] library that is used for a GUI[24] and small engines [2, 28].

In 'Reduktion des Speicherverbrauchs generierter SKilL-Zustände'[29] a similar approach to the components of an ECS is explored. Here each field is stored in a continuous container, one container per field for read objects and one for new ones. Through testing of the changed architecture it was found that reading and writing was fast but the access of fields in objects were slower. The reason for that is that the fields are not necessarily close in memory to the proxy objects, leading to cache misses. [29]

While an ECS or the optimisation in 'Reduktion des Speicherverbrauchs generierter SKilL-Zustände'[29] seems favorable to the previous solutions it was ruled out by supervisor Timm Felden. His main argument is that an identifier like an index into a container is worse than a pointer. If a container is resized, all indices have to be updated. That, however, is only possible if all identifiers, that are using the indices, are known and accessible by the system, which can not be guaranteed. In case that an proxy object would be used, as in 'Reduktion des Speicherverbrauchs generierter SKilL-Zustände'[29], it would prompt a similar problem to the one discussed in this section. All proxy objects would need to be able to access the container that store their field data. These containers would have to be modified in case that an object is added and the data in the containers would also have to be modified through the proxy objects. That results in a conflict with Rusts borrow rules.

2.2 Inheritance in Rust

Rusts inheritance differs from common OOP languages. In Rust `structs` are just a collection of data and can not inherit other `structs` or make use of code reuse through inheritance. `traits` on the other hand can inherit multiple `traits`, as can be seen in Listing 2.6. `trait B` *requires* that `trait A` is implemented for whatever type `B` is implemented for. That makes it also possible to call methods from `A` on instances of type `B`. However it is not possible to assign a variable of type `&B` to a variable that expects a type of `&A`. The need for a reference (`&`) comes from the fact that `B` is just a `trait` and can't be instantiated. A data pointer to a `struct` instance is required to create a trait object of type `B`, seen in line 11.

```
1 class A {
2     int stefi;
3 }
4 class B extends A {
5     boolean dani;
6 }
```

Listing 2.5: Inheritance in Java

SKilL uses single inheritance [17] as present in the Java example seen in Listing 2.5. In this section techniques are explored that can be used to mimic single inheritance in Rust.

```rust
trait A { fn fun(&self); }
trait B: A { fn gun(&self); }

struct Foo;

impl B for Foo { fn gun(&self) {} }
impl A for Foo { fn fun(&self) {} }
```

```rust
fn main() {
  let x = Foo;
  let rb: &B = &x;
  rb.fun();
  let ra: &A = rb; // expected
      // trait `A` found trait `B`
}
```

Listing 2.6: Trait Inheritance

2.2.1 Inheritance through Composition

Instead of inheritance Rust advocates composition. From that perspective the code from Listing 2.7 would represent the type hierarchy present in Listing 2.5. This has a few nice properties. The user would be working with types and not trait objects and thus the code would not use virtual functions. Further can up casts be performed by returning a borrow to the contained super type, but on the other hand, down casting becomes impossible as the super component has no knowledge about the child component. Since the memory layout of `structs` is undefined no assumption about a pointer offset can be made [10, 0079-undefined-struct-layout]. As alternative an option from section 2.1 could be used e.g. `Option<Rc<RefCell<T>>>` to create inheritance cycles. That would however degrade the usability and require linear time or memory to perform the casts.

```rust
struct A {
  stefi: String,
}
struct B {
  sup: A,
  dani: u32,
}
```

Listing 2.7: Inheritance emulated with Composition

2.2.2 Inheritance through Traits

If a type hierarchy like Listing 2.5 has to be realized in Rust another option is to use `traits`, which would result in something akin to Listing 2.8. The 'real' types have the suffix `Proper` and the access `traits` have the original type name. This naming scheme improves the usability for the user of the code as the user will most of the time have to use trait objects. While inheritance can be expressed this way it has a few quirks that have to be improved upon.

- Down casting is impossible as there is no cast function or keyword in the language to go from a `trait` to a `struct`, or to a `trait`.
- Up casting only works for the 'proper' type and not the access `trait` that emulates inheritance – review Listing 1.7.

To solve these two issues a `cast()` function has to be created that is able to create a trait object from another. In subsection 1.3.2 trait objects were introduced as fat pointer and this property can be abused to implement the `cast()` function. The first field of a trait object is the data pointer to the `struct` and copying the pointer is no issue. However the second field is more difficult to fill as the correct vtable pointer has to be created.

```rust
struct AProper {
  stefi: String,
}
struct BProper {
  stefi: String,
  dani: u32,
}
```

```rust
trait A {
  fn get_stefi(&self) -> &String;
  fn get_stefi_mut(&mut self) -> &mut String;
  fn set_stefi(&mut self, stefi: String);
}
trait B: A {
  fn get_dani(&self) -> u32;
  fn set_stefi(&mut self, stefi: u32);
}
```

Listing 2.8: Emulated Inheritance with Traits

Fortunately, Rust allows the creation of a trait object from a `null` pointer which can be seen in Listing 2.9 [9, 2.2. Exotically Sized Types]. [1]

```rust
struct Proper;
trait Object {}
impl Object for Proper {}

#[repr(C)]
struct TraitObject {
  pub data: *const (),
  pub vtable: *const (),
}
fn main() {
  let vtable = unsafe {
    std::mem::transmute::<_, TraitObject>(
      std::ptr::null::<Proper>() as *const Object
    ).vtable
  };
}
```

Listing 2.9: Obtaining a VTable from `null`

Listing 2.9 shows how to obtain the vtable pointer. There is a `Proper struct` and a `trait Object` that is implemented for `Proper`. Apart from Rusts memory layout, where the compiler may reorder and add padding to fields, Rust also supports C-style memory layout with the `#[repr(C)]` annotation. This is needed for the `TraitObject struct` which allows casting trait objects to it and thereby accessing the two pointers. `()` is the *unit type* comparable to `void`[9, repr(C), 7, Defining and Instantiating Structs]. In line 13 a `null` pointer is created with the type of `Proper` and then casted to a trait object of type `Object`. This trait object is then `transmuted` to a `TraitObject` instance. `_` is replaced by the compiler with the type of the parameter. Through the interface of `TraitObject` the vtable pointer can be accessed easily.

Once the vtable pointer has been obtained the two pointers can be used to create a `TraitObject` object. This object can be transmuted to a normal trait object as if it was created by the `struct` through an implicit cast. The last detail for the `cast()` function would be a way to determine which type to get the vtable from. This can be done by either

a lookup table realized through an array or a function with conditional branches. Both can be generated through the code generator.

The outstanding issue with this approach is that it has to be applied to the values themself. Any normal wrapper would inhibit the ability to perform these casts.

