Daxuan Shu

Report

Part A

My approach to this task is to analyse the two input arguments' formats and figure out a few base conditions to get a more simple task. To illustrate, I use a few base conditions to narrow the input expressions to be lists with equal length. The base conditions are:

1. If (equal? x y) is true, that is to say the x and y are structural equivalence, the returned difference summary can be simply either x or y.

```
e.g.
(expr-compare 12 12)  ⇒  12
(expr-compare #t #t)  ⇒  #t
(expr-compare #f #f)  ⇒  #f
(expr-compare '(cons a b) '(cons a b))  ⇒  (cons a b)
```

2. If x and y are not structural equivalence and if at least one of x and y is not a list, then the difference summary can be the format (if % x y) for example:

```
(expr-compare 12 20)  ⇒  (if % 12 20)
(expr-compare 'a '(cons a b))  ⇒  (if % a (cons a b))
```

3. If x and y are not the cases in the former conditions 1&2, then this means x and y are both lists and they are not structural equivalence. Thus, If list x and list y have different length, the diffference summary should be (if % x y) , for instance:

```
(expr-compare '(list) '(list a))  ⇒  (if % (list) (list a))
```

4. If x and y have different keywords at the beginning, then the returned expression should be the format ((if % x y) conding I am using recursion to compare each con cells, e.g.

```
(expr-compare '(cons a b) '(list a b))  ⇒  ((if % cons list) a b)
```

From the above 4 base conditions, I solved the cases when the two input expressions are not lists with equal length. Now, I will use recursion to parse the two (equal length) input lists x and y from the beginning arguemnt to the end regardless the bound variables like X!Y to easier the problem. (I will deal with the binding variables at the end). Before I start to write the recursion functions, I found another base condition regarding the keyword ----quote or `'`as shown:

5. If x and y are passed the above 4 conditions, then x and y should be both lists and have the same length. If there is a keyword quote or `'` in the beginning of any of x and y, then the returned summay should have the format (if % x y). e.g.

```
(expr-compare '(quote (a b)) '(quote (a c)))  ⇒  (if % '(a b) '(a c))
(expr-compare ''(let ((a c)) a) ''(let ((b d)) b))
```

Thus, I defined a function called ***prefinished-summary-expr*** to deal with the above base conditions without thinking about the bound variables. Shown as Figure 1. After testing with the test cases. This function perfectly deal with the cases without bound variables.

Part B

Now, I am going to deal with the bound variables. I am going to divide the task into two sub problems. The first part is get bound variables in let's bindings and lambda's fromals.
The second part is to sustitude the (if % X Y) for X!Y.

PART I. Getting bound variables in the expressions and store the bound variables pair in a list:
For example:
One of the sample test case with bindings is :

```
(expr-compare '(let ((a c)) a) '(let ((b d)) b))
  ⇒ (let ((a!b (if % c d))) a!b)
```

Here, the bound variabe in x is a and the bound variable in y is b. They are in the same place in the expressions. Thus, the spec requires us to use a!b to represent the bound variable in the summary expression. Now, I want to write a recursive function to get the bound variables into a list. In this example, the returned list should be `'((a b))`.

Function's format:

```
(define (get-let-bindings l1 l2 b)
```

Here, l1 and l2 are the bindings after the let keywords. For example, in the above sample test case, l1 is ((a c)) and l2 is ((b d)). And we want to store the final bound variables into the list b

Base conditions:
1. If at least one of the bindings is empty, then there is no new bound variables and we just pass through the original bound variable list b.

```
61        ;;base condition 1: at least of the x and y's bindings is empty, then simply pass through binding list b
62        ((or (empty? l1) (empty? l2)) b)
```

2. If the bindings' length are different, then it is impossible to have bound varibles and we pass the original bound variable list b.

```
63            ;;base condition 2: The length of the bindings
64            ((not (equal? (length l1) (length l2)) ) b)
```

3. If at least one of the bindings beginning with a keyword, say if, or the bindings part is a list. Since they are not variables and can not be the new bound variables. We search for a deeper level to look for the potential bound variable by passing the cdr part of the original bindings list into the . For example:

```
(expr-compare '(let ((a (lambda (b a) (b a))))
                 (eq? a ((lambda (a b) (let ((a b) (b a)) (a b)))
                         a (lambda (a) a))))
              '(let ((a (lambda (a b) (a b))))
                 (eqv? a ((lambda (b a) (let ((a b) (b a)) (a b)))
                          a (lambda (b) a)))))
```

