Assignment 5

Advanced Programming
Department of Computer Science
University of Copenhagen

Magnus/mfh144 | Mathilde/qfh987

13/10/2024

Contents

1. Introduction

In this week we used auto generated tests by *QuickCheck* to test for errors in our implementation and generators. We also modified the frequency of generated expressions, to follow some conditions. In particular we made and used the properties *parsePrinted* and *onlyCheckedErrors* to find counter-examples with *QuickCheck* and improve our code.

The tests can be run from the a5-handout directory with the following command:

```
a5-handout $ cabal test
```

2. Task: A Better Generator

We believe that our implementation works as expected, and it passes all tests.
In this task we changed the distribution of generated expressions. We did this using *frequency* instead of *oneof*. We also changed *genExp* to take a list of variable names as input. This list holds all variable names that are currently in scope, so it is extended whenever a *Lambda* or *Let* expression is generated. To get the right amount of domain errors, we added a generator to the list that only generates expressions with domain errors, and we also limited the amount of generated binop expressions by generating only one of these with nearly the same frequency as the rest of the expressions. This ensures that we have a more constant amount of domain errors.
We also made sure that all variables generated has length between 2 and 4 and are well formed, meaning that they consist of an initial alphabetic character, followed by one or more alphanumeric characters. Lastly we only generate variables that are out of scope in 1 out of 6 times where a *Var* expression is generated.

3. Task: A Property for Parsing/Printing

We believe that our implementation works as expected, and it passes all tests.
In this task we tested the property that an expression that is printed and then re-parsed, should result in the original expression. This was initially not the case. To fix the problems that arose, we changed the parsing of integers to include an optional sign in front of the digits, such that we can parse negative numbers. We also change the printer to add parenthesis around *Apply* and *TryCatch* expressions, such that expressions are grouped correctly for the parser. Printing of *Let* and *Lambda* already added the parentheses. Lastly we ensured that variable generation could never result in a keyword.

4. Task: A Property for Checking/Evaluating

We believe that the property we have defined is correct, since we see very specific failing test cases.
In this task we found that the *checkExp* and *runEval* functions do not return errors that match, that is, an error resulting from evaluation of an expression is not always present in the *checkExp* output. We found that the expressions that fails are always some combination of *Apply*, *TryCatch* and *Lambda*. An example is: `Apply (TryCatch (Lambda "test" (Pow (CstInt 1) (CstInt (-1)))) (CstBool True)) (CstBool False)`.
The reflection on why this is happening will be presented in he questions later.

5. Questions

1. Yes, the generator can generate expression that would infinitely loop. An example could be:
   `Let "gx" (Lambda "fx" (Apply (Var "fx") (Var "fx")) (Apply (Var "gx") (Var "gx"))`
   We could avoid this infinite loop by constraining the generator to never apply something to itself.
2. The first counter example was: `CstInt -1`. We fixed the implementation of *lInteger* in *Parser.hs* to add an optional sign.

The second counter example was:
`Pow (TryCatch (CstInt 0) (CstInt 5)) (CstInt -1)`
This example was due to the printer not adding sufficient parentheses for the parser, so we had to fix the implementation component

The third example was:
`Let "if" (CstInt 3) (CstBool True)`
This example was due to the generator generating invalid variable names. We fixed the generator component so *genVar* retries if the generated var is keyword.

3. The mistaken assumption in *checkExp* was to mask errors in some cases of a *TryCatch*. This shadowed errors when *Apply*, *TryCatch* and *Lambda* were used in combination. In particular when *TryCatch* has a *Lambda* function in the try-portion. A *Lambda* can not in itself be erroneous and throws no error, but once attempted used as a function in *Apply* it can produce an error.

   In conclusion, *checkExp* assumes that if the try expression has any nested subexpressions that are erroneous, then the error will be propagated upon evaluation, which is not the case with Lambda expressions.