Introduction to Rust

Luca Abeni
luca.abeni@santannapisa.it

About this Course

- 20 hours of lesson (2 credits)
- Course home page:
  `retis.santannapisa.it/luca/RustIntro`
- Contact me via email for more information
- About the exam, two possibilities:
  - small project + discussion
  - interview on theoretical aspects of the course

Why Rust?

- Rust: “modern” programming language, often presented as *safe* system language
  - What is a “safe” programming language?
  - What is a “system” programming language?
- Characterized by very strict static (build-time) checks
  - Why build-time checks?
- Advantages with respect to C++?
- Advantages with respect to $<$*your favourite safe language*$>$?

Safety

- No unique or formal definition
  - Many different definitions in literature
- Informally: a program is considered “safe” if it is possible to formally prove that it behaves correctly
  - “Behaves correctly”?
  - Or, “it does not do anything dangerous”...
  - Different possible variations...
- What about “safe” programming languages?
  - Safe programming language $\rightarrow$ enforces safety
  - A well-formed program cannot do anything dangerous
  - Given a well-formed program it is possible to formally prove that it behaves correctly

Different Kinds of Safety

- Type safety: well-formed programs cannot exhibit bugs due to type errors
  - Applying the wrong operation on the wrong type, ...
- Memory safety: well-formed programs cannot exhibit bugs due to wrong memory accesses
- Thread safety: well-formed programs cannot exhibit race conditions, deadlocks, and synchronization errors
- Other kinds of safety...
  - For example, a well-formed progam has a well-defined behaviour (no UBs in C, etc...)

Memory Safety

- No bugs due to wrong memory accesses...
  - Difficult to provide a generic definition
- Definition “by examples”... Wrong memory access:
  - Buffer overflow
  - NULL pointer dereference
  - Use after free
  - Use of uninitialized memory
  - Illegal free (of an already-freed pointer, or a non-malloced pointer)
- Things like “no accesses to uninitialized memory” do not properly catch buffer overflows, etc...

Memory Safety vs Type Safety

- Sometimes, there is no clear distincion between type safety and memory safety
- Clear buffer overflow (violation of memory safety):

```
int *v = malloc(sizeof(int) * 10);
v[10]  = 666;
```

- What about this:

```
int v[10];
v[10] = 666;
```

- Is it a buffer overflow or a type error?
  - Defines an array of 10 elements, and accesses the 11th...
- OK, C arrays are pointers, but what about C++:

```
std::vector<int> v(10,0);
v[10] = 666;
```

Enforcing Safety

- Safety can be enforced at compile time
  - Unsafe programs — whatever this means — do not even build
- Or at execution time
  - Some kind of “trusted language runtime” ensures that nothing bad happens
- According to someone, a safe program is a program that can rely on a trusted runtime
- Languages like Java try a mix of the two
  - No free() $\rightarrow$ remove the possibility to have use after free, etc
  - The JVM also enforces consistency, etc...

Breaking Memory Safety

- Features that might break memory safety:
  - No array bounds checks (or, is this type safety???)
  - Pointer arithmetic
  - NULL pointers (someone says, only if they cause UB)
  - Low-level memory management
- Low-level memory management:
  - Explicit C-style malloc()/free() (some say ”use new” and ”do not free()”)
  - Explicit assignment of arbitrary values to pointers

Possible Solutions/Mitigations

- Some “coding standards” generically forbid dynamic memory allocation
  - This is crazy: they ban the usage of functions!
- Some others (MISRA C) forbid dynamic allocation from the heap (malloc()/free())
  - Still, a partial solution.
- Alternative: using a garbage collector
  - Coming from functional programming languages; then used by Java
- Pointers in general are dangerous (some languages try to avoid them)

Static vs Dynamic Checks

- Consider the code

```
int v[10];
v[10] = 666;
```

- Should it fail to compile, or should it generate an exception at runtime?
- Static type checking: build failure
  - Early notification of (potential) bugs
  - Not always possible: what about `v[i] = 666;`?
- Dynamic type checking: exception/crash
  - Still safe (???)... Someone says “to make C safe, change all the UBs into crashes”...
  - Less useful for developers... But more for users?
  - Need for runtime support

Static or Dynamic?

- Dynamic checks are more permissive... Consider

```
int StrangeFunction(bool v)
{
  if (v) {
    x = 10;
  } else {
    x = "WTH???";
  }

  if (v) {
    return x * 2;
  }

  return len(x);
}
```

- But, is this really useful?
- If my program has potential bugs, I want it to fail to build!

