Rust Basics

CPSC 310 - Programming Languages

Type Safety in Programming Languages

- In a type-safe language, the type system enforces well defined behavior. Formally, a language is type-safe iff

  $\Gamma \vdash e : t$ and $\Gamma \vdash E$ implies $E \vdash e \Rightarrow v$ and $\vdash v : t$ or that e does not terminate.

  - $E \vdash e \Rightarrow v$ says that e evaluates to v under environment E
  - $\Gamma \vdash e : t$ says that e has type t under type environment Γ
  - $\Gamma \vdash E$ says that E is compatible with Γ, that is, for all x, $E(x) = v$ implies $\Gamma(x) = t$ and $\vdash v : t$

C/C++: Not Type-Safe

- Spatially unsafe
  - Type safety is violated by buffer overflows
  - Example

```
int x = 1, *p = &x;
int y = 0, *q = &y;
*(q+1) = 5; // overwrites p
```

- Temporally unsafe
  - Type safety is violated by dangling pointers
  - Example

```
int *x = ...malloc();
free (x);
*x = 5 // invalid pointer dereference
```

Automatic Memory Management

- Data may allocated explicitly or implicitly; data reclamation occurs automatically
- A garbage collector traces pointers in use by the program, starting from the stack and global variables
- Reference counting (for example, C++ smart pointers) keeps track of the number of references to a block of memory
- These techniques impose space and run-time costs

Memory Management in (Type-Safe) OCaml

- Local variables live on the stack
- Tuples, closures, and constructed types live on the heap
- Heap data reclaimed via garbage collection
- Example

```
let x = (3, 4) (* heap allocated *)
let f x y = x + y in f 3 (* result heap-allocated *)
None (* not on the heap -- just a primitive *)
Some 42 (* heap allocated *)
```

Language Choices

- C/C++
  - Type-unsafe
  - Low level control
  - Performance over safety and ease of use
  - Manual memory management
- Java, OCaml, Python, ...
  - Type safe
  - High level
  - Ease-of-use and safety over performance
  - Automatic memory management

Rust: Type-Safe and Performant

- A Mozilla sponsored public project since 2010
- Key properties
  - Type safe
  - Thread safe
  - Zero-overhead abstractions (performant)

Features of Rust

- Lifetimes and Ownership
- Traits as the core of an object-like system
- Variables default to immutability
- Data types and pattern matching
- Type inference
- Generics (parametric polymorphism)
- First-class functions
- Efficient C bindings

Rust Compiler and Build System

- Rust programs can be compiled using `rustc`
  - Source files end in suffix `.rs`
  - Compilation, by default, produces an executable
- Preferred: use the `cargo` package manager
  - will invoke `rustc` as needed to build files
  - will download and build dependencies
  - based on a `.toml` file `.lock` file

Using cargo

- Make a project, build it, run it

```
cargo new hello_cargo --bin
cd hello_cargo
cargo build
./target/debug/hello_cargo
```

Function Example

```
// comment
fn main() {
    println!("Hello, world!");
}
```

Let Statements

- Let statement example:

```
let x = 9000;
let x = x + 1; // shadow x
x  // 9001
```

- Let statement with type annotations

```
let x:i32= 9000;
let y:i32 = x + 1;
y  // 9001
```

Conditionals

- Example

```
fn main() {
    let n = 5;
    if n < 0 {
        print!("{} is zero", n);
    } else if n > 0 {
        print!("{} is zero", n);
    }
    else {
        print!("{} is zero", n);
    }
}
```

- Conditionals are expressions, and each branch must have the same type.

