Home Assignment 5
Advanced Programming

qhz731

October 12, 2024

1 Introduction

The main goal of this assignment is to perform property-based testing on the components from the previous assignments and fix corresponding bugs that emerged during property-based testing, the components include the printer, parser, evaluator, and type checker. I implemented an improved expression generator to generate more evenly distributed expressions, property tests for the parser and printer, and property tests for the type checker and evaluator, the bugs in printer and expression generator are fixed.

Overall, I think my solution is functionally correct. To run the tests, simply execute the `cabal test --test-show-details=always` command in the project root directory. This will run all implemented property tests and display the test results and coverage information.

2 Task: A better generator

My solution is basically functional and successfully improved the expression generator to meet the specified distribution conditions. I changed the signature of `genExp` from `Int -> Gen Exp` to `Int -> [VName] -> Gen Exp` to track variables in scope. I used `frequency` instead of `oneof` and adjusted the generation frequency of various expression types to better control the generated expression distribution to achieve the desired distribution. I implemented helper functions `genNewVar` and `genVarOfLength` functions to generate variable names from 2 to 4 characters long, additionally, the variable name generator will avoid generating reserved words.

In `Let` and `Lambda` expressions, the `VName` is chosen from a list which contains both newly generated variable names and known variable names, which increases the possibility of variable errors. The solution can be tested by the `expCoverage` property.

3 Task: A property for parsing/printing

In this task, I implemented the `parsePrinted` property, which checks whether the result of printing an expression and then parsing it is the same as the original expression. In the initial test, `QuickCheck` found some counterexamples. The first was that the

parser could not recognize negative numbers. `APL`'s `CstInt` must be non-negative, but the generator may generate negative numbers. I modified the generator to use the `abs` function when generating `CstInt` to ensure that only non-negative numbers are generated. The second is that the print function lacks the necessary parentheses in some cases, resulting in parsing ambiguity. I modified the function to add parentheses in places such as `Apply` and `TryCatch`. After making these modifications, the property `parsePrinted` finally passed 10,000 tests.

4 Task: A property for checking/evaluating

In this task, I implemented the `onlyCheckedErrors` property to check whether the errors generated when evaluating the expression are in the error list returned by the `checkExp` function. The `onlyCheckedErrors` function first uses `checkExp` to get a list of possible errors for the expression, and then actually evaluates the expression. If the evaluation succeeds, the function returns True; if the evaluation fails with an error, the function checks if the error is in the list of errors returned by `checkExp`.

5 Q&A

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

Generators may generate programs that cause infinite loops. Since generators can create expressions that contain function definitions and function applications, if these functions recursively call themselves without proper termination conditions, it may cause infinite loops. The possibility of generating infinite loop programs can be reduced by adding some restrictions to the generator, but it cannot be completely avoided. For example, when generating expressions, track the depth of function calls and set a maximum depth. Avoid direct or indirect self-calls. When generating recursive functions, make sure that the function contains the correct termination conditions.

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

- `CstInt (-1)`: The `CstInt` must be non-negative, so I changed the generator to avoid negative numbers in `CstInt`.
- `Apply (CstInt 5) (Apply (CstInt 6) (CstBool False))`: The result of printing this expression is "5 6 false", the result of parsing "5 6 false" is `Right (Apply (Apply (CstInt 5) (CstInt 6)) (CstBool False))`. This is caused by the printer, the printer did not add parentheses for `Apply` and `TryCatch` and produced ambiguity. So I changed the implementation of printer to add parentheses in correct place.

- `Let 'if' (CstInt 5) (CstInt 7)`: The generated `VName` is a reserved keyword. I modified the generator, if the generated `VName` is a reserved word, the string will be reversed.

5.3 What is the mistaken assumption in checkExp?

In the `checkExp` function, the incorrect assumption is that it assumes that all errors that occur in the first argument of the `TryCatch` expression can be masked and ignored during static analysis. This is implemented in the check function as follows:

```
check (TryCatch x y) = do
  maskErrors $ check x
  check y
```

Here, `maskErrors` is a function that suppresses any errors that occur in its argument. The basis for this assumption is that `TryCatch` catches errors at runtime, so it is safe to ignore any errors in the first augment during static check.

However, this assumption is incorrect because static errors (such as type errors or unbound variable errors) cannot be caught by `TryCatch` at runtime. These errors must be detected and reported during static analysis to ensure that the program is typed correctly and has no unbounded variables. By masking these errors, `checkExp` fails to report some problems that cannot be handled dynamically at run-time.

For example, consider the following expression:

```
Apply (TryCatch (Lambda "wh" (If (CstInt 7) (CstBool False)
  (CstInt 12))) (CstInt 73)) (CstInt 6)
```

In static analysis, the condition of the `If` expression inside the `Lambda` is `CstInt 7`, which is an integer rather than a Boolean value and should produce a `NonBoolean` error. However, since `checkExp` masked the error inside `TryCatch`, it fails to report this error. As a result, only the `NonFunction` error from the `Apply` expression is reported.

I tried to fix this bug by removing `maskErrors`, then the property `onlyCheckedErrors` could pass 100,000 tests.