Static Typing and Static Checks

- Static typing: programs with (even potential) type errors fail to build
- Dynamic typing: programs with type errors crash/generate exceptions
  - Still safe, but I prefer early notification
- Static typing requires a strong type system
  - Example: avoid the C's "automatic type promotion"
- We will see that this can help with memory safety too

The Dream

- Goal: “problematic code” (code that can have potential issues) does not even build
  - Eliminate an entire class of vulnerabilities before they ever happen
  - Cost: some valid code is considered invalid
- Need for some support at the language level!
  - Type theory can help, here!
  - Not a new idea: functional programming languages have already been there (for example)!
- Avoid heavyweight runtimes
  - Garbage collection, etc...

Tools for Safety — 1

- Static code analysis tools: search for possible issues in the code (without executing it)
- Taint analysis: check how “corrupted data” can affect the system
  - Performed as static analysis on source code or binary code
- Tools like valgrind, Address Sanitizer (asan) or other sanitizers, etc...
  - Maybe associated with fuzz testing
  - Still, this is testing, does not prevent dangerous code to build

Tools for Safety — 2

- Lots of warnings from compilers...
  - Warnings tend to change from compiler to compiler and from version to version
  - Only considered as “suggestions”
- Adopting “safe” development practices
  - Again, coding rules... Can be checked with some tools, at least
- Manual code review

Summing Up

- Lots of external tools for code analysis
  - Not really integrated with the language
- Mechanisms to detect memory errors, concurrency errors, and similar at runtime
  - Useful for testing
  - Prevent UBs
  - Need some runtime support (kasan does exist, but needs support in the Linux kernel!)
- Type/Memory safe languages exist
  - Java, C#, Haskell, Go, pick your name
  - All need a “not so lightweight” runtime
  - Still, safety is sometimes intended as “exception at runtime”...

Type/Memory Safe Languages

- Impossible to build programs that result in memory errors at runtime
- Again, various definitions of “memory error”...
- Example: Java
  - Null pointers do exist!
  - ...And null pointer dereference can happen even if you do not explicitly use null pointers!
  - But Java is safe because null pointer dereferences result in exceptions!
- Safety is often checked only dynamically
  - Sometimes, there are no other options!
- What about safe system languages?

System Languages and Safety

- Bad news: system languages have to be unsafe...
  - Why? Think about I/O...
  - To access an I/O device, raw (and unchecked) memory access is needed...
- Similitude with pure functional languages
  - A pure functional language allows no side effects...
  - ...But side effects are needed! (again, I/O...)
  - Solution: isolate side-effect in a runtime/abstract machine/well-defined software component
- Maybe, it is possible to precisely isolate unsafe sections of code?
  - Of course, this risks to open cans of worms...

Source of some Problems, again

- Buffer overflow
  - Can be statically checked only in some cases
- Issues with pointers
  - NULL pointer dereference
    - Can we really avoid NULL pointers???
  - Issues with memory de-allocation (use after free, illegal free)
    - Can we avoid C-style free()...
    - ...Without relying on garbage collectors?
  - Use of uninitialized memory
- Can we avoid pointers???

Programs: Code and Data

- Von Neumann architecture: programs == sequences of instructions that operate on data
  - Instructions and data are stored in memory
  - Long sequences of 0 and 1...
- Programming in machine language is not simple (reading/writing long sequences of bits!)
  - Assembly helps a little bit, introducing mnemonics for the machine instructions, and symbolic names for memory locations
- High-level languages introduce variables, types, and values

Variables and Values

- Variable $\leftarrow$ high-level programming languages
  - Used to abstract programs from the usage of physical/virtual memory
  - “Box” (set of memory locations) that can contain a value
  - Referenced by using a symbolic name
- Value: sequence of bits encoding some high-level concept (number, character, string, ...)
  - The encoding depends on the *type* of the variable
- Data type: defines the semantics of the variable
  - Set of possible values the variable can contain
  - Operations such values
  - ...

