Project 4

Jinhao Wei

Feb 9, 2019

Abstract

This project is mainly a practice on programming in HOL. In this project, we attempted higher order logics in HOL. We provided solutions for several problems, including problems on:

1. HOL expressions that are equivalent to ACL semantics or standard notations.
2. HOL functions that operates on HOL terms.

For each of the problems, this report contains:

- Problem Statements
- Relevant Code
- Execution Transcripts of Test Cases

Acknowledgments: This project followed Professor Shiu-Kai Chin's sample documents' format, and used multiple *.sty* files, such as *634format.sty*, *holtex.sty*, *holtexbasic.sty*, which were provided in Professor Shiu-Kai Chin's sample report. What's more, this report inherits *P2_jwei15.tex*, a file that I had created while working on Project2, so I can save some time on formatting.

Contents

Chapter 1

Executive Summary

All requirements for this project are satisfied Specifically,

Report Contents
This report consists of:

Reproducibility in ML and LaTeX
All ML and LaTeX source files compile well on the environment provided by this course.

Chapter 2

Exercise 6.2.1

2.1 Problem Statement

Our basic task in this problem is to write HOL codes that are equivalent to ACL semantics or standard notations. We will also print and observe the result of each piece of code.

There are no extra test cases required in this exercise problem.

2.2 Relevant Code

We will define each of our functions as below:

```
(*  6.2.1.1  *)
``P x ==> Q y``;

(*  6.2.1.2  *)
``P (x:num) ==> Q (y:bool)``;
``P x ==> Q y``;

(*  6.2.1.3  *)
``!x y.P x ==> Q y``;

(*  6.2.1.4  *)
``?(x:num).R(x:'a)``;

(*  6.2.1.5  *)
``~(!x.(P x) \/ (Q x)) = ?x.(~(P x)) /\ (~(Q x))``;

(*  6.2.1.6  *)
``!x.(P x ==> M x)``;

(*  6.2.1.7  *)
``?x.P x /\ Funny x``;
```

2.3 Execution Transcripts of Test Cases

Exercise 6.2.1 did not provide extra test cases, so we simply send each piece of the relevant codes to the HOL, and we will see results as below:

2.3.1 Question 6.2.1.1

In this question, we will write HOL codes that are equivalent to semantic $P(x) \supset Q(y)$.

```
(*  6.2.1.1  *)
``P x ==> Q y``;
```

The code above will give us the execution transcript as below:

```
 > # <<HOL message: inventing new type variable names: 'a, 'b>>                     1
val it =
   ``(P :'a -> bool) (x :'a) ==> (Q :'b -> bool) (y :'b)``:
   term
```

2.3.2 Question 6.2.1.2

In this question, we will constrain the data types in the previous question. Specifically, we will constrain x to *num* type and y to *bool* type. Then we will compare the running result of this "restrained" statement and last exercise's "default" statement.

```
(*   6.2.1.2   *)
``P (x:num) ==> Q (y:bool)``;
``P x ==> Q y``;
```

From the transcript below we could see that in the first evaluation, x is of type *num* and y is of type *bool*. Function P is a mapping from type *num* to *bool* and Q is a mapping from type *bool* to type *bool*. In the second evaluation, x is of type $'a$ and y is of type $'b$. Function P is a mapping from type $'a$ to type *bool* and Q is a mapping from type $'b$ to type *bool*.

```
> # val it =                                                                        2
   ``(P :num -> bool) (x :num) ==> (Q :bool -> bool) (y :bool)``:
   term
> <<HOL message: inventing new type variable names: 'a, 'b>>
val it =
   ``(P :'a -> bool) (x :'a) ==> (Q :'b -> bool) (y :'b)``:
   term
```

2.3.3 Question 6.2.1.3

In this question, we will attempt to write codes that are equivalent to a "forall" statement $\forall x, y.P(x) \supset Q(y)$.

```
(*   6.2.1.3   *)
``!x y.P x ==> Q y``;
```

If we send the above region to HOL window, we will see transcript as below:

```
> > # <<HOL message: inventing new type variable names: 'a, 'b>>                    3
val it =
   ``!(x :'a) (y :'b). (P :'a -> bool) x ==> (Q :'b -> bool) y``:
   term
```