2.2.3 C-Style Memory Layout

Similarly to the previous section is the C-memory layout used to allow for casts not otherwise possible, like it is shown in Listing 2.10. The field definitions have to appear in the same order, to create the same memory layout, for all types in a type hierarchy [9, repr(C)]. This approach has the issue that functions still wouldn't be inherited and would have to be implemented by the user, repeatedly, for all valid `structs`. Similarly to the `traits` there is an issue with casting if the memory of the `struct` is managed by a wrapper type. A nice property about this approach is that it fits better in normal Rust code and implicit casts to `traits` can still happen as the proper type is used.

```
1  trait T { fn fun(&self) -> usize; }
2
3  #[repr(C)]
4  struct Foo { pub x: usize, }
5  #[repr(C)]
6  struct Bar { pub x: usize, pub y: usize, }
7
8  impl T for Foo { fn fun(&self) -> usize { self.x } }
9  impl T for Bar { fn fun(&self) -> usize { self.y } }
10
11 fn main() {
12   let mut bar = Bar { x: 42, y: 7 };
13   let foo = unsafe { std::mem::transmute::<_, *mut Foo>(&bar) };
14   assert_eq!(unsafe { &*foo }.x, 42);
15   assert_eq!(unsafe { &*foo }.fun(), 42);
16   unsafe { &mut *foo }.x = 13;
17   assert_eq!(bar.x, 13);
18   assert_eq!(bar.fun(), 7);
19 }
```

Listing 2.10: Inheritance with C-Memory layout and Structs

2.3 Support Cyclic Graphs and Struct Inheritance

This section will focus on the implementation of the discussed solutions of section 2.1 and section 2.2. The wrapper is named `Ptr<T>` since it provides the functionality of working with the type `T` almost as if it was being pointed to by a pointer without restrictions. More concrete means this that this wrapper enables `T` to be passed around with reference counting and runtime borrow rules checking while being able to be cast to different types and trait objects.

2.3.1 Memory Layout

The majority of the `Ptr<T>` type consists of `Rc<T>` and `RefCell<T>`, combined into one cohesive wrapper. That means the original source code was copied and combined. This was needed as this type is supposed to be able to be castable, meaning that the contained value should be able to represent a `struct` or trait object. Without this manual combination the reference counting and mutability state enforcement would inhibit proper casting.

```rust
struct MetaData {
  cast_id: usize,
  strong: Cell<usize>,
  weak: Cell<usize>,
  borrow: Cell<BorrowFlag>,
}

struct Ptr<T: ?Sized> {
  meta: NonNull<MetaData>,
  value: NonNull<T>,
}
```

(a) Memory Layout from `Ptr<T>`

```rust
struct RcBox<T: ?Sized> {
  strong: Cell<usize>,
  weak: Cell<usize>,
  value: T,
}
struct Rc<T: ?Sized> {
  ptr: NonNull<RcBox<T>>,
  phantom: PhantomData<T>,
}
struct RefCell<T: ?Sized> {
  borrow: Cell<BorrowFlag>,
  value: UnsafeCell<T>,
}
```

(b) Memory Layout from `Rc<T>` and `RefCell<T>`

Listing 2.11: Memory Layout Comparison

Merging of the to types results in the memory layout seen in Listing 2.11a. Compared to Listing 2.11b this uses one additional pointer. `Rc<T>` only needs one pointer for the `RcBox<T>` that contains the meta data as well as the value. The `PhantomData<T>` is there to provide the compiler with information which 'is used when computing certain safety properties.'[11, std::marker::PhantomData]. `RefCell<T>` doesn't need any pointers whatsoever. Since there can be many `Ptr<T>`s that point to the same data but have a different views[1] the value can not be shared between the `Ptr<T>` instances. The type parameter `T` is bound by `?Sized`, which means that `T` can be a type of known size, like a `struct` and `T` can also be of unknown size, like a dynamic array `[usize]` where `usize` is the unsigned integer as element type [11, std::marker::Sized].

2.3.2 Casting with `Ptr<T>`

The implementation of casting for `Ptr<T>` uses the trait object casting as well as the C-style memory layout casting that was introduced in section 2.2. That means that for every `struct` of type `T` there is an accessor `trait` that inherits all super accessor `traits`. This combination provides the user with high flexibility of trait objects while keeping the non virtual function calls in most cases. This is the case as the user can work with `structs` instead of trait objects in most places. It also means that the user can use the accessor `traits` as bound to implement the logic for all types without manual code duplication.

[1] The data pointer is not so much the problem, but the vtable pointers from trait objects are.

Listing 2.12 shows the commented code to cast any `Ptr<T>` to `Ptr<U>`, where `T` and `U` can each be a `struct` type or a `trait` type. The `CastAble` trait bound is needed to call `cast_id()` on `U`. With both `id`s the lookup table can be accessed and depending on the value of `Option<T>` (`Some<T>` or `None`) the cast will be performed or not. The lookup table is implemented as a $N \times M$ matrix, where the elements are an `Option<VTable>`, which means that the `struct` and accessor `trait` are handled equally. What is interesting about this function and Rusts generics is, that the code does the same thing for every `T` and every `U`. That means that even if a `struct` is cast to another `struct` a trait object is created and the vtable will be inserted into it as if `U` was the accessor trait. Fortunately the vtable is at index 1 and data at index 0 because `transmute` will cut the additional pointer off in case `U` is a `struct` type.

2.3.3 Consequences of `Ptr<T>`

`Ptr<T>` is slower than a raw pointer. This slowdown can be attributed to the enforcement of the borrow rules and the memory fragmentation that is created through the small allocations.

Using `Ptr<T>` means also that the code using `Ptr<T>` and its companion types can't be parallelized. The issue is that `Ptr<T>` is reference counting to enforce borrow rules as well as the lifetime of the managed memory. That means that both, borrowing and cloning has to happen in a synchronized way to achieve object parallelism. But, the parallelism benefits can only be gained if the fields of the objects can be serialized in parallel. The binary format is designed in such a way that the field data can be read in parallel. With a little extra work parallel writing is also possible. To support that kind of parallelism at a field level, all fields in the objects would have to be enclosed in something like an `UnsafeCell<T>` for interior mutability.

`Ptr<T>` would have to be made from a `Arc<T>` and `RwLock<T>`, instead of `Rc<T>` and `RefCell<T>`, to allow for object parallelism. Which means that every object access would be behind a mutex and multiple atomic operations [11, std::sync::Arc, std::sync::RwLock]. To achieve field parallelism fields have to be wrapped in `UnsafeCell<T>` for interior mutability, since threads are involved that means that either `Mutex<T>` or `RwLock<T>`[2] have to be used. Leading to possible two mutex locks per field access when serialisation of the value and also for all accesses the users does. Apart from the objective to use Rusts *type system* for the thread safety guarantees it is further highly unlikely that the mutex locks do not ruin the performance.

Another consequence of `Ptr<T>` is, that the specification independent code can not be contained in a standalone create which can be used as library. `Ptr<T>` depends on the lookup table which has to be generated.

[2] `Mutex<T>` and `RwLock<T>` are both using `UnsafeCell<T>` internally. [11, std::sync::Mutex, std::sync::RwLock]

2.3.4 Optimisations

Since `Ptr<T>` is used in many components some time was spent to see how it could be made faster. The next sections will present optimisations that were thought of.