Immutable Variables

- Variables can be mutable or immutable
- Immutable variable: binding between a symbolic name and a value
  - Environment: set of bindings (name $\rightarrow$ value)
  - Function mapping names into values
- Variable declarations modify the environment
- There is no way to modify the value bound to a variable name
  - No assigments! Only initializations...
  - The only thing we can do is to define a new binding that *shadows* the old one

Mutable Variables

- The environment maps names into “boxes” (variables), not directly into values
- Additional function (memory) mapping variables into their contained values
  - Assignments modify the memory function, changing the value assigned to a variable in a variable (R-Value in C/C++)
- Aliasing: the same variable can have multiple names

Pointers

- Pointer type: special type, expressing references to variables
  - Possible values: memory addresses (of variables)...
  - ... + one special value, representing invalid pointers
  - The NULL value!!!
- Dereference operator: accesses the value contained in the pointed variable
  - Dereferencing the NULL value results in a runtime error!
- NULL is a value like the others; NULL dereferences cannot result in build errors

More on Data Types

- Every programming language has a set of *primitive types*
  - And many languages allow to define new types
- Simple way to define new types: apply sum or product operations to existing types
  - Product $\mathcal{T}_1 \times \mathcal{T}_2$: type with possible values given by couples of values from $\mathcal{T}_1$ and $\mathcal{T}_2$
  - Sum $\mathcal{T}_1 + \mathcal{T}_2$: type with possible values given by values from $\mathcal{T}_1$ or values from $\mathcal{T}_2$
- Sum == disjoint union; Product == cartesian product
- If $|\mathcal{T}|$ is the number of values of type $\mathcal{T}$, then $|\mathcal{T}_1 \times \mathcal{T}_2| = |\mathcal{T}_1| \cdot |\mathcal{T}_2|$ and $|\mathcal{T}_1 + \mathcal{T}_2| = |\mathcal{T}_1| + |\mathcal{T}_2|$

Algebraic Data Types

- A set (the set of the language's data types), a sum operation and a product operation... It's an algebra!
  - Algebra of the data types; types are called Algebraic Data Types!
- Issue: the sum is a disjoint union...
  - Easy to do "float + bool" (type with possible values integers or booleans)...
  - ...But what about "int + int" (or similar)?
  - The types have to be tagged somehow...

Algebraic Data Types and Constructors

- Solution adopted by many programming languages: do not sum types directly, but first apply a *tagging function* to them
  - Constructor: function generating the values of the type to be summed
  - Summing types generated by different constructors, the issue is solved!
- Variant: set of values generated by a constructor
  - Different constructors generate disjoint variants
  - Hence, instead of “int + int” we can use “Left(int) + Right(int)”

Examples

- C unions are a special case of tagged sum
- "test = i(int) + f(float)" is

```
union example {
    int i;
    float f;
};
```

- Of course, algebraic data types are more generic (0-arguments or multi-argument constructors, etc...)
- All constructors with 0 arguments: enum type
- Haskell, ML and others fully support ADT

```
datatype test = i of int | f of real;

data Test = I Int | F Float
```

Example: Option Type

- Type containing a value or nothing
  - Two constructors: “Nothing” (without arguments) and “Just” (with one argument of the desired type)
- Example: integer or nothing $\rightarrow$ Option_int = Nothing + Just(int)
- Idea: instead of using a null pointer...
- ...Use an option type: Pointer_to_int = Nothing + Just(int *)
  - Advantage: only the “Just” variant can be dereferenced...
  - NULL pointer dereferences do not even compile!

Generic Data Types

- The definition of a new type might depend on a "type variable"
  - Parametric type, depending on another type "`T`", denoted by a variable
  - Type variables, generally indicated as greek letters
- Example: generic option type
  - Not "integer or nothing", but "type α or nothing"
  - α: type variable
- In Haskell, something like

```
data Option a = Nothing | Just a
```

- Used for many other things too (lists, Monads, ...)

Recursive Data Types

- To define a data type, we must (also) define all its possible values
- Set of possible values $\rightarrow$ can be defined by induction...
- Can induction/recursion be used to define a new data type?
  - How? We need induction base and induction step
  - Induction base: one (or more) constructor(s) having 0 parameters (or, no parameters of the data type we are defining)
  - Induction step: constructor having a parameter of the type we are defining
- Looks... Confusing??? Let's look at some examples!

Recursive Data Types: Example

- Let's define the "natural numbers" data type (set of values: $\mathcal{N}$)
  - $0 \in \mathcal{N}$: constructor `zero` (with no parameters)
  - $n \in \mathcal{N} \Rightarrow n + 1 \in \mathcal{N}$: constructor `succ`, having as an argument a natural number

```
datatype nat = zero | succ of nat;
```

```
data Nat = Zero | Succ Nat
```

- How to use this funny definition?
  - Combination of *pattern matching* and *recursion*
  - Familiar to people knowing functional programming

More Interesting Example: Lists

- Lists can also be defined by induction/recursion (simple example: list of intergers)
  - Inductive base: an empty list is a list
  - Inductive step: A non-empty list is an integer followed by a list
- Recursive Data Type: a non-empty list is defined based on the list data type (constructor receiving a list as a parameter)
- Two constructors
  - Empty list constructor
  - Constructor for non-empty lists

