Safe Programming Concepts

Luca Abeni
`luca.abeni@santannapisa.it`

February 24, 2020

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 by:
  - 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!