Lookup vs Conditionals

The initial version of `Ptr<T>` used conditionals to check whether the cast is valid to make or not. It would then create the vtable pointer as introduced in subsection 2.2.2. The conditional logic was contained in a `trait` that was implemented by a macro that only needed the valid type names. That made `Ptr<T>` reusable in other projects not related to SKilL.

In an attempt to improve the performance of `cast()` the final implementation uses a lookup table. The lookup table contains the vtable pointers and absence of a pointer indicates an invalid cast. A performance improvement could however not be measured. An explanation for that is, that the trait objects are still created and that branch-prediction was in this case roughly as efficient as an access into a 2D array.

Disabling Reference Counting and Borrow Checking

A more brutal optimisation was an additional type to `Ptr<T>`. It was named `HazardPtr<T>` as there was nothing safe about that type. `HazardPtr<T>` is constructed from a `Ptr<T>` and because it was declared in the same module as `Ptr<T>` it is able to access the private fields of `Ptr<T>`. In the constructor `HazardPtr<T>` would then take the pointer to the value of `Ptr<T>`. This allowed to skip the reference counting and the borrow rules checking. Additionally were the bounds on `HazardPtr<T>` changed to include only `structs`. With that it was possible to create a `cast()` function that only casts a pointer to another pointer – no support for trait object casts is necessary in the internal code. Further were these casts unchecked. `HazardPtr<T>` was then applied to internal code that used `Ptr<T>` to serialise field data. In performance tests the use of `HazardPtr<T>` resulted in a speed up of $\times 1.2$ compared to `Ptr<T>`.

`HazardPtr<T>` was not kept however, as its unsafe nature goes strongly against the values that Rust represents. This experiment showed that the wrapper types used throughout the code base (`Rc<T>`, `RefCell<T>`) do inflict a significant performance penalty. `Ptr<T>` instances are used by pools that contain a shared vector of `Ptr<T>` instances (`Rc<RefCell<Vec<Ptr<T>>>>`[3]) which means that another speedup could be gained if these wrappers were unnecessary.

[3] `Rc<RefCell<Vec<Ptr<T>>>>` ≈ `Rc<RefCell<Vec<Rc<RefCell<T>>>>>`

2.4 Binding Architecture

This section will discuss the created architecture of generated bindings and the changes made to the one introduced in subsection 1.2.2. While the generated code is a central point of this thesis, the generator itself is not so much. The reason for that is, that the generator differs in a few points from the reference generators of other languages. This is the case as the same infrastructure was used and no modifications had to be made to support Rust.

Figure 2.1 shows the actual architecture of the generated code. Additionally to the relationships seen in the previous architecture this diagram features *traits*, *usage*, if they were generated or not and what kind of type they are. This means that more types and relationships are featured, making this diagram more complicated.

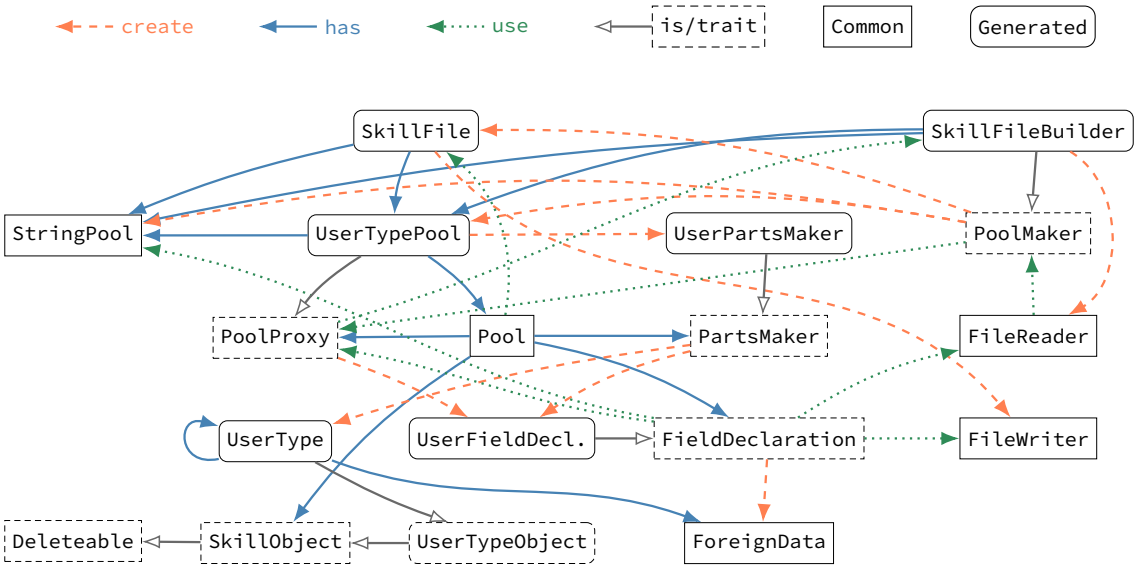


Figure 2.1: Binding Architecture

Not depicted in the image are:

- `ForeignPool`, `ForeignObject` and `ForeignFieldDeclaration` as they have the same relationships as the `UserType` equivalent.
- Convenience types like strongly typed integer or magic numbers.
- `StringBlock` as it is composed into `StringPool` and only `FileReader` and `FileWriter` use it internally.
- Iterators that are used to iterate over instances of type hierarchies and pools.
- `SkillString`, `SkillFail`, `Ptr<T>` and `Ptr<T>` companion types like `Weak<T>`, as they are used, had, and created by most components.

2.4.1 Rust Integration

Cargo uses a manifest file named `Cargo.toml`. In this file the binary or library is described. This includes dependencies, build settings, versions, tests configurations and benchmarks configurations. The file is of utmost importance for the user of the generated code as this will allow the easy inclusion in the target project. [5, The Manifest Format]

Accompanying there are `mod.rs` and `lib.rs` or `main.rs` files. These files regulate the visibility of the different modules – where each file and folder is a module. `mod.rs` files are used in directories to specify the visibility of the contained files as well as to flatten the module hierarchy. `lib.rs` or `main.rs` files are at the root of the `src` directory and mark the begin of a library or executable. These files allow additionally to specify external crates, enable features and mute warnings. Further can additional modules be declared in the files to allow for fine granular visibility. [7, 7. Modules]

The `Cargo.toml` and `lib.rs` files are quite easy to generate. For `Cargo.toml` only the library name has to be substituted. And for the `lib.rs` file the flattening of the module structure has to be performed. This is done to allow users to access `Foos` pool with `use user_project::FooPool` instead of `use user_project::foo::FooPool`.

2.4.2 SKilL File

`SkillFile` is the root of the API for the user. `SkillFile` provides functions to create and open SKilL files. An instance of `SkillFile` is also the place where the user is able to interact with `UserTypePools` and `StringPool`. This type is straight forward to generate, as it is mostly a composite type that provides access to its components. Components are `UserTypePools` and `StringPool`. Only these components have to be inserted in otherwise static code.

