# LECTURE 14

Theory and Design of PL (CS 538)

March 09, 2020

# THANKS FOR DOING COURSE FEEDBACK!

# COMMON COMPLAINTS

- Written notes for stuff on blackboard
- A lot more examples (of everything)
- More interactive coding/demos
- More readings for reviewing
- Clearer instructions on HW/WR
- Slow down!!!

# IMPERATIVE PROGRAMMING

# SEQUENCE OF STEPS

- Strongly influenced by real-world machine models
  - Think of program as a **list of instructions** to run
  - Use instructions to modify the machine state
- Notable languages
  - **FORTRAN**, **ALGOL**, **BASIC**, **Pascal**, **C** (1950s-70s)
  - OO: **Smalltalk**, **Simula**, **Java**, **C++** (1980s-90s)
  - Today: **Python**, **Ruby**, **C#**, **Perl**, **Go**, **Rust**, …

# MUTABLE STATE

- Idea of *state* is central to imperative programming
  - Registers, memory, file system, …
- Instructions describe how to *mutate* the state
  - Read, write, update, …

# SEPARATE STATE FROM CODE

- In pure FP, the code is both *program* and *state*
  - Evaluation depends on the program
  - Evaluation steps reduce the program to a value
- In imperative programming, state is separate
  - Current state of machine is not visible in code
  - Code tells us how to update the (implicit) state

# BENEFITS

- Close fit to most real-world machine models today
  - Machines have registers, memory, state
  - Machine code is (basically) list of instructions
  - Well-suited to low-level and embedded systems
- Often (but not always):
  - Natural translation to machine code
  - Fast performance

# WEAKNESSES

- Mutable state is hard to reason about
  - Can break modularity/abstraction
  - Calling a procedure can change the state in complex or unexpected ways
- Memory management is tricky
- Concurrency and parallelism are a challenge

# MANAGING MEMORY

# TYPICAL MODEL

1. Code requests memory from system
    - Should be fresh piece of memory
2. Read/write/update data via pointer or reference
    - Read files, maintain datastructures, etc.
3. Return memory when done
    - Releases memory, gives it back to system

# TRADITIONAL CHALLENGES

- If automatic memory management:
  - Slower and more unpredictable performance
  - Simply not acceptable for some applications
- If manual memory management, possible bugs:
  - *Memory leak*: forget to free memory after done
  - *Double free*: free memory more than once
  - *Use-after-free*: use memory you’ve given back

# CONCURRENCY

# THE PRESENT IS PARALLEL

- CPU speed no longer doubling every 18 months
  - Hitting limits: power and cooling
  - Moore’s law has stopped (for a while now)
- Instead of faster cores, get *more* cores
  - 2, 4, 8, 16, 32 separate CPUs
  - How to get 2x, 4x, 8x, 16x, 32x speedups?

# THREADS OF EXECUTION

- Run several parts of program in parallel
  - Better use of multiple cores, datacenters, etc.
- Break up program into different threads
  - Can be good idea, even on single core
  - I/O thread waits for file system, GUI responsive

# TRADITIONAL CHALLENGES

- No longer just a single list of instructions!
  - How to split up program?
  - How to coordinate accesses to shared memory?
  - Hard to think about all possible interleavings
- Lots of common bugs
  - *Data races*: several threads access same memory in non-deterministic order
  - *Deadlock*: no thread can run, waiting on each other

# THE RUST LANGUAGE

# HISTORY AND PRINCIPLES

- Graydon Hoare's side project at Mozilla (2006)
- Heavily supported by Mozilla as a research language
- Goal: a safe, concurrent, practical systems language
  - *Efficiency*: Very fast, programmer has control
  - *Safety*: eliminates memory and concurrency bugs
  - *Modern PL features*: influenced by FP, type systems

# EXTREMELY FAST

- Competitive with other systems language (C/C++/…)
  - But: substantially safer
- Automatic memory management, but without GC
  - Safe and predictable performance
- Designed for concurrency throughout