In this examplem the binding list of the first expression is :
`((a (lambda (b a) (b a))))`
There is a lambda keyword inside the keyword ‘let’ bindings. In order to jump over it. We recursively call the function by passing in the cdr part, which is `((b a) (b a))` in this example.

```
65          ;;base condition 3: At least one of the leading elemer
66          ((or
67              (member (car (car l1)) '(if let quote lambda))
68              (member (car (car l2)) '(if let quote lambda))
69              (list? (car (car l1)))
70              (list? (car (car l1))))
71           (get-let-bindings (cdr l1) (cdr l2) b)
72          )
```

4. If the car of the both bindings are the same. They are not new bound variables and we moves on to the cdr part. For example:

```
(expr-compare '(+ #f (let ((a 1) (b 2)) (f a b)))
              '(+ #t (let ((a 1) (c 2)) (f a c))))

  ⇒ (+
     (not %)
     (let ((a 1) (b!c 2)) (f a b!c)))
```

Here, the original binding lists are
`((a 1) (b 2))` and `((a 1) (c 2))` respectively. Obviously, `a` is not the bound variable in this case.

```
73          ;;base condition 4: The leading varibales of the two bindings are the same. It is not bound variable and jump over it.
74          ((equal? (car (car l1)) (car (car l2))) (get-let-bindings (cdr l1) (cdr l2) b))
```

5. base base condition: If none of the above four conditions are fulfilled, then this means we found two different variables in the same place of the two expressions and we store these variables as a pair into the bound variable list b and recursively find the rest part of the binding lists. Note: In order to make a list, we should cons empty list , `'()`, at the end.

```
75
76          ;;Otherwise, we found a bound variable,say a!b, and add it into binding list b. b has the list format '
77          ;;Meanwhile, recursively call the function to keep looking through the remaining expressions.
78          (else (get-let-bindings (cdr l1) (cdr l2) (cons (cons (car (car l1)) (cons (car (car l2)) empty)) b)))
```

The whole functions look like:

```
(define (get-let-bindings l1 l2 b)
    (cond
        ;;base condition 1: at least of the x and y's bindings is empty, then simply pass through binding list b
        ((or (empty? l1) (empty? l2)) b)
        ;;base condition 2: The length of the bindings of x and y are different, then it is impossible to have bou
        ((not (equal? (length l1) (length l2)) ) b)
        ;;base condition 3: At least one of the leading element is keyword or a list. It is not bound variable.
        ((or
            (member (car (car l1)) '(if let quote lambda))
            (member (car (car l2)) '(if let quote lambda))
            (list? (car (car l1)))
            (list? (car (car l1))))
         (get-let-bindings (cdr l1) (cdr l2) b)
        )
        ;;base condition 4: The leading varibales of the two bindings are the same. It is not bound variable and j
        ((equal? (car (car l1)) (car (car l2))) (get-let-bindings (cdr l1) (cdr l2) b))

        ;;Otherwise, we found a bound variable,say a!b, and add it into binding list b. b has the list format '((a
        ;;Meanwhile, recursively call the function to keep looking through the remaining expressions.
        (else (get-let-bindings (cdr l1) (cdr l2) (cons (cons (car (car l1)) (cons (car (car l2)) empty)) b)))
    )

)
```

Testing for the function **get-let-bindings** :

```
> (get-let-bindings '((a c)) '((b d)) empty)
'((a b))
>
```

```
> (get-let-bindings '((a 1) (b 2)) '((a 1) (c 2)) empty)
'((b c))
>
```

It works!

Similarly, to get the bound variables after the keyword lambda is the same as let. Except that there is one less pair of () in the bindings part after lambda.

```
(expr-compare '((lambda (a b) (f a b)) 1 2)
              '((lambda (a c) (f c a)) 1 2))
  ⇒ ((lambda (a b!c) (f (if % a b!c) (if % b!c a)))
     1 2)
```

To illustrate, as shown in the above expample, the binding part of lambda of the first expression is (a b) not ((a b)).

PART II

Now, I am going to do the PART II : To get all bound variables directly from the expressions not from the binding list as before.

Base conditions:

1. Two input expressions are structual equivalent. There is no bound variable.
2. Two input expressions are not structual equivalent and at least one of them is not a list. There is no bound variable.
3.Two input expressions are not structual equivalent and they are both lists with different length. There is no bound variable.