However, `SkillFile` is constructed by a `SkillFileBuilder`. This separation is necessary since all attributes have to be initialized on construction. That is not possible without a wrapper like `Option<T>`.

Additionally there is the `PoolMaker trait`. This `trait` is used by the `FileReader` and is only needed for a better separation of specification independent, and generated code. This separation is kept to allow for a common standalone crate of the specification independent code if a casting method is found that allows that.

2.4.3 User Definitions

For the user defined types a few types have to be generated that work closely together to provide the user with a convenient API. Since there is no benefit towards compile time, by splitting code into multiple files[4], each of these types reside in one file per user type definition.

[4] For example benefit C and C++ from forward declarations of types instead of including the header files.

User Type

Type definitions are translated to `structs` and `traits`. The `structs` contain fields that were specified for its super types as well as the fields that were specified for itself. The accessor `traits` for the `structs` contain, for every field that is not a super field, an accessor method and inherits its super accessor `traits` and interfaces. These methods are different depending on the kind of accessed type. Types that are not trivial to copy (e.g. `HashMap`) are passed by reference. Since a borrow occurs in that case there have to be three methods. Two getters, one that returns a reference to an immutable value and another that returns a reference to a mutable value. The third method is the setter that accepts a value that replaces the original one. Copyable types are accessed through functions that use the plain value. All super accessor `traits` and interfaces, as well as `SkillObject`, `ForeignObject` and `Deleteable` are then implemented for the `struct`.

Listing 2.13 shows generated code of the `UserType` named `Age`. The fields in `Age`, that are prefixed with `z_`, are internal fields. Fields that the user named with a leading `z` and non alphabetic or numeric characters are escaped with an additional `z`. The `skill_id` is used as id for the object at the serialization phase or for the user to access a particular object. The `type_id` is the index into the vtable lookup table which is used to cast the objects. `foreign_data` is a vector that stores data of fields that were not known at generation time. As previously mentioned is this data inaccessible for the user. Finally is `age` the field that the user specified in the specification.

`age` needs accessors to be accessible outside of this module which are declared in the `AgeObject trait`. `AgeObject` also inherits `SkillObject` and therefore has to implement it and also the `Deleteable trait`. The `Deleteable` trait provides an interface that allows the objects to be marked for deletion. This interface should not be visible to the user but there is no way to do that in Rust.

Additionally to the types present in the users specification `Age` also implements `ForeignObject` that belongs to the `Foreign struct`. The `Foreign` type is exclusively used to create an interface that allows the serialization of types and fields that were not known at generation time.

Type Pools

The biggest change compared to Figure 1.1 appears around the `UserTypePool`. This type had to be split up into five. The `Pool` type is needed for code reuse as Rust doesn't provide inheritance. Most of the code that is needed for `UserTypePools` is specification independent and can be reused without issue. The problem with code reuse through composition is that it is difficult to provide the general, contained type with the containing `struct`'s data or special functions. In case that the containing `struct` calls the functions of `Pool` data and special functions can be made available. But if the contained `struct` is called by something else a dependency cycle occurs that has to be broken with options from section 2.1, which can lead to violations of borrow rules. The `Pool` is used by other objects than the `UserTypePools` that do not know of the `UserTypePools`. Because of that `UserPartsMaker` exists which externalizes the special functions of `UserTypePool` so that

they are callable by Pool. Since UserTypePools are not a Pool and because they have to be able to be stored in a container the PoolProxy trait was created. PoolProxy is an interface for UserTypePools that provides access to the Pool component in UserTypePools as well as the ability to store UserTypePools of different UserTypes in containers.

Further are PoolProxy and PartsMaker responsible to create UserFieldDeclaration instances. PartsMaker is used by Pool to create UserFieldDeclaration instances that are to be read. PoolProxy on the other hand is needed to create UserFieldDeclaration instances that are missing from the serialized data but were present in the specification.

Field Declarations

Lastly there are FieldDeclarations, which are responsible for the serialisation of fields from UserTypes. For each field a FieldDeclaration is generated that implements how the field data is read and written. They also handle the setup of the different chunks that are involved if the file that is read has multiple blocks.

2.4.4 VTable-Lookup-Table

The lookup table is implemented as a $N \times M$ matrix, where M is the amount of types $+1$ to accommodate the foreign type. $N = M + 1$ to additionally accommodate the shared trait SkillObject. The memory footprint could be reduced by using a sparse array at the trade-off for lookup speed. The matrix uses Option<VTable> instances as elements – if the value is None the cast is invalid, otherwise the correct vtable is obtained. The generator produces the correct vtables with the technique introduced in subsection 2.2.2. For the lookup to be efficient[5] all user types as well as Foreign and SkillObject have to be assigned an index into the matrix. This is done by the generator and the reason to why there is not a common crate that contains specification independent code.

2.4.5 Specification Independent Code

Most of the architecture is separate from the specification and doesn't have to be generated by a generator. This includes the reading and writing of the build in types, the parsing and writing of the files, the string pool and the traits that need to be implemented. As the lookup table has to be generated which means that Ptr<T> is dependent on generated code and in turn most of the specification independent code as well. This section focuses on some of more interesting parts of these types.

[5] HashMap was testes as replacement for an array but was found to be inefficient.

Strings and their Pool

Strings have to be managed and known by the `StringPool`. That means that a `String` may not be modified by the user directly. The reason for that is, that type and field names are managed strings too. A direct modification from the user might inadvertently change the representation of the type system.

To conform to these requirements and the need for an id for each string – that is obtainable through the string – a wrapper is needed for `String`. Since inheritance is not an option `String` was made a component of `SkillString`. `SkillString` has to provide a method to obtain an immutable reference to the contained `String` to be somewhat compatible with other libraries. Since the removal of a `SkillString` that represents a literal used by a type would lead to problems there is no way for the user to delete a `SkillString` explicitly. The explicit deletion is replaced with automatic pruning on writing. Since all `SkillString` have to be wrapped in an `Rc<T>` the counters can be used to check whether the `SkillString` has to be serialized or can be discarded.

Foreign Types and Fields

Special types are used if the field or type encountered upon deserialisation was not known at the time the code was generated. The fields themself are represented through a Rust `enum` which can be thought of as a tagged `union`. The foreign field definitions are special in that they deserialise their data only if it has to be written. This is possible as the user has no way to interact with the foreign types or fields in the Rust implementation[6].

If a type has a known ancestor in a type hierarchy that type will be used instead to represent the foreign one. That way the user's code can work with the known fields without knowing anything about the foreign type. In turn that means that every user type has to have an array of `ForeignFieldData`. In case that there is no known ancestor type the `Foreign` type is used to represent the type. This type is not accessible by the user and only contains `ForeignFieldData`.