2.3.4 Question 6.2.1.4

In this question, we will attempt to constrain x to type *num* but call it as a data of type $'a$.

```
(*   6.2.1.4   *)
``?(x:num).R(x:'a)``;
```

If we send the above region to HOL, we will see transcript as below(line numbers and character numbers may change, due to different HOL windows). This is because we had previously constrained x to type *num* but later re-constrained it to type *'a*. Therefore the type inference fails.

```
> #                                                                           4
Type inference failure: the term

(x :num)

on line 12, characters 13-17

can not be constrained to be of type

:'a

unification failure message: ???
Exception-
   HOL_ERR
     {message =
      "on line 12, characters 13-17:\n\nType inference failure: the term\n\n(x :num)\n\non line 12, characters 13-17\n
\ncan not be constrained to be of type\n\n:'a\n\nunification failure message: ???\n",
      origin_function = "type-analysis", origin_structure = "Preterm"} raised
```

2.3.5 Question 6.2.1.5

In this question, we will attempt to implement a HOL equivalent statement. We will use HOL to show that "not all x satisfies either $P(x)$ or $Q(x)$" is equivalent to "there exists some x, that does not satisfy either $P(x)$ nor $Q(x)$".

```
(*  6.2.1.5  *)
``~(!x.(P x) \/ (Q x)) = ?x.(~(P x)) /\ (~(Q x))``;
```

The line above will give us the result like:

```
> # <<HOL message: inventing new type variable names: 'a>>                    5
val it =
   ``~(!(x :'a). (P :'a -> bool) x \/ (Q :'a -> bool) x) <=>
  ?(x :'a). ~P x /\ ~Q x``:
   term
```

2.3.6 Question 6.2.1.6

In this question, we will attempt to implement a HOL statement that is equivalent to "all people are mortal". We can interprete the statement this way: for all creature, if "it is a human", then "it is mortal". So we have the code below:

```
(*  6.2.1.6  *)
``!x.(P x ==> M x)``;
```

and the execution result is like:

```
> # # <<HOL message: inventing new type variable names: 'a>>                  6
val it =
   ``!(x :'a). (P :'a -> bool) x ==> (M :'a -> bool) x``:
   term
```

2.3.7 Question 6.2.1.7

In this question, we will use HOL to represent a statement "some people are funny". I thought this is the equivalent meaning as "there exists some x, someone who is a people and someone who is funny".

```
(*  6.2.1.7  *)
``?x.P x /\ Funny x``;
```

By putting the above region to HOL, we shall see the following result:

```
> <<HOL message: inventing new type variable names: 'a>>                      1
val it =
   ``?(x :'a). (P :'a -> bool) x /\ (Funny :'a -> bool) x``:
   term
```

Chapter 3

Exercise 7.3.1

3.1 Problem Statement

In this problem, we will implement a HOL function *andImp2Imp term*, which takes in terms of form $p \wedge q \supset r$ and returns $p \supset q \supset r$.

3.2 Relevant Code

```
(* Takes in ``p/\q==>r`` and returns ``p==>q==>r``*)
fun andImp2Imp term =
    let
        val (tmp1, r) = dest_imp term
        val (p, q) = dest_conj tmp1
    in
        mk_imp(p, mk_imp(q,r))
end;


andImp2Imp ``p/\q==>r``;
```

3.3 Execution Transcripts of Test Cases

If we send the above region to HOL, we will see result as below:

```
> # # # # # # # val andImp2Imp = fn: term -> term                          1
> > > val it =
   ``(p :bool) ==> (q :bool) ==> (r :bool)``:
   term
>
```

Chapter 4

Exercise 7.3.2

4.1 Problem Statement

In this problem, we will create a function named *impImpAnd term*, which takes in terms of form $p \supset q \supset r$ and returns $p \land q \supset r$. Apart of that, we will also show that this function is just a reverse of the previous function defined in Exercise 7.3.1.

