AP2024-Assignment One

By
Wanjing Hu / Computer Science, KU
fng685@alumni.ku.dk

1 Introduction

General concerns are type correct with less redundant code.

Estimation: For test cases, high coverage rate of 84 percent in Alternatives for program coverage total, estimated by cabal.

2 Tasks

2.1 Task: Functions

Lambda is written according to the ValFun return type, and Apply is written by extracting the possible types of eval results. They are functionally correct.

For Apply, we have considered the case of how to calculate when the argument expression is a Lambda, that is its env will be applied and used for the eval of the functional expression. And also the cases when the argument expression is analyzed to be a pale ValInt or ValBool, and they are just Let into the env.

Improvements: Maybe Apply is a little verbose. Because the env1 will be the same as env, but in case of the warning of shadowing we still kept it.

2.2 Task: try-catch

* Pay attention to the order of eval-ing e1 and e2, since the exp could be a Let and modify the env.

* If expression 1 and expression 2 both return Left value, the error message will stress both for better reading.

2.3 Task: Pretty-printer

It is an ordinary recursive function.

3 QA

1. What is your observable evaluation order for Apply, what difference does it make, and which of your test(s) demonstrate this?

The order is: use the var in env first, then the var from arguments.

Take (Apply e1 e2) as an example.First it will evaluation e1 and e2. Since e1 and e2 may contains Let, so after the eval of e1 and e2, the env will change to env1 and env2.

If e2 is an apply-able thing, e1 will be evaluated by using the merged environment of env, env1, env2, and the merging strategy is according to the order given. (e.g. if the order is env, env1, env2, and env2 has a same key as env, my eval will apply env first and ignore the env2 key).

Demonstrating test:

eval [("y",ValInt 10)] (Apply (Lambda "y" (Add (CstInt 0) (Var "y"))) (Let "y" (CstInt 2) (Lambda "y" (Add (Var "x") (Var "y")))))

The result is Right (ValInt 10)

2. Would it be a correct implementation of eval for TryCatch to first call eval on both subexpressions? Why or why not? If not, under which assumptions might it be a correct implementation?

It could be correct when the following is analyzing the first subexpression firstly. But the following may cause errors because of the lazy analyse:

```
case (e1, e2) of
(_, e2) -> ...
(e1, _) -> ...
```

Since the Let in e2 may influence the env in e1, and when we modify a same env, the Unkown Vars in env1 would be a Kown one, and gets a wrong eval.

Example(see Apply(order)):
eval [] (TryCatch (Var "x") (Let "x" (CstInt 1) (CstInt 1)))

If the eval e2 will not influence the env, that is, their env are independent, then it would not be a problem. Also when the extract of case is as following:
case (e1, e2) of
(e1, _) -> ...
(_, e2) -> ...

3. Is it possible for your implementation of eval to go into an infinite loop for some inputs?

There would be a infinite loop if I implement the ‘Apply‘ function is implemented recursively but I avoided it by nailing down the cases when it is calling (Apply Lambda1 Lambda2)

And I defined that when the Lambda2 could never be solved into a ValInt or ValBool or Var, it would leave the Lambda1 unchanged.