Error and Handling

Return codes are the predominant way in Rust to report issue, which was introduced in subsection 1.3.3. The issue with return codes is that they are unsafe, as they can be ignored, and that they do not provide a lot of context by themself for debugging purposes. To improve the usability of these return codes the `failure` crate [3] has been chosen to enhance the error codes. The crate provides macros that extend the codes with the option to provide a backtrace. Additionally it allows for error messages that are created and formatted at runtime and are able to provide more context than a string literal.

The error codes are implemented through 3 `enums`. `SkillFail` is the main `enum` that the user interacts with. It can either contain a `UserFail enum` value or a `InternalFail enum`

[6] In *The SKilL Language*[16] the API to access foreign types and fields was a feature, in *The SKilL Language V1.0*[17] it was changed to be a core requirement. For this thesis supervisor Timm Felden said to treat it as feature.

value. This distinction was made to further enhance the context of the error while keeping the amount of errors that the user has to react to at a minimum.

```rust
fn cast<U>(&self) -> Option<Ptr<U>>
where
  // Bound of U;
  //   U can be Sized, or Unsized
  //   U may only have references that live as long as the whole program
  //   U has to implement CastAble which is needed to call cast_id()
  U: ?Sized + 'static + CastAble,
{
  // Obtain the lookup table index of Self
  let own_id = unsafe { // Use of unsafe as a pointer is accessed
    self.meta.as_ref() // Obtain a reference to the MetaData struct
  }.cast_id; // Access the value through the reference
  // Access the lookup table with the index of Self and U
  let res = VALID_CASTS[own_id][U::cast_id()];
  // If the cast is valid unwrap res
  if let Some(vtable) = res { // vtable is the vtable of U's trait
    unsafe { // Unsafe because of pointers and transmute
      let mut t = TraitObject { // Construct a new TraitObject
        // value is either a pointer to a Sized type e.g. Struct or of a
        //   trait object. In both cases points the pointer to data, a
        //   Sized type
        data: self.value.as_ptr() as *const (),
        vtable, // Set the vtable of U
      };
      // Create a new wrapped pointer
      let value = NonNull::new_unchecked(
        // Reinterpret the memory pointed to by ts reference;
        //   If U is Unsized, a Trait, then the type of the reference will
        //     be reinterpreted to a real trait object of type U
        //   If U is a Sized type, e.g. Struct the reference will be
        //     reinterpreted as reference to U
        //     The vtable pointer will be 'cut off' in this case
        *::std::mem::transmute::<
          _, // Type of the argument (&mut *mut T)
          &mut *mut U // A reference to a mutable pointer that points to a
                      //   mutable U
        >(&mut t), // Take a reference to mutable t
      );
      let meta = self.meta.as_ref(); // Convenience variable
      // Increase the strong counter for the reference counting
      meta.strong.set(meta.strong.get() + 1); // Cell assessor methods
      Some( // Return Ptr<U> wrapped in Some to indicate success
        // Construct a new Ptr<U> with the newly casted value and shared
        //   meta data
        Ptr { meta: self.meta, value }
      )
    }
  } else {
    None // Return None to indicate a invalid cast
  }
}
```

Listing 2.12: Cast method of `Ptr<T>`

```rust
#[derive(Default, Debug)]
#[repr(C)]
pub struct Age {
  z_skill_id: Cell<usize>,
  z_skill_type_id: usize,
  z_foreign_data: Vec<foreign::FieldData>,
  age: i64,
}
pub trait AgeObject: SkillObject {
  fn get_age(&self) -> i64;
  fn set_age(&mut self, age: i64);
}
impl Age {
  pub fn new(skill_id: usize, skill_type_id: usize) -> Age {
    Age {
      z_skill_id: Cell::new(skill_id),
      z_skill_type_id: skill_type_id,
      z_foreign_data: Vec::default(),
      age: 0,
    }
  }
}
impl AgeObject for Age {
  fn get_age(&self) -> i64 { self.age }
  fn set_age(&mut self, value: i64) { self.age = value; }
}
impl foreign::ForeignObject for Age {
  fn foreign_fields(&self) -> &Vec<foreign::FieldData> {
    &self.z_foreign_data
  }
  fn foreign_fields_mut(&mut self) -> &mut Vec<foreign::FieldData> {
    &mut self.z_foreign_data
  }
}
impl SkillObject for Age {
  fn skill_type_id(&self) -> usize { self.z_skill_type_id }
  fn get_skill_id(&self) -> usize { self.z_skill_id.get() }
  fn set_skill_id(&self, skill_id: usize) -> Result<(), SkillFail> {
    if skill_id == skill_object::DELETE {
      return Err(SkillFail::user(UserFail::ReservedID { id: skill_id }));
    }
    self.z_skill_id.set(skill_id);
    Ok(())
  }
}
impl Deletable for Age {
  fn mark_for_deletion(&mut self) {
      self.z_skill_id.set(skill_object::DELETE);
  }
  fn to_delete(&self) -> bool {self.z_skill_id.get()==skill_object::DELETE}
}
```

Listing 2.13: Generated User Type Age

3 Evaluation

The following sections will focus on reasons for the missing parallelism, performance impacts from the Rust type system and testing of implemented features for the SKilL binding. This is predominantly done through the provided test suites as well as a custom benchmark.

3.1 Parallelism

A feature that wasn't implemented was the parallelisation of the serialization. In subsection 2.3.3 the consequences of `Ptr<T>` towards parallelism were discussed and concluded that the Rust type system can't be used for graph like structures that should be used safely in parallel.

Rusts guarantees for data race free programs stems from the borrowing rules. These are pushed to the runtime since the type system is unable to accommodate graph like structures while upholding the borrow rules. Wrapper types have to be used that enforce the rules at runtime. That means that the research point, how the *type system* can be used to implement the serialization in a safe parallel manner, is a negative result. While it is not impossible to implement the serialisation in a 'parallelized' way the implications make it quite clear that it would be unwise to do so. Since two mutex locks have to be made per field access, there is only a slow down to be obtained, which would be forced on the users code, too.

3.2 Tests

There are two test suites present in the SKilL repository [13] that use 28 specifications. These test suites are used for the existing bindings generators and were used to develop in a test driven manner.

The first test suite is used to test reading capabilities of bindings. To do that specifications are used to generate Rust bindings. Then a test file is generated that features a test method for every test data file. For example test data files test the built-in types or inheritance. The generated binding as well as the test file are then compiled and tested by `cargo test`. All 644 tests from this test suite, that do not require restrictions to fail, are passed successfully.

The second test suite tests the API and with that requires more thorough adjustments for each generator. This test suite does not use SKilL binary files as input. Instead it uses a specification file and a JSON file that describes objects to create a graph with objects from the specification. This means that similarly to the first test suite a binding is generated.

But instead of requiring only the reading from a given input file, here the tests, in the test file, have to use the API, of the binding, to create objects and use these to set and get values. For Rusts test suite the tests are generated in a way that the API is used to create a `SkillFile`, create graphs based on the test suites JSON input, write it to disc, read it back from disc and compare the read data to the JSON input. Similarly to the first test suite, all 189 tests pass successfully that do not require restrictions to fail.

