CSCI-GA-2110: Programming Languages, Fall 2024

PSet 3 - Written

Aditeya Baral
N19186654

1 Bad If Statements

Task 1.1 Explain what is wrong with this implementation of ifC. In particular, give an example of a small program that leads to an error during evaluation using this style of `ifC`, but which would not cause an error under a correct interpretation.

This is erroneous because the implementation evaluates the `guard` as well as both expressions `e1` and `e2` before performing the branch based on the `guard` condition. This eager evaluation *might* lead to **unexpected behaviour and errors** that could have been avoided if the expressions were evaluated after the branch, especially if any of the expressions depend on the expression carried out in the `guard`.

Consider the following snippet:

```
(ifC (= x 0)
     x
     (/ 1 x))
```

In the above example, we would evaluate the `guard`:= `(= x 0)`, `e1`:= `x` and `e2`:= `(/ 1 x)` first. However, we would encounter a zero division error while evaluating `(/ 1 x)`. This error will occur irrespective of the `guard`'s value, even though the division expression should never have been evaluated, as in a correct implementation. Additionally, there could be other side effects as well. If either branch updates the Store `sto`, it could also **propagate unwanted changes** to other constructs. There is also **unnecessary computation** being performed which could be avoided with short-circuit evaluation.

We can implement this correctly by first evaluating only the `guard` and then based on the outcome, evaluate `e1` **or** `e2`. This would correctly use short-circuit evaluation to only execute the branch that would not lead to errors.

```
(define (eval-env (env : Env) (sto : Store) (e : Expr)) : Result
  (type-case Expr e
    ...
    [ifC (guard e1 e2)
      (let* ([rguard (eval-env env sto guard)])
        (cond
          [(equal? (res-v rguard) (boolV #true))
           (eval-env env (res-s rguard) e1)]
          [(equal? (res-v rguard) (boolV #false))
           (eval-env env (res-s rguard) e2)]
          [else (error 'eval-env "ifC guard was not a boolean")]))]
    ...))
```

2 Boxes

Task 2.1 Give an example of how, on certain inputs, evaluting `myfun` causes it to return 4 instead.

The `set-box!` function modifies the passed two `box` arguments **in-place** or by **reference**. This means that when two `box` arguments are passed to `myfun`, their values are changed to `1` and `2` respectively, and their sum evaluates to `1 + 2 = 3`. This occurs only when two *separate* `box` arguments are provided (eg. `(myfun (box 0) (box 0))` and causes the two `box`s to be **updated independently**.

However, if the ***same*** or ***shared*** `box` were to be passed as arguments to the `myfun` function, then its value would first be set to `1` in the first statement, and then to `2` in the next. In the third statement, since we are referring to the ***same*** `box`, the value extracted for each operand would also be `2` and the sum of which would be `2 + 2 = 4`. Attached below is a code snippet that displays this behaviour.

```
(define shared-box (box 0))
(myfun shared-box shared-box)  ; Returns 4
```

3 Dynamic Scope and Recursion

Task 3.1 What happens when this example is executed with dynamic scope instead? Explain why.

In dynamic scoping, any variable or function declared will be accessible in a dynamic environment comprising bindings from all active function calls. As a result, the binding of a variable is determined by the most recent assignment in the stack rather than where it was lexically defined.

In the above example, the `let` binds the `fact` to the `lambda` function. Once `fact` is called within the `let`, it does find the binding in the dynamic or current execution environment and it can recursively call itself to compute the factorial. Thus, it would return a value of `5!  = 120`.

This is because the dynamic scope allows the recursive occurrence of `fact` to find its binding in the dynamic environment that includes the `let` binding. The scope for finding `fact` includes all active bindings at the time the function executes, not just the bindings available where the function was defined (as would be the case with lexical scope). Thus, `fact` is resolved at runtime based on the current execution context and each recursive call creates a new environment where `fact` is bound to the `lambda` function, and this binding is visible to all expressions executed within that call, including further recursive calls. In contrast with lexical scoping, The `fact` within the `lambda` body would be resolved based on where the `lambda` was defined and since at the point of definition, `fact` is not yet bound, it would lead to an `fact:  undefined` error.