Lists as RDTs — 1

- Two constructors
  - Empty list constructor (no parameters)
  - Constructor for non-empty lists (two parameters: an integer and a list)
- Other operations
  - `car`: returns the first element of a non-empty list (head)
  - `cdr`: given a non-empty list, returns the “rest of the list”

Lists as RDTs — 2

- How are lists generally implemented?
- Functional languages (Haskell, ML Lisp & friends, ...)
  - Recursive data type!!!
  - "`cons`" constructor: parameter of type `int * list` (or, a parameter of type `int`, but returns a function `list -> list`)
- Imperative languages: pointers!
  - Structure with 2 fields (types "`int`" and "`list*`")
  - Second field: pointer to next element
  - Cannot be of type "`list`", $\rightarrow$ use "pointer to `list`"!

RDTs vs Pointers

- See? Imperative languages use pointers and explicit memory allocation...
  - Adding an element to list implies doing some malloc()/new for a node structure, setting some “next” pointers, etc...
- ...In functional languages, RDTs avoid the need for pointers, and memory allocation/deallocation is hidden...
  - Adding an element in front of a list “`l`” is as simple as “`let l1 = cons(e, l)`” or similar!
  - The implementation of the language abstract machine will take care of allocating memory, etc...

Enforcing Type/Memory Safety

- Focus on static checks
  - When possible...
- Need for a “strong type system”
- No NULL pointers/references
  - Option types might help, here
  - Some languages already provide them
- No “arbitrary assignments” to pointers / no pointer arithmetic
- No `free()`, but no garbage collection!
  - How to do this?

Strong Type Systems

- So, what is a “strong type system”?
  - And, what is a type system after all?
- Many different definitions (once again...)
  - Purpose of a type system: defining, detecting, and preventing illegal program states
  - Done by applying constraints on the usage of variables, values, functions, ...
- Pretty theoretical stuff, we need a more pragmatic definition
- Strong type system: imposes more constraints and restrictions

Type Systems: Pragmatic Definition

- Less theoretical definition... A type system is composed of:
  - A set of predefined types
  - A set of mechanisms for building new types (based on existing ones)
  - A set of rules for working with types
    - Equivalence, compatibility (automatic conversion), inference, ...
  - Rules for type checking (static or dynamic)
- Let's see a pragmatic definition of "strong" too...

Things to Avoid — 1

- No Python-like dynamic typing

```
v = 10
print(v)

v = "Hi␣There!"
print(v)

v = None
print(v)

v = 3.14
print(v)
```

- Even if the language allows it, avoid this (ab)use of dynamic typing

Things to Avoid — 2

- No C-style automatic promotion

```
#include <stdio.h>

int main()
{
  double v = 6.66;       int    v2 = v * 2;
  int    i = 6.6 / 2.2; double d  = 6.6 / 2.2;

  printf("V=%f␣V2=%d␣I=%d␣D=%f\n", v, v2, i, d);

  return 0;
}
```

- Even with well-defined rules, static checks are weaker
  - Difficult to understand if "`int i = 6.6 / 2.2`" is a typo or a wanted conversion

Type Checking and Inference

- Strong type system $\rightarrow$ more constraints/restrictions
  - Strict rules for assignments/bindings
- The compiler can algorithmically check if a variable has the right type
  - So, why forcing the programmer/user to specify types?
  - Instead of checking the correctness of type annotations, the compiler can directly *infer the type* of each variable!
- Few exceptions due to polymorfism or similar...

Examples of Type Inference

- C++ with the "`auto`" keyword

```
auto i = 5;
```

  - But "`auto`" is more useful for things like this:

```
auto f = [](int a , int b) {
        return a + b;
};
```

- Standard ML

```
> val a=5;
val a = 5: int
> val f = fn x => x / 2.0;
val f = fn: real -> real
> fun fact n = n * fact (n - 1);
val fact = fn: int -> int
```

References, with No NULL

- Things like "`int *p = 0x666;`" must be forbidden
  - Pointer/reference initialization/assignment only:
    - From dynamic allocation (either automatic or `new`, but not `malloc()`)
    - From existing variables
- Pointers/references are always valid
  - NULL/invalid pointers/references do not exist
  - Can be handled by using option types

Garbage Collectors

- Traditional way to avoid explicit memory deallocation
- Periodically check the heap
  - Scan for unused (non-reachable) memory
  - Re-compact referenced memory in the heap, and free then one not recompacted
  - ...
- In general, non-trivial actions at runtime
  - Might need a non-negligible amount time
  - Need a complex runtime
- Can this complexity/overhead be reduced?
  - Is it possible for the compiler to automatically insert the needed memory deallocations in the generated code?

