LECTURE 20

Theory and Design of PL (CS 538)

April 06, 2020

ITERATOR ADAPTERS

FP FOR ITERATORS

- Many functional languages: operate on lists
- Rust: similar operations, but on iterators
- Transform iterators into new iterators
  - `chain`: glue two iterators in sequence
  - `zip`: pair up two iterators
  - `step_by`: iterator skipping every few elements
  - `skip/take`: skip or take first few elements
- All your favorites FP patterns
  - `map/filter/fold/scan`

EXAMPLE: MAPPING

- Common Rust operations defined on iterators

```
let v: Vec<i32> = vec![1, 2, 3];

let v2: Vec<i32> = v.iter()                 // get iterator
                    .map(|x| x + 1)         // increment each by one
                    .collect();             // turn back into vector
// same as: let v2: Vec<i32> = v.iter().map(|x| x + 1).collect();

println!("v: {}", v); // OK: .iter() doesn't take ownership
println!("v2: {}", v2);
```

- Chaining `.foo().bar().baz()` is Rust style

EXAMPLE: FILTERING

- Keep only elements satisfying predicate

```
let v: Vec<i32> = vec![1, 2, 3];

let v2: Vec<i32> = v.into_iter()          // takes ownership!
                    .filter(|x| x.is_even())
                    .collect();           // turn back into vector

println!("v2: {}", v2);
println!("v: {}", v); // Not OK: into_iter() took owernship
```

CHAINING CALLS, CHAINING STRUCTS

- Each call returns an intermediate struct
- Methods defined on these structs

```
let v: Vec<i32> = vec![1, 2, 3];

// let v2: Vec<i32> = v.iter().map(|x| x + 1).collect();

let v2iter: Iter<&i32> = v.iter();

let v2map: Map<Iter<&i32>, fn(i32)->i32> = v2iter.map(|x| x + 1);

let v2: Vec<i32> = v2map.collect();

println!("v: {}", v); // OK: .iter() doesn't take ownership
println!("v2: {}", v2);
```

RECAP: RUST REFERENCES

THE GOLDEN RULES

- Aliasing: two references to same memory
- In any scope, there can be either:
  1. *Any number* of immutable references
  2. *At most one* mutable reference
- … referring to the same data

> *One or the other: not both!*

PLAIN REFS: &T AND &MUT T

- Standard, economy class references
- Mutable/immutable view on some data
  - Does not own underlying data
  - Data guaranteed to be valid
- When a ref falls out of scope: nothing happens
  - No ownership, no destructor, no deallocation
  - May allow new borrows to be taken

BOX<T> TYPE

- Behaves almost exactly like a reference to T
  - Only difference: data is put on the heap
- Box owns the underlying data

```
let my_box = Box::new(String::from("foo"));  // store foo

let un_box = *my_box;  // get data from the box

println!("box = {}", un_box);  // looks like normal String
```

RECURSIVE TYPES

- Rust requires all data to have constant stacks size
- Problem for recursive types

```
enum IntList {
  Cons(i32, IntList),
  Nil,
}
```

- Compiler complains: don’t know size of IntList!

SOLUTION: USE A BOX

- Put the thing of unknown size (IntList) on the *heap*

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

- A bit awkward, but it works

```
fn main() {
  let list = Cons(1,
    Box::new(Cons(2,
      Box::new(Cons(3,
        Box::new(Nil))))));
}
```

STD::BOXED::BOX

- Owned data on the heap
- Behaves much like normal mutable reference
  - Can be dereferenced, assigned to, etc.
- When a box falls out of scope: heap deallocation
  - Owns data: guaranteed no live refs to data
- Can move out data by dereferencing
  - Special case for Box type!

