DSTs

Florian “Florob” Zeitz

2019-04-03

Last time's discussion

- Can you resize the object behind a `Box<[u8]>`
- Different from assigning a different `Box<[u8]>` to the same binding
- Maybe `std::mem::replace(&mut *the_box, [1, 2, 3, 4][..])`?

When Size Doesn't Matter

- Most types have a size known at compile time
- We can store them on the stack, pass them to functiions, etc.
- Sometimes we want to pass something we don't know the size of
  - array of arbitrary size
  - part of an array/vector
  - implementer of a Trait (without generics)

Regular Structs


```
1  struct Slice<'a, T> {
2      ptr: *const T,
3      len: usize,
4      _phantom: PhantomData<&'a T>,
5  }
```

- Can point to an array of any length, or slice thereof
- Done, right?

Regular Structs


```
1  struct Slice<'a, T> {
2      ptr: *const T,
3      len: usize,
4      _phantom: PhantomData<&'a T>,
5  }
```

- Can point to an array of any length, or slice thereof
- Done, right?
- What about mutably borrowing from this type?

Regular Structs


```
1  struct Slice<'a, T> {
2      ptr: *const T,
3      len: usize,
4      _phantom: PhantomData<&'a T>,
5  }
```

- Can point to an array of any length, or slice thereof
- Done, right?
- What about mutably borrowing from this type?
- Well, what if we want to pass ownership?

More Structs


```
1  struct SliceMut<'a, T> {
2      ptr: *mut T,
3      len: usize,
4      _phantom: PhantomData<&'a T>,
5  }
6
7  struct BoxStruct<T> { ... }
8
9  struct RcStruct<T> { ... }
10
11 struct ArcStruct<T> { ... }
```

- This is not composable
- And therefore doesn't scale well

What are DSTs

- Rust provides two build in types to solve this problem
  - `[T]`
  - `dyn Trait` (formerly `Trait`)
- Size of both is unknown at compile time
- They can't stand on their own as a variable or argument type

DSTs as Existential Types

- `[T]` is an existential array type: ∃ `n. [T; n]`
- `dyn Trait` is an existential value type: ∃ `T. T: Trait`

Unsizing

- As one of its few coercions Rust does unsizing:
  - `[T; n]` to `[T]`
  - `T` implementing `Trait` to `dyn Trait`
- This allows:
  - `&[T; 4]` → `&[T]`
  - `Box<[T; 4]>` → `Box<[T]>`
  - `Rc<[T; 4]>` → `Rc<[T]>`
- These pointers to DSTs have an additional word:
  - For `[T]` the length
  - For `dyn Trait` a pointer to the vtable

DSTs in Structs

- DSTs can occur in structs
- This makes the struct a DST too
- Only as the last field
- You can only create them via unsizing and via generics

DSTs and Traits and Generics

- All regular types automatically implement the `Sized` trait
- All generics are per default bound by the `Sized` trait
- You need to opt out of this default by requiring (allowing) `?Sized`

```
1 struct Foo<T: ?Sized> {
2     foo: u16,
3     bar: T
4 }
5
6 let x: &Foo<[u8]> = &Foo { foo: 12, bar: [0; 4] };
```

Last time's discussion

- Can you resize the object behind a `Box<[u8]>`
- Different from assigning a different `Box<[u8]>` to the same binding
- Maybe `std::mem::replace(&mut *the_box, [1, 2, 3, 4][..])`?

```
pub fn replace<T>(dest: &mut T, src: T) -> T
```

- There is an implicit `Sized` bound

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2019-04-03-dsts.pdf