Some Ideas (from C++!)

- Resource Acquisition Is Initialization (RAII)
  - Some kind of resource is allocated in the constructor of a class $\rightarrow$ instantiating an object allocates the resouce
  - Resource de-allocated in the destructor $\rightarrow$ when the object goes out of scope, the resource is deallocated
- Useful, for example, for mutexes ("`std::lock_guard`")...
- ...But think about memory (dynamically allocated from the heap) as a "resource"
  - Memory allocated when a "pointer" is instantiated, and freed when it goes out of scope!

Reference Counting

- How to implement the RAII approach on dynamically allocated memory?
- First idea: reference counting
  - Counter associated to each chunk of dynamically allocated memory
  - New reference to the memory $\rightarrow$ increase the counter
  - Reference destroyed (out of scope) $\rightarrow$ decrease the counter; if counter == 0, free the memory
- Low overhead, but something is still needed at runtime
- Fails miserably with circular references (including doubly-linked lists)

Special Case: Single Reference

- If we remove the possibility to have multiple references to the same data structure, things become simpler
- Dynamically allocated memory with only one reference to it $\rightarrow$ when the reference is destroyed (goes out of scope), deallocate the memory
  - No need for complex runtime support
  - The compiler can add what is needed in the generated code
- Problem: how to enforce the “only one reference to the allocated memory” property?

Smart Pointers

- Smart Pointer: data structure encapsulating a pointer (and eventually a reference counter)
- Allows to control how the pointer is used
  - Can implement reference counting
  - Can easily enforce the “only one reference” property (and free the memory when the data structure is destroyed)
- Example: C++ “`std::shared_ptr`”, “`std::weak_ptr`” and “`std::unique_ptr`”
  - Allow to implement RAII with different constraints (multiple references to single “resource”, some forms of circular references, single reference to “resource”)

Smart Pointers — 2

- Shared pointers: implement reference counting
- Weak pointers: to be used with shared pointers (get a reference without increasing the counter)
  - Allow to implement doubly-linked lists, but risk to open another can of worms
- Unique pointers: only one valid reference to the pointer memory
  - Copy between unique pointers (or direct assignment) is not possible
  - "`std::move()`" must be used instead (see "move semantic")
  - Destructor/reset $\rightarrow$ delete the pointed object

Programming Style and Programming Languages

- All of this can be done with many different programming languages...
- ...But most of the existing languages do not actually enforce the usage of safe programming techniques
  - Example: Some PLs have option types...
  - ...But also provide “forced unwrapping” (or similar) things!
- Some languages even allow to break the safety provided by some constructs!
  - C++ provides smart pointers...
  - ..But does not forbid “traditional” pointers, that can easily compromise the usage of smart pointers!

Rust History

- Started in 2006 by a Mozilla developer (Graydon Hoare) as a side project
  - First version of the compiler written in OCaml (functional programming language)
- In 2009, Mozilla realized that Firefox was suffering because of a large amount of segfaults
  - These issues could be addressed by using a “safer” language
  - ...So, Mozilla started sponsoring Rust development
- First self-hosted compiler in 2010/2011
- First release (v1) in 2015
- Continuos community growth

Rust Evolution

- Originally sponsored by Mozilla for Firefox, then evolved in a “strange way”...
  - Considered for a long time only as a “system programming language”
  - System programming: not really related to web browsers...
- Today has multiple applications (see `https://www.rust-lang.org`, “Build it in Rust”):
  - Command Line tools
  - WebAssembly
  - Networking applications
  - Embedded systems

Rust in Action

- Mozilla uses it in its new browser engine (https://servo.org/)
- Microsoft proposed as a proactive way to address security and prevent vulnerabilities:

  `https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/`
- Intel (“Rust is the future of systems programming”)
  - Used Rust for its QEMU replacement:

    `https://github.com/cloud-hypervisor/cloud-hypervisor`
- Amazon did something similar:

  `https://github.com/firecracker-microvm/firecracker`
- ...

Various Visions of Rust

- Today, Rust is supported by a large community (not only Mozilla)
  - Various visions of the language and of the “ecosystem”
- Rust as a language: safety, performance, “zero-cost abstractions” (abstractions without overhead), ...
- Rust as an ecosystem:
  - Not only compiler, but also other tools (cargo package manager, ...)
  - Set of “crates” that can be used by rust applications