# ELIMINATES MEMORY BUGS

- Know at compile-time when memory should be freed
  - No memory leaks
- Double frees, use-after-frees caught at compile time
- Novel ownership/lifetime mechanism ensures safety
  - Based on two PL ideas: regions and affine types

# "FEARLESS CONCURRENCY"

- Data races are caught at compile time
  - Leveraging ownership/lifetime mechanism again
- Eliminates whole class of common concurrency bugs
  - Data races are notoriously tricky to debug
- Supports different kinds of concurrency

# OTHER FP IDEAS ABOUND

- Modern type system
  - Datatypes, polymorphism, traits, type inference
- Emphasis on mutability and immutability
  - Encourages pure code
- FP-style programming with higher-order functions
  - Anonymous functions, maps, filters, folds, …

# REAL-WORLD ADOPTION

- Extremely active community
  - New version of the language every 6 weeks
- Tons of libraries and package
- Most popular language in StackOverflow survey
- Larger developments by Mozilla
  - Much of latest version of Firefox rewritten in Rust
  - Leverage safe concurrency, memory management

# OUR PLAN

# CORE RUST (4 WEEKS)

- Will use The Rust Programming Language book
- Fairly linear, we'll mostly go in order this time
- Many concepts will be familiar from Haskell
  - Strong types, including datatypes
  - Constructors and pattern matching
  - Traits and generics

# CONCURRENCY (2-3 WEEKS)

- Concurrency basics and concepts
- Concurrency features in Rust
- Core language for concurrency

# ADVANCED TOPICS

- Some selection of...
  - Error handling
  - Rust macros
  - Asynchronous programming
  - Unsafe Rust

# HOMEWORKS

- Three homeworks, same format
  - Rust Warmup: out today
  - HW4: Getting started writing Rust programs
  - HW5: Binary search tree, datastructures
  - HW6: Concurrency

*Start early and ask for help!*

# WE WILL CARE ABOUT...

- Memory
  - Where does it live: stack or heap?
  - When is it allocated/de-allocated?
  - Piece of data, or pointer to data?
- Aliasing
  - How many variables refer to piece of data?
  - Which variables are allowed to change data?
- Going (a little) fast

# RUST TOOLS

# READ THE DOCS

- Rust docs are extremely high quality
- Read the intro to get an idea of the module
- If needed, look up specific functions
  - Just like in Haskell: pay attention to the types!

# READ THE BOOK

- The Rust Programming Language (TRPL)
- Very high quality (free) textbook
  - Lots of examples!
- We will follow this material closely

# USE THE PLAYPEN

- Located here
- Type, compile, run code online, see result
- Use tools for formatting, linting
- Share/link code snippets (with instructors)

# CARGO

- Main package/build system for rust
- Wraps around the rust compiler, `rustc`
  - No need to call `rustc` by hand
- Useful commands
  - `cargo check`: Type/borrow checking (fast)
  - `cargo build`: Build an executable (slower)
  - `cargo run`: Run the thing
  - `cargo clean`: Clean up temporary files

# OTHER USEFUL TOOLS

- clippy (`cargo clippy`)
  - Suggestions for cleaner code
  - Follow them unless there's a good reason not to!
- rustfmt (`cargo fmt`)
  - Automatic code formatter
  - Enforce consistent style on Rust source code

# GROUND RULES

1. Don't use `unsafe` code blocks.
2. Use the default (stable) version, not beta/nightly
3. Try to avoid panicking commands
   - These commands halt program if they fail
   - Examples: `panic!`, `unwrap`, `expect`, ...
   - Unfortunately, sometimes unavoidable
4. Try not to write very slow code
   - Prefer loops over recursion
   - But: read HW instructions!
5. Compiler is very noisy, but fix your warnings

# A TASTE OF RUST

# DOING BASIC I/O

- Printing: `println!("value: {}", var)`
- Reading a line: `io::stdin().read_line`

```
fn main() {
  let mut guess = String::new();  // new mutable string variable
  let secret    = 42;             // secret number is always 42

  println!("Guess a number!");

  io::stdin().read_line(&mut guess);  // read into guess

  println!("You guessed: {}", guess)
}
```

