Static Analysis for C++ Rust-Like Lifetime Annotations

Susana Monteiro, Nuno Lopes

Background

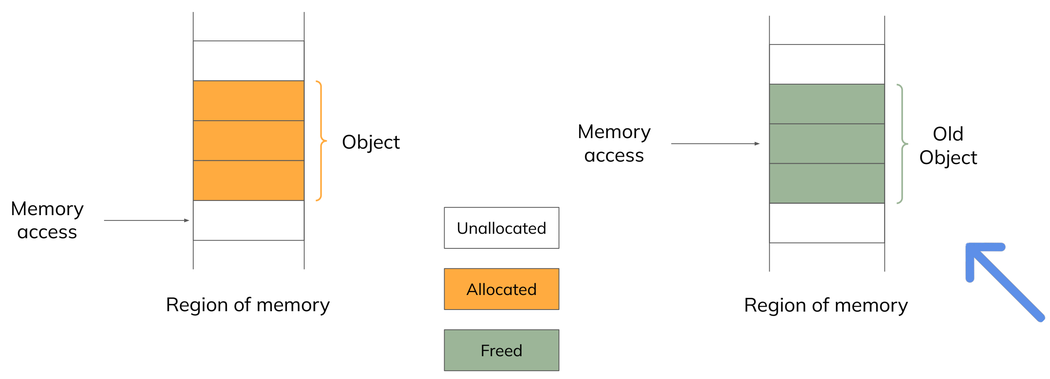
- Most security vulnerabilities today are related with memory safety.
- Some languages, namely the C++ programming language, are not memory-safe.
- One solution is to use a memory-safe language instead: Rust.
- There are two kinds of memory errors:

SPATIAL

- Bounds of objects
- Rust enforces it at run time

TEMPORAL

- Lifetime of objects
- Rust enforces it at compile time



- Translating C++ code to Rust has to be done incrementally, which requires interoperability between these languages.
- This interoperability brings up some challenges, such as Rust's concept of lifetimes.

```rust
fn first_char<'a>(s: &'a str) -> &'a str {
    &s[0];
}
```

- Rust-like lifetimes annotations were recently implemented in Clang as an extension to C++ and these can be used to solve the above challenge.

Rust and Lifetimes

- Lifetime: the scope for which a reference is alive.
- Each reference has a lifetime.
- Statically ensure that references are valid, preventing temporal memory errors.
- Developers can add lifetime annotations to the code.

```rust
fn smallest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() < y.len() {
        x
     } else {
        y
     }
}
```

returns a reference with the same lifetime ('a) as the parameters

- Constraint: the return value should be valid as long as the function's arguments are.
- The verification of lifetime annotations in Rust is flow-insensitive.

```rust
fn f<'a, 'b>(x: &'a i32, y: &'b i32, b: bool) -> &'a i32 {
    let mut p = x;
    if b {
        return p;
    }
    p = y;
    x
}
```

overrides previous value of p

```
error: return p;
            ^ function was supposed to return data with lifetime `'a`
            but it is returning data with lifetime `'b`
```

Lifetime Annotations in C++

- Rust-like lifetime annotations were recently implemented in Clang.
- The function `smallest`, written previously in Rust, can be written in C++, with the respective lifetimes annotations $a.

```cpp
const std::string& $a smallest(
    const std::string& $a s1, const std::string& $a s2) {
  if (s1.length() < s2.length()) {
    return s1;
  } else {
    return s2;
  }
}
```

Static Analyzer

- We are developing a static analyzer.
- **Goal**: check if Rust-like lifetimes annotations in C++ code are correct.
- The tool is being developed in Clang, using its static analysis capabilities.
- The analysis is flow-insensitive and intra-procedural.

```cpp
int *$b fn(int *$a p) {
  int *x = p;
  int *y = x;
  return y;
}
```

Implementation

The analysis is divided into **3 steps**.

Here we show the steps to analyze the previous example.

Step 1: create a graph of dependencies between variables with no lifetime annotation.

```
x -> p
y -> x
```

Step 2: propagate the dependencies until there are no more changes.

```
x -> p
y -> x, p
```

Step 3: check if the code is valid and generate the necessary warnings.

```
example.cpp:2:9: warning: function should return data with lifetime
'$b' but it is returning data with lifetime '$a'
        return p;
        ~~~~~~~~^
example.cpp:1:20: note: declared with lifetime '$a' here
int *$b fn(int *$a p) {
           ~~~~~~~~^
```

Next steps

- Returning objects created inside of a function.
- Pointer aliasing.
- Rust's concept of ownership.

This work was partially supported by Google and national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020.