After the above three conditions, from now on, the following conditions are based on the two input expressions are both lists with the same length.

4.leading keyword are both ‘let
5. leading keyword are both ‘lamda
6. If the two input expressions are both beginning with a list
7. recursively call the function to go over the rest part of the expressions.

I comment every lines in my code to explain how does the code work.

```
(define (get-all-bindings x y b)
    (cond
        ;;base condition 1:structural equivalence?
        ((equal? x y) b)

        ;;base condition 2:Not structural equivalence and at least one of x and y is not a list
        ((not (and (list? x)(list? y))) b) ;; if at least one of x and y is not a list, then simpl

        ;;base condition 3:x and y are both lists and not structural equivalence and different len
        ( (not (equal? (length x) (length y) ) ) b )

        ;;base condition 4:leading keywords are both let
        ((and (equal? (car x) 'let) (equal? (car y) 'let))
         (append (append (get-let-bindings (cadr x) (cadr y) empty) ;; get the bound variables ins
                         (get-all-bindings (cadr x) (cadr y) empty)) ;; in case there are nested b
                 (append (get-all-bindings (cdr x) (cdr y) b) b) ;; recursively call the function
         )
        )

        ;;base condition 5: leading keywords are both lamda, similar to condition 4.
        ((and (equal? (car x) 'lambda) (equal? (car y) 'lambda))
         (append (append (get-lambda-bindings (cadr x) (cadr y) empty) ;; get the bound variables
                         (get-all-bindings (cadr x) (cadr y) empty)) ;; in case there are nested b
                 (append (get-all-bindings (cdr x) (cdr y) b) b) ;; recursively call the function
         )
        )

        ;;base condition 6: Dealing with the case when both x and y's car parts are lists . e.g. (
        ;;                                                                                    ⇒ ((la
        ( (and (list? (car x)) (list? (car y)))
          (append (get-all-bindings (car x) (car y) empty) (get-all-bindings (cdr x) (cdr y) b))
        )
        ;; tail recursive
        (else (get-all-bindings (cdr x) (cdr y) b)) ;; recursively call the funtion to parse the e


    )
)
```

Testing function get-all-bindings:

```
[> (get-all-bindings '((lambda (a b) (f a b)) 1 2) '((lambda (a c) (f c a)) 1 2) ]
 empty)
 '((b c))
[> (get-all-bindings '(let ((a c)) a) '(let ((b d)) b) empty)                    ]
 '((a b))
 >
```

It works!

Part C

Now, I have the difference summary without the format of bound variables from part A.

I have the gotten all bound variables from the two input expressions.

The only thing left is to replace the (if % X Y) with the bound variable X!Y in the proper position.

For example:

```
> (prefinished-summary-expr '(let ((a c)) a) '(let ((b d)) b))
'(let (((if % a b) (if % c d))) (if % a b))
> (get-all-bindings '(let ((a c)) a) '(let ((b d)) b) empty)
'((a b))
>
```

The only thing left is to replace (if % a b) to a!b from the above example.

First, I am going to write a function to combine the bound variable as the format X!Y from the bound variable list ‘((x y))

After searching from the Racket maual. I found the string->symbol and symbol->string can be used to implement the function:

```
132  (define (combine-binding a b)
133    (string->symbol (string-append (symbol->string a) "!" (symbol->string b)))
134  )
```

```
> (combine-binding 'a 'b)
'a!b
>
```

Then, I need to know where should I replace it.

```
133   ;;(cons 'a (cons 'b empty))
134   (define (is-binding a b bindings)
135       (member (cons (cons a(cons b empty)) empty) bindings)
136   )
137
```

```
> (is-binding 'a 'b '((a b)))
#t
> (is-binding 'a 'c '((a b)))
#f
>
```

It works!

Finally, I am going to iterate through my expression and replace with bound variables if need.

Base conditions:
1. If the result difference summary expression is empty, then should not replace anything.
2. If the result difference summary expression is not a list, then there is no need to replace because there is no bound variable. E.g.
`(expr-compare 12 12)  ⇒  12`
3 Since the length of the format (if % X Y) is 4, we only need to check when the length of current element is 4. If so, get X and Y and check if they are bound variables by checking if they are member inside the bound variables list. If so, combine X and Y to X!Y and substitute for X!Y.
4. recursively call the function to iterate through all the remainning elements in the expressions.

