Rust Smart Pointers

Luca Abeni
luca.abeni@santannapisa.it

References and (Smart) Pointers

- Reference: additional name for a value/variable
- A rust reference always points to values/variables on the stack
- A rust reference can only borrow vaules (and never own them!)
- What about memory allocated from the heap?
  - Some other form of pointers is needed!
  - In rust, **smart** pointers: data structures embedding a pointer, and adding some features to it
  - Remember C++ smart pointers?

Smart Pointers in Rust

- Big difference with references: smart pointers can own data!
  - And when the smart pointer owning the data is destroyed, the data is automagically freed
- From the programmer's perspective, most smart pointers can be used as references
- Most important smart pointers: only pointer to memory allocated from the heap (`new` replacement), pointer with referene counting, and pointer with atomic reference counting (for multi-threaded applications)
- Smart pointers are also hidden in vectors and strings

Smart Pointers vs References

- A reference *borrows* some value, but does not actually owns it
- Standard borrowing rules:
  - At any given time, you can have either (but not both) one mutable reference or any number of immutable references
  - References must always be valid
- The second rule has important consequences (remember lifetime annotations?)
- Pointers can actually own the pointed values!
- Various kinds of pointers: only one owner per value, multiple owners per value, dynamic check of the borrowing rules, ...

Allocating Data from the Heap

- Dynamic memory allocation (from heap): `new` method of the `Box<T>` data type
  - Generic data type
  - Parametric respect to the type of the data to be allocated
- Of course, it uses RAII!!!
  - The data is freed when the smart pointer is destroyed
- Exercize: try to implement a recursive data type
  - Remember, for example natural numbers as a sum type?
  - Rust does not hide dynamic memory allocations, so...

Remember?

```
struct S {
  v: i32
}

fn WorkOnS() {
  let mut p = Box::new(S{v: 5});

  p.v = ...
    /* use p ... */
    ...
}
```

- Now we know the meaning of "`Box::new()`"
  - Notice: the type parameter "`S`" is inferred by the compiler
  - Otherwise, we could have used "`Box<S>::new(S{v:5})`"

Reference Counting

- Data allocated with "`Box::new()`" has one single owner
  - Property needed for RAII
  - Assignment has a move semantic
  - Alternative to move: explicitly duplicating the data ("`clone()`" method)
- Using `Rc<T>`, "`clone`" returns a pointer pointing to the same data... But increases a reference counter!
- Destroying the `Rc<T>` variable, the counter is decreased; when it is 0, the data is freed
- Note: "`Rc<T>`" is similar to a shared reference: it cannot be mutable

Runtime Borrow Checking

- For references (and `Box<T>`, and `Rc<T>`) borrow checking is performed at build time
  - it is not possible to get mutable data from "`Rc<T>`"
  - This does not build:

```
let t  = Box::new(S{v: 666});
let s  = &t;
let s1 = &mut t;
```

- `RefCell<T>` allows to perform the checks at runtime
  - Step in the wrong direction?
  - Probably yes, but in some cases it is unavoidable
  - Example: reference counting with mutable references!

Reference Counted Mutable References

- How to get a mutable reference from "`Rc<T>`"?
  - Simple idea: wrap a "`RefCell<T>`" inside the "`Rc<T>`"
  - Then, the "`Rc<T>`" can be cloned and the "`RefCell<T>`" can be mutably borrowed!

```
let v  = std::cell::RefCell::new(S{v:666});
let s  = std::rc::Rc::new(v);
let r1 = s.clone();
let r2 = s.clone();

println!("{} {}",
         (*r1.borrow()).v, (*r2.borrow()).v);
(*s.borrow_mut()).v = 0;
```

Summing Up...

- Number of owners:
  - `Rc<T>` enables multiple owners of the same data
  - `Box<T>` and `RefCell<T>` have single owners
- Build-time vs runtime checks:
  - `Box<T>` allows immutable or mutable borrows checked at compile time
  - `Rc<T>` allows only immutable borrows checked at compile time
  - `RefCell<T>` allows immutable or mutable borrows checked at runtime

Interior Mutability

- `RefCell<T>` allows mutable borrows checked at runtime
  - It is possible to mutate the value inside the `RefCell<T>` even when the `RefCell<T>` is immutable
- "Internal state" mutable even if the variable is immutable: interior mutability
- `Rc<T>` and `RefCell<T>` can be combined to have multiple owners for a variable that can be dynamically borrowed as mutable
  - Programming pattern sometimes found in Rust: `RefCell<T>` inside an `Rc<T>`

Smart Pointers as References

- How to actually use a smart pointer in rust?
  - In general, smart pointers are not references (they cannot "*directly replace*" a reference)...
  - ...But can be dereferenced (using the "`*`" operator) to get the wrapped value!
- So, if "`p`" is a "`Box<i32>`", then "`&(*p)`" is a "`&i32`"
  - Notice: "`*`" can often be omitted!
  - So, we can use "`&p`"
- Small exception: for "`RefCell<T>`" we must explicitly invoke "`borrow()`" or "`borrow_mut()`"
  - Getting a value of type implementing "`Ref<T>`" or "`RefMut<T>`"
  - It can be dereferenced to apply "`&`" or "`&mut`"

How Does this Work?

- Something like "`&(*p)`" looks very strange...
  - Dereferencing a pointer/reference to get a reference again???
- In reality, "`*`" is not just transforming a reference in the referenced value...
  - It can be applied to any type implementing the "`Deref`" trait...
  - "`fn deref(&self) -> &Self::Target`" is the method to be implemented ("`&Self::Target`" is the type of the referenced value)
- So, "`&(*p)`" "`deref()`" to get a reference, then dereferences it, and then gets a reference again!

Using Smart Pointers

- `Box<T>` can be used without complications (apply "`*`" to it, etc...)
- `Rc<T>` offers the "`clone()`" method to increase the reference counter (creating a copy)
  - How to avoid issues with circular references? Again, weak references!
  - "`Rc<T>::downgrade()`" returns a *weak reference*
  - Must be *upgraded* at runtime to be used (if the referenced vaue has been freed, "`upgrade()`" returns "`None`"
- `RefCell<T>` must be explicitly borrowed (runtime checking!) calling `borrow()` or `borrow_mut()`