Furthermore 10 additional tests were created. Some of these tests use multiple specifications to test the interoperability of Rust bindings and the `Foreign` types. Another is used to test for memory leaks that can be created through the reference counting of `Ptr<T>`. Lastly there is a test that tests the *custom* feature. These manually created tests are executed as part of the second test suite as they depend on the generated bindings.

To validate that the created SKilL binary files are interoperable with other existing tools and bindings `skillView`[15] was used throughout the development process. The tool allows to explore SKilL binary files. Since the tool didn't show any issues with the files and because the first test suites input data comes from other generators, interoperability is demonstrated.

3.3 Performance

In Figure 3.1 a performance comparison of Rust, C++ and Java is shown. Figure 3.1a offers an more detailed view for files that are smaller than 20Mb while Figure 3.1b features all data points. Figure 3.1c shows a logarithmic plot of the data and features a guide line of $y = 10x$. It can be seen that Rust has a steeper inclination than the guide line albeit much shallower than the other languages. Furthermore can be seen that Java matches the other transfer rates better if the files are larger as the start from the program and JVM is more noticeable in smaller files.

The displayed data was generated by a script that generated the bindings from the specification of 'SKilLed LLVM'[26]. It would then generate a main function that would read, write and again read a file given as an argument, 100 repetitions each. After compiling all binaries with the default optimisation levels, the harness would time the execution of the binaries while supplying a selection of data files from 'SKilLed LLVM'[26] that is listed in Table 3.2. Because of memory leaks in the C++ version this setup had to be chosen, instead of a more library focused one, where the binary would serialize the file multiple times in the same process. On the other hand, the C++ version has more features implemented than the Rust version. Because of these facts this comparison should be taken with caution but the large divergence of run times indicates that other more fundamental pieces differ. The test system information is listed in Table 3.1. The system was not interacted with and all user applications were closed while testing.

A `perf` analysis didn't show any particular hot spot that could be optimized. The code seems to be generally slow which should be expected if every `Ptr<T>` instance access requires at least two integer additions for the borrow checking and at least four if it is obtained from a `UserTypePool`. Since the memory locations, that are pointed to by `Ptr<T>`, are not one contiguous piece but fragmented memory locations it is further possible that the bigger

Component	Information
CPU	i7-4702HQ
Memory	16GB
Storage	SSD SM841
Arch Linux	4.18.11
GCC	8.2.1
Clang	7.0.0
Java	Oracle 11
Rust	1.30.0-nightly
Optimization	3
SKilL	`59e9c1f9`
`cppCommon`	`dd6021f`

Table 3.1: Test System Information

File (release)	Size	Nodes
tty.sf	107K	5.371
make.sf	1.1M	49.649
screen.sf	3.2M	139.779
bash.sf	6.1M	256.383
gnuplot.sf	8.1M	345.310
git.sf	16M	601.669
vim.sf	19M	781.822
llvm-split.sf	52M	2.043.963
lli.sf	151M	5.357.733
clang.sf	256M	9.044.428
opt.sf	322M	10.856.418

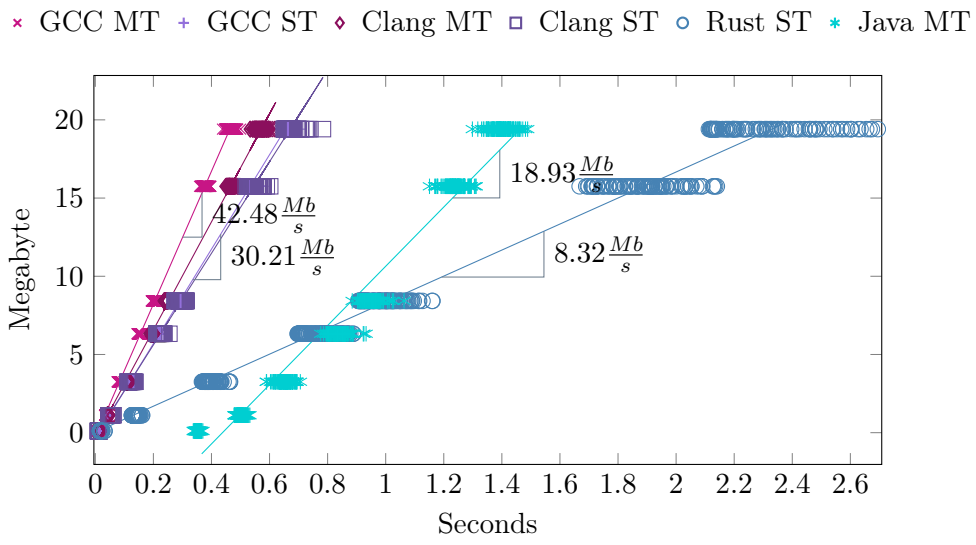
Table 3.2: Selected Performance Input Test Data from 'SKilLed LLVM'[26]

spread of runtime per file is caused by memory fragmentation. It can also be assumed that even with a 'perfectly' parallelized implementation the speedup of the parallel processing would not be enough to catch up to C++. While the GCC C++ single thread version is slower than the multi thread version (×1.4) the rust version is ×3.3 slower than the single threaded GCC C++ version (×4.6 slower than the multi thread C++ version). LLVM is not responsible for the longer run-times since the run-times of the C++ versions, compiled with clang, almost match the ones of GCC.

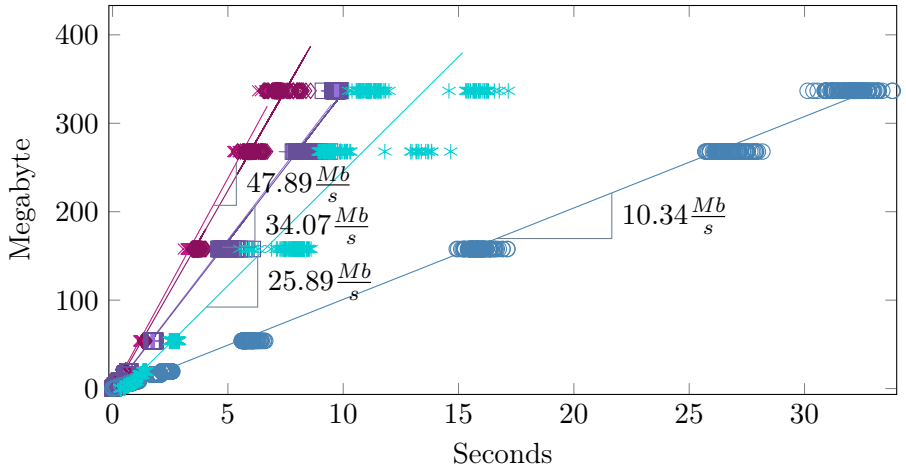
3.4 SKilL Improvements

The one thing that could be improved in a new specification would be that annotation fields do not have to use two byes to signal that the annotation is `NULL`. Currently the pool id and object id is given and set to 0 but since there can't be a pool with id 0 the additional byte for the object id can be saved.