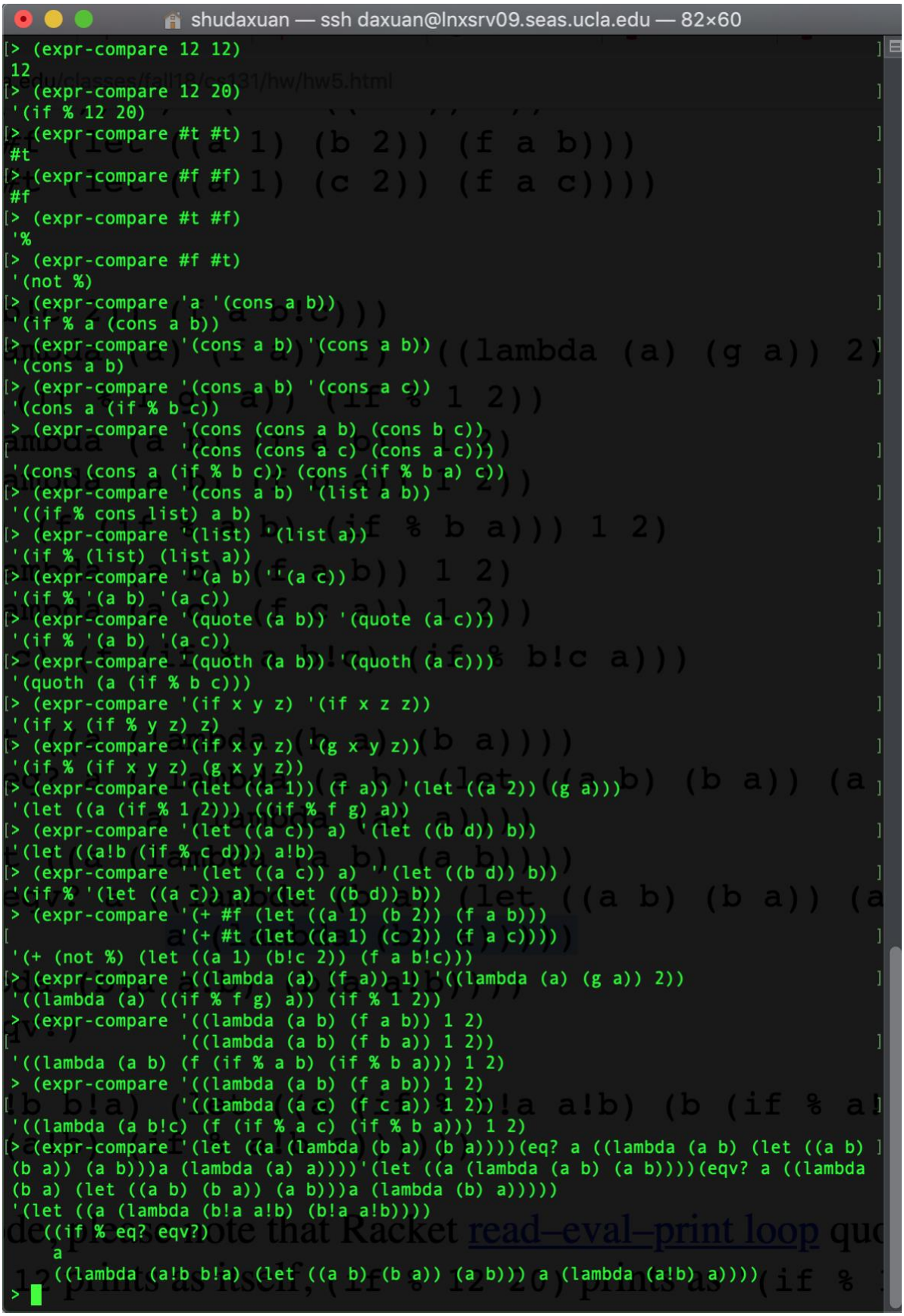
```
(define (replacer expr bindings)
  (cond
    ((empty? expr) expr)
    ((not (list? expr)) expr)
    ((and
      (equal? (length expr) 4)
      (and (equal? (car expr) 'if) (equal? (cadr expr) '%))
      (and (is-binding (caddr expr) (cadddr expr) bindings))
    ) (combine-binding (caddr expr) (cadddr expr)))
    (else
        (cons (replacer (car expr) bindings) (replacer (cdr expr) bindings)))
  )
)
```