4.2 Relevant Code

```
(* Takes in ``p==>q==>r`` and returns ``p/\q==>r``*)

fun impImpAnd term =
    let
        val (p, tmp) = dest_imp term
        val (q, r) = dest_imp tmp
    in
        mk_imp(mk_conj(p,q),r)
end;


(* Takes in ``p/\q==>r`` and returns ``p==>q==>r``*)
fun andImp2Imp term =
    let
        val (tmp1, r) = dest_imp term
        val (p, q) = dest_conj tmp1
    in
        mk_imp(p, mk_imp(q,r))
end;


andImp2Imp ``p/\q==>r``;

impImpAnd(andImp2Imp ``(p/\q)==>r``);
andImp2Imp(impImpAnd ``p==>q==>r``);
```

4.3 Execution Transcripts of Test Cases

From the above code, we will see transcripts as below. The first *val it* is the result of test case $p \supset q \supset r$. The second and the third *val it* shows that *impImpAnd* and *andImp2Imp* are each others' reverse.

1

```
> val impImpAnd = fn: term -> term
val andImp2Imp = fn: term -> term
val it =
   ``(p :bool) ==> (q :bool) ==> (r :bool)``:
   term
val it =
   ``(p :bool) /\ (q :bool) ==> (r :bool)``:
   term
val it =
   ``(p :bool) ==> (q :bool) ==> (r :bool)``:
   term
val it = (): unit
>
*** Emacs/HOL command completed ***
```

Chapter 5

Exercise 7.3.3

5.1 Problem Statement

In this problem, we will implement a HOL function *andImp2Imp term*, which takes in terms of form $\neg\exists x.P(x)$ and returns $\forall x.\neg P(x)$.

5.2 Relevant Code

```
(*takes in ~?x.P x and returns !x.~(P x)*)

fun notExists term =
    let
        val (x, y)= dest_exists(dest_neg(term))
    in
        mk_forall(x, mk_neg y)
end;

notExists ``~?z.Q z``;
```

5.3 Execution Transcripts of Test Cases

If we send the above region to HOL, we will see result as below:

```
> # # # # # # # val notExists = fn: term -> term                                    1
> > <<HOL message: inventing new type variable names: 'a>>
val it =
   ``!(z :'a). ~(Q :'a -> bool) z``:
   term
>
```

Appendix A

Source Code for Exercise 6.2.1

The following codes are from *ex-6-2-1.sml*

```
(*   6.2.1.1   *)
``P x ==> Q y``;

(*   6.2.1.2   *)
``P (x:num) ==> Q (y:bool)``;
``P x ==> Q y``;

(*   6.2.1.3   *)
``!x y.P x ==> Q y``;

(*   6.2.1.4   *)
``?(x:num).R(x:'a)``;

(*   6.2.1.5   *)
``~(!x.(P x) \/ (Q x)) = ?x.(~(P x)) /\ (~(Q x))``;

(*   6.2.1.6   *)
``!x.(P x ==> M x)``;

(*   6.2.1.7   *)
``?x.P x /\ Funny x``;
```

Appendix B

Source Code for Exercise 7.3.1

The following code is from *ex-7-3-1.sml*

```
(*Takes in ``p/\q==>r`` and returns ``p==>q==>r``*)
fun andImp2Imp term =
    let
        val (tmp1, r) = dest_imp term
        val (p, q) = dest_conj tmp1
    in
        mk_imp(p, mk_imp(q,r))
end;


andImp2Imp ``p/\q==>r``;
```

Appendix C

Source Code for Exercise 7.3.2

The following code is from *ex-7-3-2.sml*

```
(*Takes in ``p==>q==>r`` and returns ``p/\q==>r``*)

fun impImpAnd term =
    let
        val (p, tmp) = dest_imp term
        val (q, r) = dest_imp tmp
    in
        mk_imp(mk_conj(p,q),r)
end;


(*Takes in ``p/\q==>r`` and returns ``p==>q==>r``*)
fun andImp2Imp term =
    let
        val (tmp1, r) = dest_imp term
        val (p, q) = dest_conj tmp1
    in
        mk_imp(p, mk_imp(q,r))
end;


andImp2Imp ``p/\q==>r``;

impImpAnd(andImp2Imp ``(p/\q)==>r``);
andImp2Imp(impImpAnd ``p==>q==>r``);
```

Appendix D

Source Code for Exercise 7.3.3

The following code is from *ex-7-3-3.sml*

```
(*takes in ~?x.P x and returns !x.~(P x)*)

fun notExists term =
    let
        val (x, y)= dest_exists(dest_neg(term))
    in
        mk_forall(x, mk_neg y)
end;

notExists ``~?z.Q z``;
```