MAKING REFERENCES SMARTER

“SMART POINTERS”

- Look like references, but do more
- Control over ownership, sharing, de-allocation, etc.
- We use these all the time in Rust
  - Examples: String, Vec, …
- Need unsafe Rust to implement these things

FIRST OPERATION: DEREFERENCE

- For references, * operation gets underlying data
  - Example: `*ref` returns target of `ref`
- Dot notation does something similar
  - Example: `ref.foo()`

SECOND OPERATION: DROP

- Data is dropped when its *owner* goes out of scope
- When reference is dropped, nothing happens
  - Reference borrows data, doesn’t own it
- Can customize drop to do more things

DEREFERENCING

THE DEREF TRAIT

- Treat smart pointers like regular references
- Get plain, immutable reference to data
  - `DerefMut` trait similar for mutable

```
trait Deref {
  type Target;

  fn deref(&self) -> &Self::Target;
}
```

- Compiler converts: `*sp` to `*(sp.deref())`

EXAMPLE

- Augment plain data with some extra side data

```rust
struct MyBox<T> {
  data: T,        // underlying data
  size: i32,      // side info
  flag: bool,     // side info
}

impl<T> Deref for MyBox<T> {
  type Target = T;

  fn deref(&self) -> &T {
    // get a reference to underlying data
    &(self.data)
  }
}
```

QUICK ASIDE: AUTO-DEREF

- Why do these all work?

```
fn is_none<T>(&self) -> bool     // method of Option<T>
                                 // take Option ref, to bool

let my_opt = None;

let b1 = (&my_opt).is_none();    // OK: &my_opt is ref
let b2 = my_opt.is_none();       // but my_opt is not ref

let b3 = (&&my_opt).is_none();   // wait
let b4 = (&&&&&&&&&my_opt).is_none();  // ???
```

COMPILER INSERTS DEREFS

- Infers how many * and deref needed
  - Exact rules are not exactly specified
  - Mostly: Just Works
- Current best description (from stackoverflow)
  - If have thing of type S and expecting type &T
  - Deref/* arbitrarily many times until type is T
  - Then add back a &
- Makes deeply nested refs much easier to use
  - Just don't think too hard about pointer types

DE-ALLOCATING

THE DROP TRAIT

- Describe *destructor*: what to run when cleaning up

```
trait Drop {
  fn drop(&mut self);
}
```

- Before var goes out of scope, call `var.drop()`
- Effect: tells stored data to do de-allocation

EXAMPLE

- Print a message when dropping data

```
struct DropLoudString { data: String }
impl Drop for DropLoudString {
  fn drop(&mut self) { println!("Dropping `{}`!", self.data); }
}

fn main() {
  let c = DropLoudString { data: String::from("foo") };
  {
    let d = DropLoudString { data: String::from("bar") };
    println!("DropLoudStrings created.");
    // Dropping `bar`!
  }
  // Dropping `foo`!
}
```

REFERENCE COUNTED POINTERS

MULTIPLE OWNERS

- Immutable underlying data
- Smart pointer tracks number of owners
  - Increments when pointer is copied
  - Decrements when pointer is dropped
- Data dropped when there are no owners

WHAT COULD GO WRONG?

- No owner: need to figure out when to deallocate
- Multiple references share view on data
  - Mutation is dangerous

IN RUST: STD::RC::RC

- `Rc<T>` is type of reference counted pointer to T
- `Rc::new(foo)`: make new pointer holding `foo`
- `Rc::clone(rc_pt)`: make a copy of `rc_pt`

EXAMPLE: SHARING LISTS

- Try to share a part of a list, but doesn't work

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

fn main() {
  let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
  let b = Cons(3, Box::new(a)); // OK: owner is now b
  let c = Cons(4, Box::new(a)); // Not OK: a is not owner
}
```

SOLUTION: USE RC

- Explicitly make call to `clone` when sharing

