CS421:Programming Languages and Compilers **Fall 2017**

Unit Project Report

Xinyu Zhou, NetID: xinyuz4 *Handed In: February 20, 2018*

1 Introduction

This project aims to implement an auto grader for α-equivalence transformation. In a real problem, a pair of lambda calculus expression that are α-equivalent are given. Student are expected to use a series of one-step induction to prove that these two expression are α-equivalent. The auto grader aims to judge the correctness of each step. For example, given $(\lambda x.(\lambda y.y)x)y \sim \alpha \sim (\lambda x.(\lambda z.z)x)y$ as conclusion, student can answer rule as "Application Rule (left)", and write down hypothesis $(\lambda x.(\lambda y.y)x) \sim (\lambda x.(\lambda z.z)x)$.

Given a pair of expressions, some rules can be adapted to decompose the expression (called "conclusion") into sub-expressions (called "hypothesis"). If each pair of sub expression are α-equivalent or exactly the same, the pair of main expression is α-equivalent.

There are five rules:

1. Terminate: two expressions are exactly the same.
2. Abstraction: $\lambda x.e_1 \sim \lambda x.e_2 \rightarrow e_1 \sim e_2$
3. Application: $e_1\ e_2 \sim e_3\ e_4 \rightarrow e_1 \sim e_3, e_2 \sim e_4$
4. Transition: $e_1 \sim e_3 \rightarrow e_1 \sim e_2, e_2 \sim e_3$
5. α-conversion: $\lambda x.exp \overset{\alpha}{\rightarrow} \lambda y.(exp[y/x])$. This rule is treated as side condition.

In each step, one conclusion will be given, and student should choose a proper rule and type in the hypothesis that get the conclusion through the rule. The auto grader then check the rule and hypothesis to judge the correctness of the step. To achieve this, a right way to denote the lambda calculus and a method to judge α-conversion are both necessary.

In conclusion, the project should design a lexer and parser to parse the input, a data type for lambda calculus, and a set of rules to make sure each step is correct. The project is programmed by OCaml, and some code comes from or inspired by the lecturer's code.

2 Function Achieved

2.1 Inputs and Outputs

The input type is OneStepInput (string * string * string list), where the first string is conclusion, the second string is rule, and the string list is the list of hypothesis strings. Then a function legal_one_step (fun onestep_input $\rightarrow$ error) should be called to get the output. Each conclusion and hypothesis should be a legal lambda calculus expression pair: the string should have two legal lambda calculus expressions connected with "$\sim a \sim$", and the character "λ" in the expression should be denoted by "%".

The output is an error type, which catches most of the possible errors that can be make in the one-step induction. It also contains "NoError", which means the induction is correct. User can also call print_error error to get the readable information.

2.2 Modules and Useful Functions

Module Name	Usage
Lambda_lex	Lexer
Lambda_parse	Parser, output data type of lambda calculus
Lambda	Get binding relationship of lambda calculus
Match_rule	Check the correctness of rule and hypothesis
Unit_test	Do some auto test from txt file
Lambda_test	Interactive test environment

Table 1: Modules

Function Name	Module Name	Usage
get_binding_relation	Lambda	Get binding relationship of lambda calculus
parse_exp	Match_rule	parse input string into expression pair
is_alpha_conversion	Match_rule	Check α-conversion
legal_onestep	Match_rule	Main function for one step auto checker
print_error	Match_rule	Print the readable information of one step result

Table 2: Useful Functions

3 Data Types

3.1 Lambda Calculus and α-Equivalence

To denote the lambda calculus expression, a nature thought is simulate the definition of it:

type lam =
| VarLam of string (* x *)
| AbsLam of string * lam (* λ x.e *)
| AppLam of lam * lam (* e_1 e_2 *)

where string denote variables. The parser in my project comes from the exist lambda calculus auto grader. For example, a lambda calculus $(\lambda x.(\lambda y.y)x)y$ will be parsed into

$$AppLam(AbsLam(x,(AppLam(AbsLam(y,VarLam(y)),x)),y)$$

To denote two lambda calculus expression are α-equivalent, a mark "$\sim \alpha \sim$" needed to be added between two expressions. As a result, the final express of an α-equivalent relationship will be denoted as a pair of expressions.