Recursive Function Example

```
fn fact(n:i32) -> i32 {
    if n == 0 { 1 }
    else {
        let x = fact(n-1);
        n * x
    }
}

fn main() {
    let res = fact(6);
    println!("fact(6) = {}", res);
}
```

Mutation

- Mutation is useful when iterating

```
fn fact(n: u32) -> u32 {
    let mut x = n;
    let mut a = 1;
    loop {
        if x <= 1 { break; }
        a = a * x;
        x = x - 1;
    }
}
```

Looping

- While loops
  - `while <expression> <block>`
- For loops
  - `for <pattern> in <expression> <block>`
- Example

```
for x in 0..10 {
    println!("{}", x);
}
```

Looping (Continued)

- `loop`, `while`, and `for` are expressions
  - return the final computed value
  - `break` may take an expression which becomes the loop's final value
- Example

```
let mut x = 5;
let y = loop {
    x += x - 3;
    println!("{}", x);
};
print!("{}", y); // prints 35
```

Data: Scalar Types

- Integers
  - signed: `i8`, `i16`, `i32`, `i64`, `isize`
  - unsigned: `u8`, `u16`, `u32`, `u64`, `usize`
- Characters (unicode)
  - `char`
- Booleans
  - `bool = {true, false}`
- Floating point numbers
  - `f32`, `f64`
- Note: arithmetic operators are overloaded

Tuples

- Tuples
  - 0-tuple: `unit ()`
  - n-tuple type: *(t1, . . . , tn)*
  - n-tuple expression: *(e1, . . . , en)*
  - Accessed by pattern matching or like a record field
- Example

```
let tuple = ("hello", 7, 'a');
assert_eq! (tuple.0, "hello");
let (x, y, z) = tuple;
```

Arrays

- Array operations
  - Creating an array (can be mutable or not), but must be a fixed length
  - Indexing
  - Assigning to an array index
- Example

```
let nums = [1,2,3];
let x = nums[0]; // 1
let mut xs = [1,2,3];
xs[0] = 1; // xs is mutable
let y = nums[5]; // run-time error
```

Array Iteration

- Rust provides a way to iterate over a collection (which includes arrays)
- Example

```
let a = [1, 2, 3, 4, 5];
for element in a.iter() {
    println!("value: {}", element);
}
```

  - `a.iter` produces an iterator
  - the `for` syntax calls `.next()` on the iterator until no elements are left

Testing

- In any language, we need to test code
- In most languages, testing requires extra libraries
- Testing in Rust is a first-class citizen
    - The testing framework is built into `cargo`

Unit Testing in Rust

- Unit testing is for local or private functions
  - these tests can be in the same file as the code
- Use `assert!` to test that something is true
- Use `assert_eq!` to test that two things that implement the `PartialEq` trait are equal

Unit Testing in Rust Example

```
fn add(a: i32, b: i32) -> i32 {
    a - b // bad implementation
}

#[cfg(test)]
mod tests {
    #[test]
    fn test_add() {
        assert_eq!(add(1,2), 3);
    }
}
```

Integration Testing In Rust

- Integration testing is for APIs and whole programs
- Creating a `tests` directory
- Create different files for testing major functionality
- Files do not need `#[cfg(test)]` or a special module, but still need `[test]` around each function
- Tests refer to code as if were an external library
  - Declare it as an external library using `extern crate`
  - Include the functionality you want to test with `use`

Integration Testing In Rust Example

```
extern crate my-project;
use my-project::add;

#[test]
pub fn test_add() {
    assert_eq!(add(1,2), 3));
}
#[test]
pub fn test_negative_add() {
    assert_eq!(add(1,-2), -1));
}
```

Running Tests

- `cargo test` runs all tests
- `cargo test s` runs all tests that contain `s` in the name
- By default, console output is hidden
  - use `cargo test -- --nocapture` to unhide it

Fun Fact

- The original Rust compiler was written in OCaml
- Now, the Rust compiler is written in Rust
  - How is this possible?
- Bootstrapping
  - The first compiler written in Rust is compiled by the compiler written in OCaml
  - Use the binary from the Rust compiler to compile itself
  - Keep updating the binary through self-compilation
  - Be careful not to lose that binary