Programming Languages: Recitation 6

Goktug Saatcioglu

03.07.2019

1 HW4 Problem 1

- The issue here is that we do not check whether when we do `grades[entry]` that `entry` is within bounds of the array `grades`.
- In C array bounds are not checked and accessing a location beyond the bounds of an array is undefined behavior. This means that we can either access some other location or crash the program.
- When we start the program we see that we create a stack frame for `main`. This stack frame also contains space for `student_id` at some fixed location in the frame. After receiving input from the user which also allocates a new stack frame and then pops it, we allocate a new stack frame for the function `get_transcript_entry`. Then we create two other stack frames and pop them for the calls to `get_transcript_size` and `load_transcript`. At this point there is some fixed distance between the stack frame for `get_transcript_entry` and `main` meaning there is a fixed distance between `get_transcript_entry` and the variable `student_id`. We can give a negative value to `entry` so to access the contents of the stack and find such a value to get the number stored in `student_id`.
- Note 1: We may also need to pass a positive value depending on how the stack is oriented.
- Note 2: If the variable `student_id` is not stored on the stack due to the compiler only using registers for the variable, then this attack will not work.

2 HW4 Problem 2

- With deep binding and static scoping we create a closure for `set_x` before passing it into `foo`. Here, by the referencing environment, the `x` in `set_x` refers to the global `x`. So any calls to `set_x` inside `foo` will update the global `x`. Then we define `print_x` in `foo` and this time due to static scoping it will print the local `x` inside the function. So the call `s(y)` updates the global `x` and the call to `print_x` will print `0`. Then we return the function `print_x` will create another closure and we see by the referencing environment that now `x` refers to the local `x` in `foo`. The call to `set_x` updates the global `x` and then we call `p()` which is the `print_x` returned by `foo`. Here we restore the closure and see that we were referring the local `x` so we print `0`. Overall, we print the following.

```
0
0
```

- With shallow binding and dynamic scoping we no longer create closures for function calls and use the most recent declaration of a variable. So in `foo` the call `s(y)` will use `set_x` which will set the local `x` to 1 since that is the most recent one. Then the `print_x` function will print the local `x` giving us `1` printed out. Next, the function is returned but no closure is created. The call to `set_x` will set the global `x` since that is the most recent variable on the stack so the global `x` is 2. Then the call `p()` will be the call `print_x()` which will use the current referencing environment and print the global `x`. So we print out `2`. Overall, we print the following.

```
1
2
```

3 HW4 Problem 3

- When both variables are call by value we evaluate the actuals before they are passed into the function and then pass a copy of the actuals to the function. Furthermore, the order of evaluation is left-to-right. So the first parameter becomes 1 and the second parameter becomes 2 while also updating the global `z` to 2. We do `x = y + y` where `y` is 2 so that `x` becomes 4 but this has no effect on `z` because we have call by value. So we print out `4` and then after exiting the function we print out `2` as we had updated `z` before. Overall, we print the following.

```
4
2
```

- Now x is call by reference meaning any update to `x` will also update the value stored in the actual. Since the evaluation strategies are both strict we evaluate the actuals before passing them in. So we end up with the same scenario as above but now the expression `x = y + y` will also update `z` since `x` is call by reference. So we print `4` inside the function and then print `4` after exiting the function because we updated `z` inside `params`. Overall, we print the following.

```
4
4
```

- This time we have call by value and call by name. The `z` inside the expression `z = z + 1; z` will refer to `z = 1;`. So we pass in `1` and the expression `z = z + 1; z` to the function. Then the expression `x = y + y` can be viewed as (using textual substitution) `x = z = z + 1; z + z = z + 1; z` where the first sub-expression becomes 2 and sets `z` to 2 and then the second sub-expression becomes 3 and sets `z` to 3. So `x` becomes 5, we print `5` and make no change to `z` before the function call. Since we updated `z` to 3 we end up printing `3` with the very last print statement. Overall, we print the following.

```
5
3
```

- Now `x` is call by reference so the expression `x = z = z + 1; z + z = z + 1; z` will update `z` to 5 after all sub-expression are evaluated. Overall, we print the following.

```
5
5
```

4 HW4 Problem 4

- The solution is given below.

```
def until(b: => Boolean)(body: => Unit): Unit = {
  if (!b) {
    body
    until(b)(body)
  }
}
```

If the condition is not met then we should evaluate the `body` of the loop and then call the function again. Since `body` is also a function, it will use the referencing environment of the first call to `until` and then update that environment via all calls to `body`. The same is true for `b` as it is also a function. So everytime we evaluate either their closures are returned and the referencing environment is updated if necessary. This way we can get custom control-flow constructs in Scala.

5 Let Bindings in OCaml

- We can introduce local variable bindings in OCaml using let-bindings, i.e. we do `let ... in ...` where the `...` refers to some other valid OCaml expression.
- For example consider the following OCaml code.

```
let add_mod x y =
  let z = 10
  in
  (x + y) mod z
```

This is equivalent to the following Scala program.

```
def add_mod(x: Int, y: Int): Int = {
  val z = 10
  (x + y) % z
}
```

So we see that we introduce a local function `z` that just returns a constant value 10. The function is only in scope for the expression `(x + y) mod z` and this is indicated using the keyword `in`.