# BASIC ERROR HANDLING

- `read_line` returns something of type `Result`
  - Like `Either` in Haskell: `Ok(val)` or `Err(e)`
- We can chain another function call to handle error

```
fn main() {
  let mut guess = String::new();
  let secret    = 42;
  println!("Guess a number!");

  // Chain two function calls: read_line and expect
  io::stdin().read_line(&mut guess)
             .expect("Failed to read line");
             // panic if read_line fails

  println!("You guessed: {}", guess);
}
```

# MATCHING AND COMPARING

- Rust has pattern matching and *traits*
  - Very similar to typeclasses
- `Cmp` trait gives comparison function (in Haskell, `Ord`)

```
// pattern match
let guess: u32 = match guess.trim().parse() {
  Ok(num) => num,
  Err(_)  => { println!("Not a number! Picking 0..."); 0 },
}

// compare: cmp method coming from Cmp trait
match guess.cmp(&secret) {
  Ordering::Less    => println!("Too small!"),
  Ordering::Greater => println!("Too big!"),
  Ordering::Equal   => println!("Just right!"),
}
```

# A SMALL GUESSING GAME

```
loop {
  io::stdin().read_line(&mut guess)
             .expect("Failed to read line");
  let guess_num: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
  }
  match guess_num.cmp(&secret) {
    Ordering::Less    => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal   => { println!("Just right!") ; break ; }
  }
}
```

# VARIABLES AND ASSIGNMENTS

# BASIC DECLARATION

```
let x = 5;
println!("The value of x is {}", x);
```

- Let-bindings to declare variables; types inferred
- Variables belong to a block

# BRACES MARK BLOCKS

- Can open and close new blocks with braces
- Inner blocks can use variables from surround blocks
- Outer blocks can't use variables from inner blocks

```
let outer = 5;
// Start new block
{
  let inner = 6;
  println!("The value of outer is {}", outer);    // OK
}
// End new block
println!("The value of inner is {}", inner); // Not OK
```

# A NORMAL EXAMPLE

```rust
let x = 42;

println!("The int x is: {}", x);  // OK

let y = x;

println!("The int y is: {}", y);  // OK

println!("The int x is: {}", x);  // OK
```

# A STRANGE EXAMPLE

```
let x = String::from("A string!");
println!("The string x is: {}", x);  // OK
let y = x;
println!("The string y is: {}", y);  // OK
println!("The string x is: {}", x);  // Not OK???
```

# OWNERSHIP

1. Each piece of data has an *owner*
    - Thing responsible for deallocating data
2. Each piece of data has *exactly* one owner
    - If data has no owner, it is deallocated (*dropped*)

# OWNERSHIP IS UNIQUE

- Fundamental concept in Rust
- When assigning, ownership is *moved*
- By default, types have “move semantics”

# A STRANGE EXAMPLE

```
let x = String::from("A string!");     // Owner: x
println!("The string x is: {}", x);
let y = x;                            // Owner: y
println!("The string y is: {}", y);  // OK: y is owner
println!("The string x is: {}", x);  // x isn't the owner!
```

# IMPLICIT MOVES

- Generally: data is *not* copied—data is moved
- For some types: copied, instead of moved
  - Usually: for primitive, simple types
  - Must be explicitly marked in type definition
- These types are said to implement `Copy`
  - Or: types have “copy semantics”

# A NORMAL (?) EXAMPLE

```
let x = 42;

println!("The int x is: {}", x);   // Owner: x

let y = x;                          // Make copy of 42

println!("The int y is: {}", y);   // Owner: y

println!("The int x is: {}", x);   // Owner: x
```

# DEFAULT: IMMUTABLE

```
let x = 5;                                      // OK
println!("The value of x is {}", x);

let y = x + 1;                                  // OK
println!("The value of y is {}", y);

x = 6;                                      // Not OK
println!("The value of x is {}", x);
```

- Variables can only be set once
- Setting again: compiler error

# WHY IMMUTABLE?

- Immutable variables are easier to think about
  - Given `let x = 5;`, can replace x by 5 below
- Require programmer to explicitly mark mutable vars
  - Only use if they really need it
- Helpful information for compiler
  - Optimizations
  - Sharing