3.2 Binding Relationship, Mapping and Environment

To judge two lambda calculus expression are α-equivalent or not, and to judge whether two expression can be transformed by an α-conversion step, we need to know the binding relationship of a lambda calculus. Here I introduces the contents of the environment, how to update and use it will be included in section 4.

To distinguish the three situation: free, binding and bounded of variables, a data type binding_relation is in charge:

type binding_relation =

| Bind of int * int list * string (* (index, bounded var index, name) *)

| Free of int * string (* index, name *)

A full binding relation list of a lambda calculus is a list of binding relations. Each variable in the lambda calculus has an index, i.e. its order. If a variable is free, its index and name are directly recorded as Free. If a variable follows λ, i.e. as a binding variable, then it will have a new Bind item in the list, with an empty bounded list and its name recorded. If a variable is bounded by a binding variable, it will be added in the bounded list of the binding variable. As an example, $(\lambda x.(\lambda y.y)x)y$ has a binding relationship [Bind(1, [4], x), Bind(2, [3], y), Free(5, y)]

What always happens is a variable name be reused to define a different variable. For example, (λ x. (λ x. x) x) has two group of x's. To record the relationship between name and identifier of variables, a map is needed. A map is a list of (string * int) pairs, saving the name of a variable and the index of it, respectively.

The last thing in the environment is a counter simply count the number of variables before getting into the expression, thus the index of first variable in the expression. It is useful when parsing recursively.

3.3 Rules and Errors

Rules are errors are simple data types, one sub type for each situation. Thus they can be easily expanded for new inputs and outputs. As directly inputs and outputs respectively, both rules and errors have functions to convert string to them and vice versa.

4 Algorithms

4.1 Environment

Given a lambda calculus expression denoted by types introduced in section 3.1, the environment variable is updated recursively.

First we should know how to mark identical variables. Say we have $(\lambda x.(\lambda y.y)x)y$, it is actually ($\lambda$ VAR1. (λ VAR2. VAR2) VAR1) VAR3. To simplify it, we can use integer k_i to denote to VARi, as far as for any i $\neq$ j, $k_i \neq k_j$. Thus, we can use the index of the variable in the lambda calculus as the identifier of the variables where the variable first appears. By searching a name in a map list, if a name first appears, the name denotes a new identifier, so a (name, index) pair can be added into the map. If a name has already in the map, we can either find the identifier of the name, or update the name to a new identifier. For simplification, I use a stack rather than list for the map, which also let searches get the newest map pair. The map stack of $(\lambda x.(\lambda y.y)x)y$ is [(y, 5), (y, 2) ,(x, 1)].

At the beginning, environment start with counter as 0, empty map list and empty binding relationship list. then scan the outer layer of the lambda calculus variable.

1. AbsLam: first increment counter, then update the map with the binding variable, and add new Bind(counter, []) to binding relationship list, and finally apply to the bounded part recursively.

2. AppLam: simply apply to the two part recursively.

3. VarLam, i.e. a single variable, then first let counter increment by one, then search the map. If the name not exists, simply add a new Free(counter, name) to the binding list. If the name exists with identifier x, then search the binding list and bound it to the binding variable with identifier x (i.e. append to the list in Bind(x, [idx list])). If the identifier is a free variable, just append Free(counter, name) to binding list.

After apply recursively, an environment will be returned. Although it return the changed counter and binding relationship list, it should return the untouched map rather than the changed map, because the map in a layer should not be influenced by deeper layers. Finally, in the base layer, only the binding relationship list is passed out, which will be used to judge α-conversion.

4.2 α-Conversion Judgment

An α-conversion step is a step of renaming variables. In an α-conversion, the name of one binding variable changes, and the name of all the variable bounded on it are also changed. Variable with the same name but not bounded together should not changed, and only one binding-bounded relationship can change their name in one step. For example, $(\lambda x.(\lambda y.y)x)y$ can become $(\lambda x.(\lambda z.z)x)y$ after α-conversion.

To judge an α-conversion step is legal or not, two things need to be check: (1) all changes happens in one binding-bounded relationship, (2) name of all the variable in the relationship are changed to the same name. It is simple to judge α-conversion by using binding relationship list introduced above.

