Aflevering 3 - Programmerinssprog

Exercise (67 - a)**.**

Adapt your implementation of function calls from MiniScala v4 to higher-order functions.

• • •

Solution. The code for higher-order functions, is not much different from the last handin. The main difference is that closures are now `closureVals`. We can then create maps between identifiers and these `closureVals`, and define how they should be evaluated in `CallExp`. This allows for higher-order functions, since a higher order function, is essentially just a function that takes a `closureVal` as an argument. In other words, since functions are treated as values, and functions take values as arguments, we can create functions that take functions as arguments.
When evaluating a `CallExp(funexp, args)`, we evaluate the `funexp`. As `funexp` is an identifiere we get some `Val`. We then ensure that this is specifically a `ClosureVal`. We then use the same method from last week, where we check that the arguments are the right types, and then bind them to the parameters. To allow for mutual recursion, we create `closureVals` for any defintions in the functions scope. We then type-check the return value, before evaluating the function.

Exercise (67 - b)**.**

Implement anonymous functions (lambdas)

• • •

Solution. Lambdas are easy! Given our current environment and a `LambdaExp`, we have everything we need to make a `ClosureVal`. Since we dont type-check the return values we just pass `None` as the `optrestype`, and similarly `List.empty` as we dont have mutual recursion for lambdas. `CallExp` doesnt even require any changes! Calling a lambda expression just returns a `ClosureVal`, and assigning the lambda an identifier doesnt change this.

Exercise (68 - a)**.**

Adapt your implementation of type-checking for function calls to higher-order functions. (Hint: you may find the makeFunType function useful.)

• • •

Solution. As functions are now values, their types `FunType` live in the same type environment as the other types. We make use of the `makeFunType` function, to extend the type environment with maps of the identifiers to the their associated FunType. As before, we typecheck the body and the parameters. Since the parameters can be of FunType now, we allow for higher-order functions When we call a function, we fetch its `FunType`, by typechecking the id. We then check that the parameters passed, have the correct types and finally return the return type.

Exercise (68 - b)**.**

Implement type-checking for anonymous functions (lambdas). (Hint: see the video for slide 35.)

When type-checking a lambda we return its `FunType`. We construct this by creating a `List[Type]` and populating it with the parameter types. We then type-check the body, in an environemt including these types. We then combine the parameter types and body type and return this.

Exercise (68 - d)**.**

Make sure that your tests from week 4 still run. Extend your test suite with at least 5 tests that properly test your implementation of higher-order functions. Here is a stub you may use for your tests.