Boxes and Recursive Data

CAS CS 392: Rust, in Theory and in Practice

February 24, 2025 (Lecture 10)

Outline

Reminder: Stack vs. Heap

Everything in Rust is put on the stack by default

Things put on the stack must have fixed size, known at compile time

The data associated with vectors and strings are put on the heap because *they're implemented that way*:

```
pub struct Vec<T> {
    ptr: NonNull<T>, // ignore the NonNull part for now
    cap: usize,
    len: usize,
}
```

(The structure itself it put on the stack like everything else)

Question

```c
int main(void) {
  int *x = (int*)malloc(sizeof(int));
  *x = 5;
  free(x);
  return 0;
}
```

What if we want to put something on the heap anyway?

- Suppose we have a very large array (of fixed size), but we want to make sure it has a single owner (side-stepping copy semantics)
- Suppose we wanted to work with data whose size is *not* fixed?

Boxes

Boxes are a type-safe way to allocating memory on the heap:

```
fn main() {
    let b = Box::new(5);
    println!("b = {b}");
}
// prints: b = 5
```

As usual, deallocating data is handled automatically through Rust's ownership system

Note: There's something interesting going on in this example, /how does rust know to print 5 and not something "box-like"?

Recursive Data

Boxes are necessary to define recursive data types like lists and trees:

```
// THIS DOES NOT COMPILE
enum List {
    Cons(i32, List),
    Nil,
}
fn main() {}
```

This code does not compile because Rust *cannot possibly know* how much space to allocate on the stack for this type

Type Layout

How much space do different structures take? How does Rust know?

```
struct Foo = {
    foo: i32,
    bar: i64,
    baz: i8
}
```

How much space does a value of the above structure take up?

Type Layout

The layout of every type in our program is determined at compile time

```
struct Foo = {
    foo: i32,
    bar: i64,
    baz: i8
}
```

A **type layout** consists of

1. Size (how many bytes does a value take up)
2. Alignment (what addresses values can be stored at, must be 2^k)
3. field offsets (where do fields live in the data, if applicable)

size_of and align_of

Rust provides primitives for determining the size and alignment of a type:

```
struct Foo {bar : i32, foo : i64, baz : i8}

fn main() {
    println!("{}", std::mem::size_of::<Foo>());
    println!("{}", std::mem::align_of::<Foo>());
    // print:
    // 16
    // 8
}
```

Type Layout Guarantees

```
struct Foo = {foo: i32, bar: i64, baz: i8}
```

Rust guarantees three things (at the moment) for structures:

1. Fields are aligned (they respect the alignment of the type of the field, this might require padding)
2. Fields do not overlap (seems obvious)
3. The alignment of a structure is the maximum over the alignments of its fields

Rust *does not* guarantee that fields are laid out so in the same order they're defined

What about enumerations?

```
enum Message {
    Quit,
    Move { x: i32, y: i64 },
    Write(String),
    ChangeColor(i32, i32, i32)
}
```

- every discriminant/constructor/variant gets a `u8` tag (so only 255 discriminants)
- and carries its data as if it where a structure

The size/alignment is the maximum over the size/alignment of very discriminant

Cons Lists

Coming back to lists, what is the size (and alignment) of this type?

```
// THIS DOES NOT COMPILE
enum List {
    Cons(i32, List),
    Nil,
}
```

The size/alignment of a `Cons` is (morally speaking) the same as that of:

```
struct ConsDeterminant {
    tag: u8,
    value : i32,
    tail : List
}
```

which is $1 + 4 + \textit{padding} + ???$

Size and Alignment of Boxes

A `Box` is just a structure with a `usize` pointer to data on the heap, so the size and alignment will match that of `usize` (which is 8 on my machine):

```
fn main() {
    println!("{}", std::mem::size_of::<Box<String>>());
    println!("{}", std::mem::align_of::<Box<String>>());
    // prints:
    // 8
    // 8
}
```

Size of a Cons Cell

Coming back to lists, what is the size (and alignment) of this type?

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

The size/alignment of a `Cons` is (morally speaking) the same as that of:

```
struct ConsDeterminant {
    tag: u8,
    value : i32,
    tail : Box<List> // usize
}
```

which is ???

Aside: Null Pointer Optimization

Given a structure with a single unit-like constructor, we can use the *null pointer* instead of a tag!

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

The size (and alignment) of a `Cons` is (morally speaking) the same as that of:

```
struct ConsDeterminant {
    value : i32,
    tail : Box<List> // usize
}
```

which is 16 (and 8).

Outline

Boxes vs. Pointers

Boxes are similar to pointers, but like references *we can't have "shared" boxes*

In particular, we can't define circular data structures

```
fn main() {
    // THIS DOES NOT COMPILE
    let mut l : List = Cons(1, Box::new(Nil));
    if let Cons (_h, t) = l {
        *t = l
    };
}
```

Boxes vs. References

The only real difference between boxes and references is that they don't automatically implement the borrow semantics of Rust

Any type in Rust can be made to behave like a reference using the `Deref` trait:

```
use std::ops::Deref;
struct MyBox<T>(T);

impl<T> Deref for MyBox<T> {
    type Target = T;
    fn deref(&self) -> &Self::Target {&self.0}
}
```

Using Boxes

If we want to make a reference to the data held, by the box, we can make a "reference" to the box itself:

```
fn main() {
    let x : Box<i32> = Box::new(5);
    let y : &i32 = &x;
    assert_eq!(*y, 5);
}
```

If we want to move the data from a box, we can deference the box itself

```
fn main() {
    let x = Box::new(5);
    assert_eq!(*x, 5);
}
```

Deref Coercions

Types which implement `Deref` are "chained" when dereferenced:

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}

fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m); // instead of hello(&(*m)[..]);
}
```

Outline

Boxes

Deref

Workshop

Task

In groups of three(ish):

- Implement `get` with `u32` indices for our recursive `List` type
- design a recursive data type for arithmetic expressions on integers and define an evaluator for it. For practice, have it return a `Result`, with an `Error` enumeration for dealing with division-by-zero