Part D

Finally, combine all the functions to get the difference summary as we want.

```
(define (expr-compare x y)
  (replacer (prefinished-summary-expr x y) (get-all-bindings x y empty))
)
```

Testing:


shudaxuan — ssh daxuan@lnxsrv09.seas.ucla.edu — 82×60
> (expr-compare 12 12)
12
> (expr-compare 12 20)
'(if % 12 20)
> (expr-compare #t #t)
#t
> (expr-compare #f #f)
#f
> (expr-compare #t #f)
'%
> (expr-compare #f #t)
'(not %)
> (expr-compare 'a '(cons a b))
'(if % a (cons a b))
> (expr-compare '(cons a b) '(cons a b))
'(cons a b)
> (expr-compare '(cons a b) '(cons a c))
'(cons a (if % b c))
> (expr-compare '(cons (cons a b) (cons b c))
'(cons (cons a c) (cons a c)))
'(cons (cons a (if % b c)) (cons (if % b a) c))
> (expr-compare '(cons a b) '(list a b))
'((if % cons list) a b)
> (expr-compare '(list) '(list a))
'(if % (list) (list a))
> (expr-compare ''a b) ''a c))
'(if % 'a b) 'a c))
> (expr-compare '(quote (a b)) '(quote (a c)))
'(if % '(a b) '(a c))
> (expr-compare '(quoth (a b)) '(quoth (a c)))
'(quoth (a (if % b c)))
> (expr-compare '(if x y z) '(if x z z))
'(if x (if % y z) z)
> (expr-compare '(if x y z) '(g x y z))
'(if % (if x y z) (g x y z))
> (expr-compare '(let ((a 1)) (f a)) '(let ((a 2)) (g a)))
'(let ((a (if % 1 2))) ((if % f g) a))
> (expr-compare '(let ((a c)) a) '(let ((b d)) b))
'(let ((a!b (if % c d))) a!b)
> (expr-compare ''(let ((a c)) a) ''(let ((b d)) b))
'(if % '(let ((a c)) a) '(let ((b d)) b))
> (expr-compare '(+ #f (let ((a 1)) (b 2)) (f a b)))
'(+ #t (let ((a 1)) (c 2)) (f a c))))
'(+ (not %) (let ((a 1) (b!c 2)) (f a b!c)))
> (expr-compare '((lambda (a) (f a)) 1) '((lambda (a) (g a)) 2))
'((lambda (a) ((if % f g) a)) (if % 1 2))
> (expr-compare '((lambda (a b) (f a b)) 1 2)
'((lambda (a b) (f b a)) 1 2))
'((lambda (a b) (f (if % a b) (if % b a))) 1 2)
> (expr-compare '((lambda (a b) (f a b)) 1 2)
'((lambda (a c) (f c a)) 1 2))
'((lambda (a b!c) (f (if % a c) (if % b a))) 1 2)
> (expr-compare '(let ((a (lambda (b a) (b a)))) (eq? a ((lambda (a b) (let ((a b)
(b a)) (a b)))a (lambda (a) a))))'(let ((a (lambda (a b) (a b)))) (eqv? a ((lambda
(b a) (let ((a b) (b a)) (a b)))a (lambda (b) a)))))
'(let ((a (lambda (b!a a!b) (b!a a!b))))
((if % eq? eqv?)
a
((lambda (a!b b!a) (let ((a b) (b a)) (a b))) a (lambda (a!b) a))))
>

It passes all the test cases except for the last one, It seems could not deal with the case when there is a nested if keyword inside the binding. I tried to solve it always has the same problem.

Reference

1. Chapter 9 Primitive syntax. (n.d.). Retrieved March 19, 2019, from http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-12.html#node_sec_Temp_14

2. DakshIdnani. (n.d.). DakshIdnani/CS131. Retrieved March 19, 2019, from https://github.com/DakshIdnani/CS131/blob/master/hw5/expr-compare.ss

3. Flatt, M., & PLT. (n.d.). The Racket Reference. Retrieved March 19, 2019, from https://docs.racket-lang.org/reference/index.html?q=quoth

4. (n.d.). Retrieved March 19, 2019, from https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Comments.html

5. Shido, T. (n.d.). Retrieved March 19, 2019, from http://www.shido.info/lisp/scheme4_e.html