# VARIABLE SHADOWING

- Can redeclare same variable several times

```
let x = 5;
println!("The value of x is {}", x);          // 5

let x = x + 1;
println!("The value of x is {}", x);          // 6

let x = x + 2;
println!("The value of x is {}", x);          // 8
```

- Note: not recursive definitions (like Haskell)

# DECLARING MUTABLE

- Use keyword `mut` for let-bindings

```
let mut x = 5;                                              // OK
println!("The value of x is {}", x);

x = 6;                                                      // OK
println!("The value of x is {}", x);

x = x + 1;                                                  // OK
println!("The value of x is {}", x);
```

# STATEMENTS AND EXPRESSIONS

# TRADITIONALLY: SEPARATION

- Expressions: don't change the state
  - Compute by *evaluation* (rewriting)
  - Evaluate to some value
  - No side-effects
  - Example: Haskell programs
- Statements: transform the state
  - Compute by *execution*
  - Produce some final state

# RUST BLURS THE DIFFERENCE

- "Expressions": produce some final value
- "Statements": does not produce value
  - Effectively, something ending in a semicolon
  - Does not produce a value

*Both may change the state!*

# CONTROL FLOW

# “CONTROL”

- Recall: program executes a sequence of statements
- During execution, “control” is the current statement
- Also sometimes called *program counter*

# “FLOW”

- Control moves steps from statement to statement
- Statements can redirect where the control goes next
- *The* central concept in imperative programming

# SEQUENCING

# THE SEMICOLON

- Main use: gluing two statements together

```
let mut x = 1;
let mut y = 100;

x = y;
y = y + 1;
```

- Order matters! Different result:

```
y = y + 1;
x = y;
```

# OTHER USE: DISCARD RESULT

```
let mut input = String::new();

io::stdin().read_line(&mut guess);
//               ^--- Returns something!

println!("Guessed: {}" guess);
```

- `read_line` returns a value of type `Result`
- Trailing semicolon discards this value

# BRANCHING

# IF-THEN-ELSE

- Hopefully familiar…

```
let number = ...;

if number < 5 {
  println!("so small!");
} else if number > 10 {
  println!("so large!");
} else {
  println!("so OK!");
}
```

# CAN PRODUCE VALUE

- Branches must produce same type of value

```
let number = ...;

let branched = if number < 5 {
  "big!"
} else if number > 10 {
  "small!"
} else {
  "OK!"
}
```

# PATTERN MATCHING

- Match on an enumeration (sum type)

```
let x = 42;
let y = 55;

let cmp_result = x.cmp(&y);

match cmp_result {
  Ordering::Less    => println!("so small!"),
  Ordering::Equal   => println!("exactly equal!"),
  Ordering::Greater => println!("way big!"),
}
```

- Again, branches can produce values (of same type)

# IF-LET MATCHING

- Sometimes: want to check for specific constructor

```
let maybe_string = Some(String::from("Hello World!"));

...

if let Some(str) = maybe_string {
  ... str ...
} else {
  ...  // Can't use str here!
}
```

# REPEATING

# LOOP

- Repeats a command (forever)
- Use `break` to exit loop, `continue` to jump to start

```
let mut x = 20;

loop {
  x = x + 1;
  if x < 42 {
    println!("not yet...");
    continue;
  } else if x = 42 {
    println!("done!");
    break;
  } else {
    println!("uh-oh");
  }
}
```

# WHILE

- Repeats a command while some condition is true

```
let mut x = 20;

while x != 42 {
  x = x + 1;
  println!("not yet...");
}
println!("done!");
```

# WHILE-LET MATCHING

- Almost the same as if-let, but in a loop

```
let mut maybe_string = Some(String::from("Hello World!"));

...

while let Some(str) = maybe_string {
  ... str ...
}
```

# FOR-LOOPS

- Rust for-loops iterate over range, like this:

```
let my_array = [10, 20, 30, 40, 50];

// For-loops in Rust automatically use iterator
for element in my_array {
  println!("the value is: {}", element);
}
```