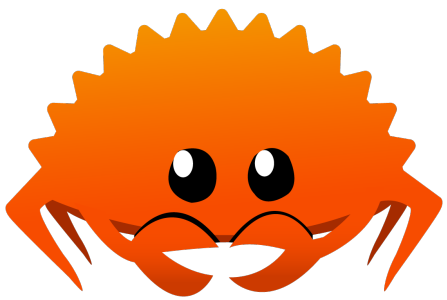
Fluence Compute Engine (FCE), interior mutability, and TryInto

Lukas Prokop

December 17, 2020

RustGraz community



Fluence Compute Engine

FCE

What is FCE?

> *Fluence Compute Engine (FCE) is a general purpose Wasm runtime that could be used in different scenarios, especially in programs based on the ECS pattern or plugin architecture. It runs multi-module WebAssembly applications with interface-types and shared-nothing linking scheme.*

Presentation by michaelvoronov:

> *Mike Voronov has more than 10 years of experience in C++ and 2+ years experience in Rust and WebAssembly*

Interior mutability

Reference semantics

Revision: shared versus mutable references.

```
fn overwrite(base: &u32, new_value: &u32) {
  base = new_value;
}
fn main() {
  let value = 3;
  overwrite(&value, &42);
  println!("{}", value);
}
```

Does it compile?

Reference semantics

Revision: shared versus mutable references.

```
fn overwrite(base: &u32, new_value: &u32) {
  base = new_value;
}
fn main() {
  let value = 3;
  overwrite(&value, &42);
  println!("{}", value);
}
```

Does it compile? No, *lifetime mismatch*.

Reference semantics

```
fn overwrite<'a>(base: &'a u32,
                 new_value: &'a u32) {
  base = new_value;
}
fn main() {
  let value = 3;
  overwrite(&value, &42);
  println!("{}", value);
}
```

Does it compile?

Reference semantics

```
fn overwrite<'a>(base: &'a u32,
                 new_value: &'a u32) {
  base = new_value;
}
fn main() {
  let value = 3;
  overwrite(&value, &42);
  println!("{}", value);
}
```

Does it compile? No, *cannot assign to immutable argument 'base'* .

Reference semantics

```
fn overwrite<'a>(mut base: &'a u32,
                 new_value: &'a u32) {
  base = new_value;
}
fn main() {
  let value = 3;
  overwrite(&value, &42);
  println!("{}", value);
}
```

Does it compile?

Reference semantics

```
fn overwrite<'a>(mut base: &'a u32,
                 new_value: &'a u32) {
  base = new_value;
}
fn main() {
  let value = 3;
  overwrite(&value, &42);
  println!("{}", value);
}
```

Does it compile? Yes, but prints 3, not 42.

Reference semantics

Modifying the reference …

```
fn overwrite<'a>(mut base: &'a u32,
                 new_value: &'a u32);
```

versus

```
fn overwrite<'a>(base: &'a mut u32,
                 new_value: &'a u32);
```

… modifying the value behind a reference.

Reference semantics

```
fn overwrite<'a>(base: &'a mut u32,
                 new_value: &'a u32) {
  *base = *new_value;
}
fn main() {
  let mut value = 3;
  overwrite(&mut value, &mut 42);
  println!("{}", value); // prints 42
}
```

Excursion: constant variables

Small excursion: Benedikt also pointed out the following snippet:

```
fn overwrite<'a>(base: &'a mut u32,
                 new_value: &'a u32) {
  *base = *new_value;
}
fn main() {
  let mut value = 3;
  overwrite(&mut 42, &mut 42);
  println!("{}", value); // prints 3
}
```

Apparently, rust creates *two* local variables with value *42*. This is interesting since constants of same value need not be allocated twice usually.

Mutability semantics

1. A variable is mutable; or not.
2. A reference is mutable; or not.
3. If a struct instance bound to a variable is mutable; its members are mutable too (*inherited mutability*).
4. Mutability checks happen at *compile time*.

```
struct User {
  id: u32,
  posts_count: u32,
}
let mut meisterluk = User {
  id: 1,
  posts_count: 42
};
```

What about a data structure that modifying elements in the background without knowledge of the user?

Interior mutability types in rust

Multithreaded? Use atomic datatypes or locks (Mutex, RwLock, …)!
Otherwise, …

`Cell<T>` Provides interior mutability for some value.

`RefCell<T>` Provides interior mutability for some value and returns references in its API.

`UnsafeCell<T>` Underlying primitive for the types above

… implemented with *runtime* checks!

Cell

`Cell<T>`: This type wraps an existing value and provides interior mutability.

```
use std::cell::Cell;
struct User {
  id: u32,
  count_posts: Cell<u32>,
}

fn main() {
  let meisterluk = User {
    id: 1,
    count_posts: Cell::new(42),
  };
  // meisterluk.id = 4;  // error: meisterluk is not mutable
  meisterluk.count_posts.replace(55);
  println!("User=({}, {})",
    meisterluk.id,
    meisterluk.count_posts.get()
  );
}
```

Cell API

`Cell<T>`: This type wraps an existing value and provides interior mutability.

