Advanced Programming
Home Assignment 5

Carlo Rosso, Chinar Shah

Contents

1 Introduction

The first task was the hardest, because we had to figure out how to change the generator. Probably it was so difficult because we had to understand a new concept: the `Gen` monad, and how to use it. The second task was time consuming, because we had to figure out what was wrong with an implementation we did not write and then we had to fix it. We tried to make the least changes to make everything work, but it was not easy. The third task was overall easy, because we did not have to fix anything. Since we did not fix the code, we feel like we may have missed something, therefore we feel less confident about our answer to the last question. The following describes how to run the tests

1. unzipping `a4-handout.zip`;
2. entering the `a4-handout` directory through the terminal;

3. running `cabal test` in the terminal.

2 A better generator

First of all we figure out which error can be produced by which expression. Because in the next step, we change the expressions distribution to make `expCoverage` work.
As suggested in the assignment, we also modify the generator to keep track of the available variables, indeed we update the generators of the sub-expressions `Let` and `Lambda` to store any new variable in scope. Therefore, we fix `Var` to make it prefer variables in scope.

3 A property for parsing/printing

The implementation of `parsePrinted` is quite straightforward, but it took a lot of time to figure out how to fix some of the bugs. We will discuss them later in the report.

4 A property for checking/evaluating

Also the implementation of `onlyCheckedErrors` is straightforward, we simply follow the instruction provided in the assignment.

5 Questions

Can programs produced by your generator loop infinitely? If so, would it be possible to avoid this?

Yes, it is possible for the expression generator to produce a program which loops infinitely. For example, while it is very unlikely, it is possible that the generator produces the following expression:

```
Apply
    (Lambda "y" (Apply (Var "y") (Var "y")))
    (Lambda "z" (Apply (Var "z") (Var "z")))
```

A way to prevent the generator from producing such expressions is to fix the generator of the `Apply` expression not to allow a `Var` expression to be generated as the function argument of the `Apply` expression.

What counter-examples did `parsePrinted` produce? For each counter-example, which component (implementation, generator or property) did you fix?

1. `Var "-/564afg"`: we fix the generator to make use only of valid `VName`.

2. `CstInt (-4)`: we fix the implementation to parse negative integers and not just the negative sign.
3. `Apply (CstBool True) (Apply (CstBool False) (CstBool True))`: we fix the implementation of `printExp` to add parentheses around the second argument of the `Apply` expression if it is an `Apply` expression.
4. `Lambda "m3b4" (Add (TryCatch (CstInt 17) (Lambda "t7" (CstBool False))) (CstBool True))`: we fix the implementation of `printExp` to add parentheses around the `TryCatch` expression.
5. `Let "if" (CstInt (-79)) (CstInt (-37))`: we fix the generator of the variables name to avoid using reserved keywords.

What is the mistaken assumption in `checkExp`?

One failing expression is `Apply (TryCatch (Lambda "zn" (Var "c7dh")) (CstBool False)) (CstBool True)`. From the reported expression, we can see that the `TryCatch` is applied to a `Lambda`, which will fail when evaluated. The mistaken assumption is that if the evaluation of the first argument of `TryCatch` does not fail, then such expression can be considered as checked and its evaluation will not fail. While, the lambda must be applied to an argument to actually be evaluated. Therefore if the first argument of `TryCatch` is a `Lambda`, `checkExp` should list also the errors which can be produced by the body of the `Lambda`.
In second place, we note that sometimes the testing of `onlyCheckedErrors` doesn't end, which happens because the generator produces a program which does not terminate.