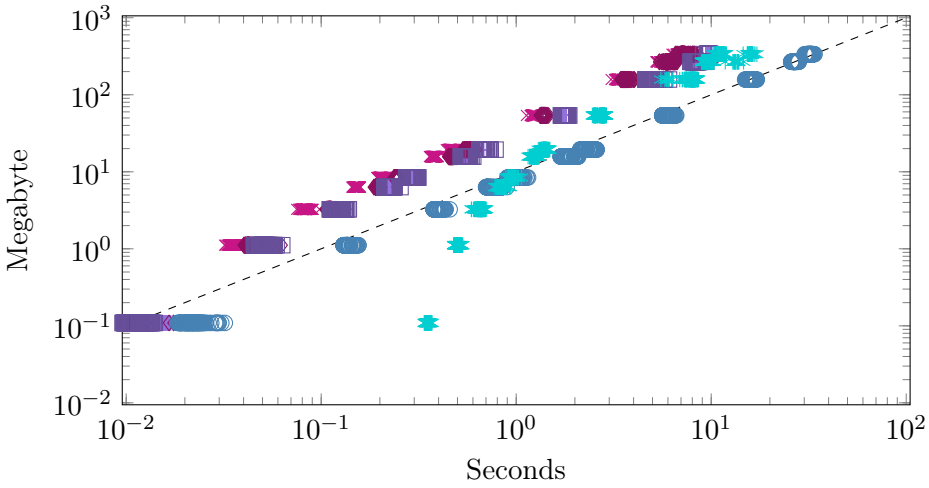
Another improvement was made to Rusts and C++ implementation of size calculation, reading and writing for variable length integers, used by `v64` and pointers. Listing 3.1 and Listing 3.2 show reference implementations, found in *The SKilL Language V1.0*[17]. The C++ implementation already featured the unrolled loop and defered size calculation, but it still used the same bit masks and shifts and needed two integers for decoding. Listing 3.3, Listing 3.4 and Listing 3.5 show the improved versions of size calculation, encoding and decoding. The previous decoding implementation needed one `OR`, one `SHIFT`, one `COND`, two `AND` and two `ASSIG` operations per byte. The improved version needs one `AND` operation and one `ASSIG` operation less per byte and made an integer obsolete. The size calculation can be done with just one comparison per byte, removing a additional `AND` operation. Likewise the additional `AND` operation was removed from the encoding function.



(a) Files < 20Mb



(b) All test files



(c) Logarithmic graph of all test files, with guide line $y = 10x$.

Figure 3.1: Performance comparison between Rust, C++ and Java

```
uint8_t* encode(uint64_t value) {
  // calculate effective size
  int size = 0;
  {
    uint64_t buckets = value;
    while (buckets) {
      buckets >>= 7;
      size++;
    }
  }
  if (!size) {
    uint8_t[] result = new uint8_t[1];
    result[0] = 0;
    return result;
  } else if (10 == size)
    size = 9;
  // split
  uint8_t[] result = new uint8_t[size];
  int count = 0;
  for (; count < 8 && count < size - 1; count++) {
    result[count] = value >> (7 * count);
    result[count] |= 0x80;
  }
  result[count] = value >> (7 * count);
  return result;
}
```

Listing 3.1: Reference encode implementation for v64[17]

```
uint64_t decode(uint8_t* v64) {
  int count = 0;
  uint64_t result = 0;
  register uint64_t bucket;
  for (; count < 8 && (*v64) & 0x80; count++, v64++) {
    bucket = v64[0];
    result |= (bucket & 0x7f) << (7 * count);
  }
  bucket = v64[0];
  result |= (8 == count ? bucket : (bucket & 0x7f)) << (7 * count);
  return result;
}
```

Listing 3.2: Reference decode implementation for v64[17]

```
static uint64_t size(uint64_t v) {
  return (v < 0x80U) ? 1 : ((v < 0x4000U) ? 2
      : ((v < 0x200000U) ? 3 : ((v < 0x10000000U) ? 4
      : ((v < 0x800000000U) ? 5 : ((v < 0x40000000000U) ? 6
      : ((v < 0x2000000000000U) ? 7 : ((v < 0x1000000000000000U) ? 8
      : 9)))))));
```

Listing 3.3: Optimized size implementation for `v64`

```
inline void encode(uint64_t v) {
  if (v < 0x80U) {
    *(position++) = (uint8_t)(v);
  } else {
    *(position++) = (uint8_t)(0x80U | v);
    if (v < 0x4000U) {
      *(position++) = (uint8_t)(v >> 7U);
    } else {
      *(position++) = (uint8_t)(0x80U | v >> 7U);
      // ...
                if (v < 0x1000000000000000U) {
                  *(position++) = (uint8_t)(v >> 49U);
                } else {
                  *(position++) = (uint8_t)(0x80U | v >> 49U);
                  *(position++) = (uint8_t)(v >> 56U);
                }
      // ...
    }
  }
}
```

Listing 3.4: Optimized encode implementation for `v64`

```
inline int64_t decode() {
  uint64_t v;

  if ((v = *(position++) >= 0x80U)) {
    v = (v & 0x80U - 1) | (*(position++) << 7U);
    if (v >= 0x4000U) {
      v = (v & 0x4000U - 1) | (*(position++) << 14U);
      // ...
                if (v >= 0x1000000000000000U) {
                  v = (v & 0x1000000000000000U - 1) | (*(position++)<<56U);
                }
      // ...
    }
  }
  return v;
}
```

Listing 3.5: Optimized decode implementation for `v64`

4 Summary and Future Work

This thesis features the implementation of a generator that produces Rust bindings for SKilL. The binding was supposed to implement the code language of SKilL as well as the *documented*, *escaped* and *interfaces* features. Further the binding has to show that it is compatible with current bindings and tools. A central point was to research how the Rust type system can be used to execute the serialisation process in a parallelized manner safely.

The generator that has been implemented for Rust produces bindings that satisfy the core language of SKilL, with the exception of the public reflection API and enforced restrictions. Both deficits were acknowledged by supervisor Timm Felden. Furthermore, the *auto* and *custom* features of SKilL were implemented.

The correctness of the generated bindings was tested through existing test suits, new tests for the test suits as well as manual testing with existing tools. The later was done to further prove that interoperability exists between Rust and other bindings.

Parallelism, supported by the type system, was not achieved however. The central issue was that SKilL allows for graph like structures which conflict with Rusts borrow rules. These rules are essential for thread safety as they act like a reader-writer lock on references that exists as long as the reference. To create cycles wrappers are used that enforce the borrow rules at runtime. Since these wrappers have a state the state has to be accessed in a synchronized fashion if these wrappers are shared across threads. Also object parallelism is not enough, field parallelism is required to benefit from parallelism. This means that fields of the objects have to be wrapped by a synchronizing wrapper too. The resulting architecture was deemed too inefficient in comparison to a sequential solution.

