Homework Assignment 2

Sugeerth Murugesan

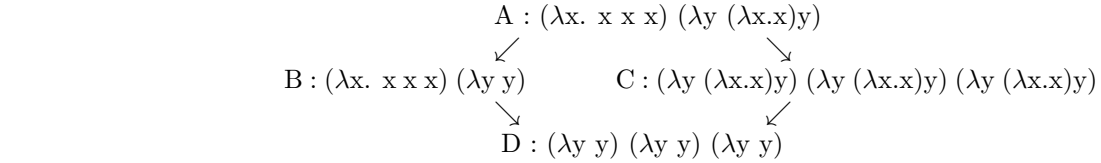
May 2, 2014

1 Diamond property

The Diamond property states that a relation If a $\rightarrow$ b and a $\rightarrow$ c, then there exists d such that b $\rightarrow_*$ d and c $\rightarrow_*$ d, where a,b,c,d are λ expressions.

Unfortunately, this idea does not work in general. It means that it is not possible to obtain c in just one step from b , c . Let us consider an example here that does not satisfy these conditions.

Consider λ expression



The lambda term A reduces to B and C. C and B reduce to D, but there is not a single step reduction from C to D while there is a single reduction from B to D. This as a result does not satisfy the diamond rule.

2 Normal order, call-by-name. call-by-value

To consider the λ terms that represent the evaluation order of the three orders are :

Normal order: In a normal order the evaluation order is governed by evaluating the outermost leftmost redex is first. [1]

Consider

(λa (λb.a)) c ((λd.e)d)

[Taking into consideration the outermost leftmost redex for evaluation]

$\mapsto_\beta$ (λa c) ((λd.e)d)

$\mapsto_\beta$ (λa c) e

$\mapsto_\beta$ c

Call-by-name:

Call-by-name evaluates outermost leftmost redex first, but not inside abstractions. Also it is an application of the lazy-evaluation in Haskell, the expression is evaluated only if it is needed.

One such example is:

(λa (λb.a)) c ((λd.e)d)
$\mapsto_\beta$ (λb.c ((λd.e)d))
$\mapsto_\beta$ c ((λd.e)d))
$\mapsto_\beta$ (λb.c)e)
$\mapsto_\beta$ c
Call-by-value:
Call-by-name evaluates the arguments to function call first.
(λa (λb.a)) c ((λd.e)d)
$\mapsto_\beta$ (λa (λb.a)) c (e)
$\mapsto_\beta$ (λb.c) e
$\mapsto_\beta$ c

3 Encoding lists in lambda calculus

Encoding lists in lambda calculus is done using church encoding for pairs[1].Essentially, the church encoding represents data and operators in lambda calculus. A basic 2-tuple pair can be defined in terms of TRUE or FALSE using the church rncoding for pairs[1].

The pair consists of a function that takes a function as an argument. The argument is applied to the two component pair.

When defining a list into consideration we take into account
nil for empty list,
length for the length of the list,
cons for prepending items in the list and
append for appending items.
to encode this using a pair type, the definition can be,
PAIR $\equiv \lambda x.\lambda y.\lambda z.\ z\ x\ y$
FIRST $\equiv \lambda p.p.(\lambda x.\lambda y.x)$
SECOND $\equiv \lambda p.p.(\lambda x.\lambda y.y)$

FUNCTION

nil $\equiv$ pair true true
Means that the first element of the pair is true and hence the list is null

cons $\equiv \lambda H.\lambda T.\ pair\ false\ (pair\ H\ T)$ [2] Means that a node is created and head is assigned to H , tail is assigned to T.

append $\equiv \lambda H.\lambda T.\ pair\ true\ (pair\ H\ T)$ [2] Means that a node is created and head is assigned to H , tail is assigned to T and the new node is inserted from the head.

length $\equiv \lambda l.l(\lambda H.\lambda T.\lambda D.\ pair\ (SUCC\ pair))\ (pair\ nil\ nil)$ [2] Also natural numbers are defined as:

SUCC $= \lambda n.\lambda f.\lambda x.\ f(n\ f\ x)$ [2] Usage a special case of rescursive defintion of the lists bu using the foldr operation. [3]. The last pair would

have the value (n-1 n) and returning the tail of the pair would yield "n" the value of length.

4 Encoding expressions

An expression of exp(n,m) would be adding n m times :

POW : $\equiv \lambda y.\lambda z.\ z\ y$ [1] [2]

For a predessesor would be:

PRED :$\equiv \lambda n.FIRST\ (\ n\ \phi\ (PAIR\ 0\ 0\ ))$ [2]

Where ϕ is the shift and increment function that maps a (m,n) to (m,n+1) [1]

5 Encoding factorial functions

Considering the factorial function which is

F(n) = 1, if n=0;

else n * F(n-1) [2]

As the argument in the factorial expression would recieve the expression itself, it will amount to recursion. We define a "self-referencing" pointer 'q' that will be passed to itself within the body. [2]. WE will use a fixed point combinator to return a self-replicating lambda function.

S := $\lambda q.\ \lambda n.$ (1, if n =0 ;else n * (r (n-1)))
with q x = F x = S q x to hold,so q= S q =: FIX S and

F:= FIX S where FIX g:= (q where q = g q) = g (FIX g)
so that FIX S = S (FIX S) = ($\lambda n.$ (1, if n = 0;else n* ((FIX S) (n-1))))

FIX operator being:
W := $\lambda g.$ (λx.g (x x)) (λx.g (x x))

Recursive call to the factorial function of n = 2

1. (W S) 2
2. S (W S) 2
3. ($\lambda q.\ \lambda n.$ (1, if n =0 ;else n * (r (n-1)))) (W S) 2
4. ($\lambda n.$ (1, if n = 0 ;else n * ((W S) (n-1)))) 2
5. 1, if 2 = 0;else 2 * ((W S) (2-1))
6. 2 (S (W S) (21))

7. 2 ((n.(1, if n = 0; else n ((W S) (n1)))) (21))
8. 2 (1, if 3 = 0; else 3 ((W S) (11)))
9. (2 (1 (S (W S) (11))))
10. (2 (1 ((n.(1, if n = 0; else n ((W S) (n1)))) (11))))
11. (2 (1 (1, if 0 = 0; else 0 ((Y G) (01)))))
12. (2 (1 (1)))
13. 2 [2] [3]

References

[1] http://en.wikipedia.org/wiki/ChurchencodingChurchpairs , *Church Pairs*

[2] http://en.wikipedia.org/wiki/LambdacalculusPairs, *Lambda calculus in Pairs*

[3] Ken Slonneger Chapter 5, http://homepage.cs.uiowa.edu/ slonnegr/plf/Book/Chapter5.pdf *LAMBDA CALCULUS*

[4] Zhengdong Su ECS 240 teaching UC Davis, http://www.cs.ucdavis.edu/ su/teaching/ecs240-s14/lectures/lecture04.pdf *LAMBDA CALCULUS*

[5] Zhengdong Su ECS 240 teaching UC Davis, http://www.cs.ucdavis.edu/ su/teaching/ecs240-s14/lectures/lecture05.pdf *LAMBDA CALCULUS*

[6] Ranjit Jhala CSE 230: Principles of Programming Languages, Winter 08 UCSD, http://cseweb.ucsd.edu/classes/wi08/cse230/ *Programming Languages*

[7] Franco Barbanera Short Introduction to the Lambda-calculus, http://www.dmi.unict.it/ barba/LinguaggiII.html *Programming Languages*