1. `pub const fn new(value: T) -> Cell<T>`
2. `pub fn set(&self, val: T)`
3. `pub fn swap(&self, other: &Cell<T>)`
4. `pub fn replace(&self, val: T) -> T`
5. `pub fn into_inner(self) -> T`
6. `pub const fn new(value: T) -> Cell<T>`

RefCell

`RefCell<T>`: This type wraps an existing value and provides references for interior mutability.

```
use std::cell::RefCell;

struct User {
  id: u32,
  count_posts: RefCell<u32>,
}

fn main() {
  let meisterluk = User {
    id: 1,
    count_posts: RefCell::new(42),
  };
  meisterluk.count_posts.replace(55);
  println!("User=({}, {})",
           meisterluk.id,
           meisterluk.count_posts.borrow());
}
```

RefCell API

1. `pub const fn new(value: T) -> RefCell<T>`
2. `pub fn borrow(&self) -> Ref<'_, T>`
3. `pub fn try_borrow(&self) -> Result<Ref<'_, T>, BorrowError>`
4. `pub fn borrow_mut(&self) -> RefMut<'_, T>`
5. `pub fn try_borrow_mut(&self) -> Result<RefMut<'_, T>, BorrowMutError>`
6. `pub fn as_ptr(&self) -> *mut T`
7. `pub fn get_mut(&mut self) -> &mut T`
8. `pub unsafe fn try_borrow_unguarded(&self) -> Result<&T, BorrowError>`

RefCell

```
use std::cell::RefCell;

struct User {
  id: u32,
  count_posts: RefCell<u32>,
}

fn main() {
  let meisterluk = User {
    id: 1,
    count_posts: RefCell::new(42),
  };
  let a = meisterluk.count_posts.borrow_mut();
  let _ = meisterluk.count_posts.borrow_mut();
}
```

Runtime error!

```
thread 'main' panicked at 'already borrowed: BorrowMutError',
      src/main.rs:14:36
note: run with `RUST_BACKTRACE=1` environment variable
      to display a backtrace
```

TryInto / Into / TryFrom / From

Motivation

1. Sometimes it is convenient to convert one type into another (*coercion*, *casting*)
2. Usually done explicitly in rust with `as` keyword
3. But when calling a function, we often know the source and target target. How can we convert it?

4 traits

```
pub trait Into<T>: Sized {
  fn into(self) -> T;
}
pub trait TryInto<T>: Sized {
  type Error;
  fn try_into(self)
    -> Result<T, Self::Error>;
}
pub trait From<T>: Sized {
  fn from(_: T) -> Self;
}
pub trait TryFrom<T>: Sized {
  type Error;
  fn try_from(value: T)
    -> Result<Self, Self::Error>;
}
```

4 traits

1. Try traits permit conversion to fail.
2. All traits are reflexive (T can be converted *into* T).
3. Prefer to implement `TryFrom` instead of `TryInto`
4. Implementing `From` automatically provides one with an implementation of `Into`.

Example implementation

Example via Rust by Example:

```
use std::convert::{TryFrom, TryInto};

#[derive(Debug, PartialEq)]
struct EvenNumber(i32);

impl TryFrom<i32> for EvenNumber {
  type Error = ();
  fn try_from(value: i32)
    -> Result<Self, Self::Error>
  {
    if value % 2 == 0 {
        Ok(EvenNumber(value))
    } else {
        Err(())
} } }
```

Example implementation

```
fn main() {
  // TryFrom
  assert_eq!(EvenNumber::try_from(8),
             Ok(EvenNumber(8)));
  assert_eq!(EvenNumber::try_from(5),
             Err(()));

  // TryInto
  let result: Result<EvenNumber, ()> = 8i32.try_into();
  assert_eq!(result, Ok(EvenNumber(8)));
  let result: Result<EvenNumber, ()> = 5i32.try_into();
  assert_eq!(result, Err(()));
}
```

Strings to vectors

Strings can be converted into a vector of bytes.

```
fn main() {
    let a: Vec<u8> = String::from("hello").into();
    println!("{}", a[0]);
}
```

Into as trait bound

Trait bounds can specify that any type convertible into another will be accepted.

```
fn is_hello<T: Into<Vec<u8>>>(s: T) {
  let bytes = b"hello".to_vec();
  assert_eq!(bytes, s.into());
}

let s = "hello".to_string();
is_hello(s);
```

Epilogue

Quiz

Who maintains the WASM standard?

What is mutable about a mutable reference?

How can you implement casting for your own type?

When do you implement Into instead of TryInto?

Quiz

Who maintains the WASM standard?

WebAssembly became a World Wide Web Consortium recommendation in Dec 2019

What is mutable about a mutable reference?

How can you implement casting for your own type?

When do you implement Into instead of TryInto?

Quiz

Who maintains the WASM standard?

WebAssembly became a World Wide Web Consortium recommendation in Dec 2019

What is mutable about a mutable reference?

The value the reference is pointing to

How can you implement casting for your own type?

When do you implement Into instead of TryInto?

Quiz

Who maintains the WASM standard?

WebAssembly became a World Wide Web Consortium recommendation in Dec 2019

What is mutable about a mutable reference?

The value the reference is pointing to

How can you implement casting for your own type?

Implement Into/TryInto or From/TryFrom

When do you implement Into instead of TryInto?

Quiz

Who maintains the WASM standard?

WebAssembly became a World Wide Web Consortium recommendation in Dec 2019

What is mutable about a mutable reference?

The value the reference is pointing to

How can you implement casting for your own type?

Implement Into/TryInto or From/TryFrom

When do you implement Into instead of TryInto?

If the conversion cannot fail under any circumstances.

Next time

Next meetup	Wed, 2021/01/27
Topic	Cross-compilation: compiling for the raspberry PI

Thank you!