- Now consider the following OCaml code.

```
let f a b =
  let x = a + b in
  let mul y = x * y
  in
  x + mul 2 + mul 3
```

This program is equivalent to doing the following in OCaml.

```
let f a b = x + x * 2 + x * 3
```

An equivalent Scala code is as follows.

```
def f(a: Int, b: Int): Int = {
  val z = 4 + 5
  def mul(y: Int) = x * y
  x + mul(2) + mul(3)
}
```

As we can see, each let binding creates a function and introduces it into the local scope. So, `let x` is actually a constant function that return the sum of `a` and `b` and `let mul` is a function that evaluates `x` and then multiples the result by some value `y`. Each consecutive `let` introduces the expression into the expressions below it (thus, introducing it into the scope of the lower expressions) and we finally get a result with `x + mul 2 + mul 3`.

- What happens with the following OCaml progam?

```
let f a b =
  let mul y = x * y in
  let x = a + b in
  x + mul 2 + mul 3
```

Answer: We get an error since `x` is not bound to anything.

- What happens with the following OCaml program?

```
let x = 0
let f a b =
  let mul y = x * y in
  let x = a + b in
  x + mul 2 + mul 3
```

Answer: It compiles as `x` is now bound to something.

- What's the difference between the program above and the program given below?

```
let f a b =
  let x = 0 in
  let mul y = x * y in
  let x = a + b in
  x + mul 2 + mul 3
```

Answer: Now `x` is a local function rather than a global one so its scope is only active inside `f`.

- Finally, let bindings are actually synctactic sugar for curried functions. So, the function

```
let mul x y = x * y
```

becomes

```
let mul = fun x -> fun y -> x * y
```

which reads as `mul` is the function that takes a parameter `x` and returns a function that takes a parameter `y` which then calculates the result we are interested in.

- Consider again the following OCaml code.

```
let f a b =
  let x = a + b in
  let mul y = x * y
  in
  x + mul 2 + mul 3
```

Question: What does the desugared version of this program look like.

```
let f = fun a -> fun b ->
  let x = a + b in
  let mul = fun y -> x * y
  in
  x + mul 2 + mul 3
```

6 Computing with Lists in OCaml

- From the lecture notes we can see that we can compute with lists in OCaml. Some examples include: fold left, fold right, map, reverse, append and so on.
- The OCaml standard `List` module (library) provides many list functions.
- See: https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html.
- Let's implement the `map2` function. The specifications are as follows. `map2` takes a function that takes two arguments, a list `xs` and another list `ys` and returns `[f x1 y1; ...; f xn yn]`. If the two lists are not the same length then the function should raise an `Invalid_argument` but we can ignore this for now.
- The non-tail recursive solution is given below.

```
let rec map2 f xs ys = match xs, ys with
  | [], [] -> []
  | x :: xs, y :: ys -> (f x y) :: map2 f xs ys
```

We see that the compiler complains since the pattern matching is not exhaustive. What is the problem? If the two lists are not of the same length we have actually 4 different possibilites: (empty, empty), (not empty, empty), (empty, not empty) and (not empty, not empty). One option is to give feedback to the user using `failwith`. This looks as follows.

```
let rec map2 f xs ys = match xs, ys with
  | [], [] -> []
  | x :: xt, y :: yt -> (f x y) :: map2 f xt yt
  | _ -> failwith "The two lists have different length!"
```

The other is to raise an `Invalid_argument` exception. Let's try this below.

```
let rec map2 f xs ys = match xs, ys with
  | [], [] -> []
  | x :: xt, y :: yt -> (f x y) :: map2 f xt yt
  | _ -> raise Invalid_argument
```

We see that `Invalid_argument` is actually a function that takes an argument. So we can actually print descriptive error messages depending on the context. If the length of the first list is larger than the length of the second list we would like to report that to the user. How can we do this?

```
let rec map2 f xs ys = match xs, ys with
  | [], [] -> []
  | _, [] -> raise (Invalid_argument "Len(list1) > len(list2)!")
  | [], _ -> raise (Invalid_argument "Len(list2) > len(list1)!")
  | x :: xt, y :: yt -> (f x y) :: map2 f xt yt
```

Either approach works and gives us a working version of `map2`.

- Finally, there is something else wrong with our function, what is it? It is not tail recursive! The tail recursive version is given below.

```
let map2 f xs ys =
  let rec map2_tr f xs ys acc = match xs, ys with
    | [], [] -> acc
    | _, [] -> raise (Invalid_argument "Len(list1) > len(list2)!")
    | [], _ -> raise (Invalid_argument "Len(list2) > len(list1)!")
    | x :: xt, y :: yt -> map2_tr f xt yt ((f x y) :: acc)
  in
  let reverse xs acc = match xs with
    | [] -> acc
    | x :: xt -> reverse xt (x :: acc)
  in
  reverse (map2_tr f xs ys [])
```

Why is it necessary to reverse the answer? Because the tail recursive version is adding elements to the head of the list so we get a reversed version of the list we want. We can then do a tail recursive reverse as shown in the lecture. This solution is actually equivalent to the module function `List.rev_map2` which is the tail-recursive version of `List.map2`.