For two expressions, first get the binding relationship of them, then compare the items in two lists pair by pair. There can be three situations: (1) Two item are exactly the same, that means variables in the item are not changed at all. (2) Two items are both Bind(int, int list, string), and have only difference in the string part (i.e. name of the variable), which means this variable changes its name correctly. (3) Other conditions, including list length not same, binding / bounded not match, Bind vs Free, etc., all of which means two expression have different binding relationships. After comparison, if one and only one pair are in situation (2) and all other pairs are in situation (1), then we know there's exactly one correct renaming happens, thus a legal α-conversion step. The pseudo code is below:

```
function is_alpha_conversion(exp1, exp2) {
  bl1 = get_bind_relation_list(exp1);
  bl2 = get_bind_relation_list(exp2);
  if bl1.size() != bl2.size() return False;
  count = 0;
  for(i in bl1.size()) {
    item1 = bl1[i], item2 = bl2[i];
    if (item1 == item2) {;}
    else if (item1 and item2 are both Bind,
             and item1 and item2 only different on name) {
             count ++;
             }
    else {return False;}
  }
  return count == 1;
}
```

4.3 Rules Judgment

For each one-step transformation, student should state the rule, and give the hypothesis related to the rule. Part of the hypothesis might be a change of expressions in the conclusion, and part might be rewrite (i.e. copy) of expressions.

To make sure a rule and the related hypothesis are correct, three things need to be check: (1) The rule is applicable for the conclusion, (2) The changed part are changed correctly, (3) The rewrite part are rewrite correctly.

("$\rightarrow$" stands for α-conversion.)

Rule	Formula	Condition
Terminate	$\frac{}{e_1 \sim e_2}$	$e_1 \equiv e_2$
Left Conversion	$\frac{e_3 \sim e_2}{e_1 \sim e_2}$	$e_1 \rightarrow e_3$
Right Conversion	$\frac{e_1 \sim e_3}{e_1 \sim e_2}$	$e_2 \rightarrow e_3$
Abstraction	$\frac{e_1 \sim e_2}{\lambda x.e_1 \sim \lambda y.e_2}$	$x \equiv y$
Application	$\frac{e_1 \sim e_3; e_2 \sim e_4}{e_1\ e_2 \sim e_3\ e_4}$	
Left Application	$\frac{e_1 \sim e_3}{e_1\ e_2 \sim e_3\ e_4}$	$e_2 \equiv e_4$
Right Application	$\frac{e_2 \sim e_4}{e_1\ e_2 \sim e_3\ e_4}$	$e_1 \equiv e_3$
Transition	$\frac{e_1 \sim e_2; e_2 \sim e_3}{e_1 \sim e_3}$	

Table 3: Rules

Table 3 shows all the rules supported. In each rule, e_i indicates for identical expressions.

Formula includes conclusion below line and hypothesis above line, and condition aside. Take Left conversion as example, for any conclusion $e_1 \sim e_2$, if applying Left Conversion rule, three things need to be check: (1) the two expressions in conclusion fits the type in the formula (here always fits), (2) in hypothesis, the second expression is exactly the same with the second term in conclusion, and (3) the first expression in conclusion and in hypothesis are α-conversion to each other.

There are different errors that can be made during the rules. Most of them are caused by dissatisfaction of the three conditions. By check the conditions it is easy to raise related error there.

5 Conclusion

In the project, I have implemented a one-step α-equivalence auto judgment algorithm. In specific, the program can accept one conclusion, one rule and a list of hypothesis, and decide whether the rule and the hypothesis are good for the conclusion, and give appropriate hint of the mistake. There is a interactive test environment and a unit test tool for easy understanding and for extension. From this project, I have had better understanding of OCaml, lambda calculus and compiling. It is also a challenge to write detailed report and documentation, for a project that ease understanding and further implementation, by myself and anyone who will deal with it.

Reference

1. CS 421: Programming Languages and Compilers, Lecture Slides

   `https://courses.engr.illinois.edu/cs421/fa2017/CS421D/lectures/25-26-lambda.pdf`

2. Wikipidea: Lambda Calculus

   `https://en.wikipedia.org/wiki/Lambda_calculus`

3. OCaml Tutorial

   `https://ocaml.org/learn/tutorials/index.html`