LECTURE 15

Theory and Design of PL (CS 538)

March 11, 2020

TYPES IN RUST

NUMBERS

- Full range of integer types
  - `i32` is signed 32-bit, `u8` is unsigned 8-bit, …
- Floating point types
  - `f32` is 32-bit floating, `f64` is 64-bit, …

TYPES INFERENCE

- Most types inferred, sometimes annotations needed
  - Single `:`, not double `::`
- Type conversions using `as`

```
let guess            = "42".parse(); // what type to parse to?
let guess_u16: u16 = "42".parse();
let guess_u32: u32 = "42".parse();

let guess_u64: u64 = guess_u32 as u64;
```

BOOLS AND CHARS

```
let my_true_bool = true;

let my_false_bool: bool = false; // with annotation

let little_z = 'z';
let fancy_z = 'ℤ';
let heart_eyed_cat = '😻';
```

- Rust designed with support for UTF-8

STRUCTS

- Very similar to records (big tuples)
- First, define names and types of the fields:

```
struct User {
  username: String,
  email: String,
  sign_in_count: u64,
  active: bool,
}
```

CREATING AND ACCESSING

- Record syntax for creating, dot-syntax for accessing:

```rust
// make a struct variable of type User
let user1 = User {
  email: String::from("someone@example.com"),
  username: String::from("someusername123"),
  active: true,
  sign_in_count: 1,
};

// access various fields
let user1_email  = user1.email;
let user1_active = user1.active;
```

CREATING: SHORTCUT

- Initializing field from a variable with that name

```
let email = String::from("someone@example.com");
let username = String::from("someusername123");
let active = true;
let sign_in_count = 1;

let user1 = User { email, username, active, sign_in_count };
```

UNNAMED STRUCTS

- Structs without field names
- Access fields with `.0`, `.1`, `.2`, ...

```
struct Point(u32, u32, u32);

let my_point = Point(1, 2, 3);

let fst_coord = my_point.0;
let snd_coord = my_point.1;
let thd_coord = my_point.2;
```

TUPLES

- Get components via dot notation: `.0`, `.1`, etc.
- Or: use pattern matching

```
let foo = (500, 6.4, 1);                              // plain tuple

let bar: (bool, f32, i32) = (true, 0.1, 5);   // annotated

let (x, y, z) = foo;                                   // pattern match
println!("The tuple is ({}, {}, {})", x, y , z);

let x = bar.0; let y = bar.1; let z = bar.2; // projections
println!("The tuple is ({}, {}, {})", x, y , z);
```

MUTATING

- Can mutate individual fields of a mutable struct

```rust
// make a mutable struct variable
let mut user1 = User {
  email: String::from("someone@example.com"),
  username: String::from("someusername123"),
  active: true,
  sign_in_count: 1,
};

// change value of email field
user1.email = String::from("anotheremail@example.com");
```

STRINGS

- Special type for Strings: not just a list of characters!
  - Implementation is highly optimized
  - Memory allocation, resizing, etc. all automatic
  - See docs for many, many functions
- Build strings with constructor

```
let my_new_string = String::from("Hello!");

let my_bad_string = "Hello!";  // Not OK!
```

SLICES

- Often: want a view into a contiguous piece of data
- Can’t change it, but can read from it
- In Rust: called a *slice*

```
let array = [1, 2, 3, 4, 5];

let slice = &array[1..3];  // type: &[i32]

let fst = slice[0];  // 2
let snd = slice[1];  // 3
let thd = slice[2];  // 4
```

STRING SLICES

- Same idea, but for strings: special type `&str`

```
let s = String::from("hello world");

let hello = &s[0..5];   // type: &str
let world = &s[6..11];  // type: &str
```

SLICES DON'T OWN DATA

- Someone else owns the data, not the slice
- Can copy slices, pass slices around, etc.

```rust
let s = String::from("hello world");

let hello  = &s[0..5];
let hello2 = hello;

println!("Hello? {} Hello! {}", hello, hello2);  // This is OK
```

ENUMS IN RUST

THINK: SUM TYPES

- Type taking several possible values (OR)

```
enum Color {
  Red,
  Green,
  Blue,
}

enum Time {
  HoursMinutes(i32, i32),
  Minutes(i32),
}
```

CONSTRUCTING ENUMS

- Take name of enum type, add constructor after it

```
let my_color       = Color::Red;
let my_time        = Time::HoursMinutes(6,30);
let my_other_time  = Time::Minutes(1080);
```

A FAMILIAR FRIEND: OPTION

- Rust's version of `Maybe`
- Parameterized by a type T (just like `Maybe a`)
- Don't need prefix `Option::`

```
enum Option<T> {
  Some(T),
  None,
}

let something = Some(5);
let nothing   = None;
```

PATTERN MATCHING IN RUST

- Just like `case` in Haskell...

```
let maybe_int = Some(42);

match maybe_int {
  None     => println!("Nothing here!"),
  Some(n)  => {
    println!("Just an int: {}", n);
    println!("Doing lots of stuff!");
  }
}
```