Apart from the issue with parallelism, inheritance and casting were interesting subjects. Rust features inheritance for `traits` which are comparable to Javas `interfaces` but not for `structs` which can be compared to C `structs`. Furthermore, only casts from a `struct` to a `trait` are possible, not however from a `trait` to a `struct` or from a `trait` to a `trait`. To allow all casts a method was created that requires the `structs` to have a C memory layout. The method used a lookup table to check if a cast is valid or not. To execute casts to a `trait` a vtable pointer is necessary, which is also contained in the lookup table.

The performance of Rust bindings is $\times 4.6$ slower than the parallelized C++ version. This is due to the missing parallelism and the enforcement of borrow rules at run time. Since most time was spent searching for solutions for the previously mentioned issues and an initial Rust implementation there are opportunities to optimise Rust bindings. For a parallelized version the architecture and wrapper types would have to be changed drastically. One option for such an architecture would be to use pointers throughout the internal code which would mean that that code would suffer from the same issues as C and C++.

Bibliography

[1] Diggory Blake. *Dynamically query a type-erased object for any trait implementation*. 2018. URL: https://github.com/Diggsey/query_interface (visited on 07/10/2018).

[2] Amethyst Community. *Data-driven game engine written in Rust*. 2018. URL: https://www.amethyst.rs/ (visited on 28/08/2018).

[3] Rust Community. *failure - a new error management story*. 2018. URL: https://github.com/rust-lang-nursery/failure (visited on 10/09/2018).

[4] Rust Community. *Rust - Frequently Asked Questions*. 2018. URL: https://www.rust-lang.org/en-US/faq.html (visited on 28/08/2018).

[5] Rust Community. *Rust - The Cargo Book*. 2018. URL: https://doc.rust-lang.org/cargo (visited on 28/08/2018).

[6] Rust Community. *The Rust Programming Language*. 2018. URL: https://rust-lang.org (visited on 18/08/2018).

[7] Rust Community. *The Rust Programming Language - Book 2018*. 2018. URL: https://doc.rust-lang.org/book/2018-edition (visited on 27/08/2018).

[8] Rust Community. *The Rust Programming Language - Book First Edition*. 2018. URL: https://doc.rust-lang.org/book/first-edition (visited on 27/08/2018).

[9] Rust Community. *The Rust Programming Language - Nomicon*. 2018. URL: https://doc.rust-lang.org/nomicon (visited on 26/09/2018).

[10] Rust Community. *The Rust Programming Language - Requests for Comments*. 2018. URL: https://github.com/rust-lang/rfcs (visited on 27/08/2018).

[11] Rust Community. *The Rust Programming Language - Standard Library Documentation*. 2018. URL: https://doc.rust-lang.org/std (visited on 27/08/2018).

[12] Rust Community. *The Rust Programming Language - The Rust Reference*. 2018. URL: https://doc.rust-lang.org/reference (visited on 27/08/2018).

[13] Timm Felden. *A reflection-based viewer and editor for binary skill files*. 2018. URL: https://github.com/skill-lang/skill (visited on 04/10/2018).

[14] Timm Felden. 'Änderungstolerante Serialisierung großer Datensätze für mehrsprachige Programmanalysen'. de. PhD thesis. 2018. DOI: 10.18419/opus-9661.

[15] Timm Felden. *Cross platform, cross language, easy-to-use serialization interface generator*. 2018. URL: https://github.com/skill-lang/skillView (visited on 02/10/2018).

[16] Timm Felden. *The SKilL Language*. Englisch. Technischer Bericht Informatik 2013/06. Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Sept. 2013, p. 43.

[17] Timm Felden. *The SKilL Language V1.0*. Deutsch. Technischer Bericht Informatik 2017/01. Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Jan. 2017, p. 64.

[18] Franco Eusébio Garcia and Vânia Paula de Almeida Neris. 'A data-driven entity-component approach to develop universally accessible games'. In: *International Conference on Universal Access in Human-Computer Interaction*. Springer. 2014, pp. 537–548.

[19] PhD (auth.) Iain D. Craig MA. *Object-Oriented Programming Languages: Interpretation*. 2017.

[20] Intel Intel. 'and IA-32 architectures software developer's manual'. In: *Volume 3A: System Programming Guide, Part* 1.64 (), p. 64.

[21] Ismail Khoffi. *Graphs and arena allocation*. 2018. URL: https://github.com/nrc/r4cppp/blob/master/graphs/README.md (visited on 28/08/2018).

[22] Patrick Lange, Rene Weller and Gabriel Zachmann. 'Wait-free hash maps in the entity-component-system pattern for realtime interactive systems'. In: *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2016 IEEE 9th Workshop on*. IEEE. 2016, pp. 1–8.

[23] Rust Leipzig. *Idiomatic tree and graph like structures in Rust*. 2018. URL: https://rust-leipzig.github.io/architecture/2016/12/20/idiomatic-trees-in-rust/ (visited on 28/08/2018).

[24] Raph Levien. *Entity-Component-System architecture for UI in Rust*. 2018. URL: https://raphlinus.github.io/personal/2018/05/08/ecs-ui.html (visited on 28/08/2018).

[25] Niko Matsakis. *Modeling graphs in Rust using vector indices*. 2018. URL: http://smallcultfollowing.com/babysteps/blog/2015/04/06/modeling-graphs-in-rust-using-vector-indices/ (visited on 28/08/2018).

[26] Daniel Pfister. 'SKilLed LLVM'. MA thesis. Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau, 2018.

[27] Raimondas Pupius. *SFML Game Development By Example*. Packt Publishing Ltd, 2015.

[28] Simon Rönnberg. *Physics library for use in Specs*. 2018. URL: https://docs.rs/rhusics/0.2.0/rhusics/ (visited on 28/08/2018).

[29] Jonathan Roth. 'Reduktion des Speicherverbrauchs generierter SKilL-Zustände'. Deutsch. Masterarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, May 2015, p. 82.

[30] Simon Sapin. *Borrow cycles in Rust: arenas v.s. drop-checking*. 2018. URL: https://exyr.org/2018/rust-arenas-vs-dropck/ (visited on 28/08/2018).

[31] Thomas Schaller. *The Specs Book*. 2018. URL: https://slide-rs.github.io/specs (visited on 28/08/2018).

[32] Herb Sutter and Andrei Alexandrescu. *C++ coding standards*. Pearson India, 2004.

[33] *Working Draft, Standard for Programming Language C++*. Standard n4713. International Organization for Standardization, 27th Nov. 2017.

List of Figures

List of Tables

List of Listings

List of Acronyms

API Application Programming Interface. 8, 15, 33, 36, 41, 42, 49

ECS Entity Component System. 24, 25

GUI Graphical User Interface. 25

IR Intermediate Representation. 10

JSON JavaScript Object Notation. 41, 42

JVM Java Virtual Machine. 42

OOP Object Oriented Programing. 12, 25

SKilL Serialization Killer Language. 7–9, 21, 25, 31, 33, 41–43, 49, 54

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, the 10th October 2018 _____

Roland Jäger

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, den 10. Oktober 2018 _____

Roland Jäger