```
enum List {
  Cons(i32, Rc<List>),  // change Box to Rc
  Nil,
}

fn main() {
  // Note: a is now Rc<List>, not List
  let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));

  let b = Cons(3, Rc::clone(&a)); // OK: clone reference
  let c = Cons(4, Rc::clone(&a)); // OK: clone reference
}
```

TOY MODEL OF RC

- Main Rc struct: holds a `Box<T>`, int count
  - One total, for all users
- Rc handle struct: points to main struct
  - One per user
- Clone: handle -> main -> increment count
  - Copy the **handle** (not main struct!)
- Deref: handle -> main -> boxed data
- Drop: handle -> main -> decrement count
  - If count zero, drop main struct and box
  - Drop the **handle** (not main struct!)

WHY IS THIS (MOSTLY) SAFE?

- Track how many pointers to data, deallocate at zero
  - Danger: reference cycles will leak memory
- Ban mutation entirely
  - Don’t hand out mutable refs to data
  - Don’t implement DerefMut

SMARTER POINTERS

CLONE ON WRITE

- Smart pointer to some data
- If need immutable access: don’t clone
  - Multiple readers safely share same copy
- If need mutable access: clone an owned copy
  - Clone lazily, on demand

STD::BORROW::COW

- Essentially, an enum
  - Cow::Borrowed: points to borrowed value
  - Cow::Owned: points to owned value

```
let mut cow = Cow::Borrowed("moo");   // borrowed &str
println!("What does the cow say? {}", cow);   // doesn't clone
cow.to_mut().make_ascii_uppercase();   // clones to owned String
println!("What does the cow say now? {}", cow);   // MOO
```

WHAT COULD GO WRONG?

- Each holder of smart pointer thinks it owns data
- May try to mutate "own copy" of data
- Behind the scenes, may all be sharing same data
- Don't want other mutations to show up in my data

WHY IS THIS SAFE?

- Can only get mut ref through `to_mut`
- As long as no one calls this, it's safe to share
  - No mutation == no problem with aliasing
- Old idea in computer science

INTERIOR MUTABILITY

- Sometimes: immutable fn mutates “under the hood”
  - Essentially, lie about mutability
- Example: memoization
  - In first call, cache answer (mutate state)
  - In next calls, lookup answer
  - Want: client shouldn’t know about mutation!

STD::CELL::CELL

- Holds owned value T, gives out owned values
- Types lie: claim Cell is *immutably* borrowed

```
fn set(&self, val: T)

fn take(&self) -> T

fn replace(&self, val: T) -> T

let c = Cell::new(5);
c.set(6);
let six = c.take();
```

WHAT COULD GO WRONG?

- A lot, it turns out
- Might mutate when there are other immut refs out
  - Allowed since set/replace borrows immutably!

WHY IS THIS SAFE?

- Cell never gives borrows to T, only owned values!

STD::CELL::REFCELL

- Holds owned value T, gives out references to T
  - Alias rules checked at runtime: may panic!

```
fn borrow(&self) -> Ref<T>

fn borrow_mut(&self) -> RefMut<T>   // actually: mut borrow!

let c = RefCell::new(5);

let mut_ref = c.borrow_mut();
*mut_ref  = 7;

let other_ref = c.borrow();  // runtime panic: already mut ref
```

WHAT COULD GO WRONG?

- Even more stuff might go wrong
- Really handing out refs to the inner data T
  - Mutable and immutable refs to T?
  - Two mutable refs to T live at same time

WHY IS THIS SAFE?

- Need to enforce aliasing rules for safety
- RefCell: enforce rules at runtime
  - If borrowing rules fail, panic

IN MORE DETAIL...

```
fn borrow(&self) -> Ref<T>

fn borrow_mut(&self) -> RefMut<T>    // actually: mut borrow!
```

- Gives out “Ref” and “RefMut”
- Not actually references—more smart pointers!
  - Track how many borrows of RefCell are alive

PERVASIVE IN RUST

- Quite common in C++ as well
- Stay tuned: smart pointers for locking
  - Ref to locked value, exclusive access
  - Customized drop: auto unlock the lock!