RUST FUNCTIONS

TOP-LEVEL FUNCTIONS

- Typical way to declare functions
- Type annotations needed for parameters and return
  - Can be inferred, but required for documentation
- Unlike in Haskell: functions can perform effects

```
fn foo(x: i32, y: i32) -> i32 {
  ... x ... y ...
}
```

CALLING FUNCTIONS

- Normal syntax: supply arguments and get result
- No partial application: must supply *all* arguments

```
fn add_up(x: i32, y: i32) -> i32 { x + y }

fn main() {
  let added = add_up(10, 12);

  println!("The sum is: {}", added);
}
```

MOVING VERSUS BORROWING

“MOVING” ARGUMENTS

- Operationally: arguments passed “by value”
- Ownership of argument passes into the function
  - Caller can’t use arguments after calling!
  - “Arguments moved into function”
- Function can return argument to return ownership

EXAMPLE: MOVE

```rust
fn take_own(s: String) { ... }

fn main() {
  let my_string = String::from("Hello!");

  take_own(my_string);  // Pass the string to function

  println!("Still there? {}", my_string);  // Not OK: it's gone!
}
```

“BORROWING” ARGUMENTS

- Operationally: arguments passed “by reference”
- Ownership of argument *doesn’t* change
  - Original owner (caller, caller-of-caller, …) owns arg.
- “Function borrows arguments” (from the owner)
  - Will give it back to owner when done with it

EXAMPLE: BORROW

```
fn take_borrow(s: &String) { ... }

fn main() {
  let my_string = String::from("Hello!");

  take_borrow(&my_string);                 // "Borrow" ref to fn

  println!("Still there? {}", my_string);   // OK: still owner
}
```

OTHER WAY OF DEFINING FNS: METHODS

ADD FUNCTIONS TO A STRUCT/ENUM

- Useful pattern: associate fns with structs/enums
  - Primary argument: the object (“self”)
  - Other arguments: stuff to access/modify self
- “Methods” in object-oriented programming

SYNTAX: IMPL BLOCKS

```
struct Person {
  name: String,
  age: u32,
}

impl Person {
  fn print_me(&self) {
    println!("name: {} age: {}", self.name, self.age);
  }

  fn get_name(self) -> String { self.name }

  fn get_age(&self) -> u32 { self.age }
}
```

SELF PARAMETER

- First parameter of method is always called “self”
  - Never write type `T`: comes from `impl T`
  - But: annotations (`&`, `mut`) are very important!

TRANSLATING METHODS

```
impl Person {
  fn get_name(self) -> String { self.name }
}

// Takes ownership of person! Same as:
fn get_name_fn(p: Person) { p.name }
```

TRANSLATING METHODS

```
impl Person {
  fn print_me(&self) {
    println!("name: {} age: {}", self.name, self.age);
  }
}

// Borrows person! Same as:
fn print_me_fn(p: &Person) {
  println!("name: {} age: {}", p.name, p.age);
}
```

CALLING METHODS

- Use dot-notation, can chain calls together
- Chaining is awkward for regular functions

```
let my_person = Person {
  name: String::from("Chicken Little"),
  age: 2,
};

my_person.print_me();  // same as: print_me(&my_person);

// Get name and append ", Jr." to it
let my_name_jr = my_person.get_name().push_str(", Jr.");
```

MODELING AN IMPERATIVE LANGUAGE

CORE LANGUAGE: REVISITED

- Model essential language features
  - Which programs are well-formed?
  - How should programs behave?
- Imperative languages: memory, variables, state, …
- Our plan: layer on top of (pure) expressions
  - Also called a “While-language”

EXTENSION 1: MEMORY

- Different models capture different aspects
  - Scope of variables? Allocation? Types?
- Simplest: memory (“store”) maps var. names to ints
  - Global, integer variables only

EXPRESSIONS IN STORES

```
var   = "x" | "y" | "z" | ... ;

bexpr = "true" | "false"
      | bexpr "&&" bexpr | aexpr "<" aexpr | ... ;

aexpr = var | num-cons
      | aexpr "+" aexpr | aexpr "*" aexpr | ... ;
```

- Expressions can mention program variables
  - Meaning depends on the current memory
  - Not: variables in lambda calculus (fn args)
- But: expressions don't *change* memory

EXTENSION 2: COMMANDS

- Add commands as a new layer of the language

```
comm = "skip"                                     (* do-nothing comm *)
     | var ":=" aexpr                             (* assign to var *)
     | comm ";" comm                              (* sequencing *)
     | "if" bexpr "then" comm "else" comm         (* if-then-else *)
     | "while" bexpr "do" comm ;                  (* while loops *)
```

OPERATIONAL SEMANTICS

- How do imperative programs step?
  - Depends on the current memory!
- Idea: define how command-store *pairs* step
  - Model how the program and memory change

BLACKBOARD (AND WR4)