Lambda Calculus Through Prolog

Aaron Gorenstein

January 1, 2021

1 Introduction

⟨*expression*⟩ ::= ⟨*name*⟩
  | ⟨*function*⟩
  | ⟨*application*⟩

⟨*function*⟩ ::= λ ⟨*name*⟩ . ⟨*expression*⟩

⟨*application*⟩ ::= (⟨*expression*⟩ ⟨*expression*⟩)

⟨*Initial Lambda Calculus Definitions*⟩≡

```
expression(L) :- name(L).
expression(L) :- function(L).
expression(L) :- application(L).

function([lambda, V, B]) :- name(V), expression(B).

application([E1, E2]) :- expression(E1), expression(E2).

name(X) :- not(is_list(X)), ground(X), X \= lambda.
```

⟨*Initial Lambda Calculus Definitions*⟩+≡

```
function([lambda, V, B], V, B) :- function([lambda, V, B]).

application([E1, E2], E1, E2) :- application([E1, E2]).
```

⟨*Beta Reduction*⟩≡

```
beta_reduction(F,A,R) :- function(F,V,B), replace(V,B,A,R).
```

⟨*Replace Predicate*⟩≡
```
replace(V,V,A,A) :- name(V).
replace(V,W,_,W) :- name(W), V \= W.
replace(V,F,_,F) :- function(F,V,_).
replace(V,F,A,R) :- function(F,W,B), W \= V,
    replace(V,B,A,S),
    function(R,W,S).
replace(V,P,A,R) :- application(P,E1,E2),
    replace(V,E1,A,R1),
    replace(V,E2,A,R2),
    application(R,R1,R2).
```

⟨*BetaReductions.tests*⟩≡
```
beta_reduction([lambda, x, x], a, R), R == a.
beta_reduction([lambda, x, x], b, R), R \= a.
beta_reduction([lambda, x, x], [lambda, y, y], R), R == [lambda, y, y].
beta_reduction([lambda, x, [x, z]], [lambda, y, y], R), R == [[lambda, y, y], z].
```

⟨*Evaluate*⟩≡
```
evaluate(L, L) :- name(L).
evaluate(L, L) :- function(L).
evaluate(L, L) :- application(L, E1, E2), name(E1).
evaluate(L, R) :- application(L, E1, E2), function(E1),
    beta_reduction(E1, E2, R1),
    evaluate(R1, R).
evaluate(L, R) :- application(L, E1, E2), application(E1),
    evaluate(E1, R1),
    application(S, R1, E2),
    evaluate(S, R).
```

⟨*Sugaring*⟩≡
```
desugar(L,[N,E],R) :-
    function(F,N,L),
    beta_reduction(F,E,R).

desugar_all(L,[],L).
desugar_all(L,[M|T],R) :- desugar(L,M,S), desugar_all(S,T,R).
```

⟨*Alpha Reduction*⟩≡

```
alpha_reduction(L, L) :- name(L).
alpha_reduction(L, R) :- application(L, E1, E2),
    alpha_reduction(E1, R1),
    alpha_reduction(E2, R2),
    application(R, R1, R2).
alpha_reduction(L, R) :- function(L),
    gensym(alpha_,X),
    beta_reduction(L, X, BB),
    alpha_reduction(BB, ABB),
    function(R, X, ABB).

canon(L, R) :- reset_gensym(alpha_), alpha_reduction(L, R).

isomorphic(A, B) :- canon(A, C), canon(B, C).
```

⟨*AlphaReduction.tests*⟩≡

```
isomorphic([lambda, x, x], [lambda, y, y]).
```

⟨*Resugaring*⟩≡

```
resugar(L,[N,E],N) :- isomorphic(L, E).
resugar_all(L, [], L).
resugar_all(L, [M|T], R) :- resugar(L, M, S), resugar_all(S, T, R).
```

⟨*FirstLoop.filetest*⟩≡

```
(define true (lambda x (lambda y x)))
(define false (lambda x (lambda y y)))
(define and (lambda x (lambda y ((x y) false))))
(define not (lambda x ((x false) true)))
(define nand (lambda x (lambda y (not ((and x) y)))))
(print)
(execute ((nand true) true))
(execute ((nand true) false))
(execute ((nand false) true))
(execute ((nand false) false))
(halt)
```

⟨*Main Loop*⟩≡

```
% This (I think) takes a list of atoms like ['(', 'lambda, 'x', 'x', ')']
% and turns it into the internal prolog represention [lambda, x, x].
% __The tricky part was nesting parenthesis and matching them.__
parse(['('|T], [M|L], R) :- parse(T,M,[')'|S]), parse(S,L,R).
parse([')'|T], [], [')'|T]).
parse([N|T], [N|L], R) :- N \= ')', N \= '(', parse(T,L,R).
parse([],[],[]).
parse(S,L) :- parse(S,L,[]).


% This is the main I/O driver: it reads from stdin and tokenizes
% things into "names" or parenthesis. That's it.
read_s(S) :- get_char(C), read_s(C,S).
read_s(C,S)     :- whitespace(C), get_char(D),      read_s(D, S).
read_s(C,[C|S]) :- parens(C),     get_char(D),      read_s(D, S).
read_s(C,[N|S]) :- name_char(C),  read_name(C,N,D), read_s(D, S).
read_s(C,[])    :- end_of_line(C).

whitespace(' ').
parens('(').
parens(')').
name_char(C) :- char_code(C,N), N >= 65, N =< 90. % upper case
name_char(C) :- char_code(C,N), N >= 97, N =< 122. % lower case.
name_char(C) :- char_code(C,N), N == 95.
name_char(C) :- char_code(C,N), N >= 48, N =< 57.
end_of_line('\n').

read_name(C,N,E) :-
    get_char(D),
    read_name_chars(S,D,E),
    atom_chars(N,[C|S]).
read_name_chars([C|S], C, E) :-
    name_char(C), !, % needed for io
    get_char(D),
    read_name_chars(S, D, E).
read_name_chars([],C,C) :- not(name_char(C)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is just raw output helpers, designed to output the result in a way
% that's compatible with our input-parser.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

atom_concat_list([],'').
atom_concat_list([A|T],S) :- atom_concat_list(T,R), atom_concat(A,R,S).
```

```
% this creates a "pretty printer" for lambda expressions, translating
% from the internal prolog representation to a more scheme-like parens
% syntax.
internal_lambda_string(L,S) :- function(L,V,B),
    internal_lambda_string(V,VS),
    internal_lambda_string(B,BS),
    atom_concat_list(['(lambda ', VS, ' ', BS, ')'],S).

internal_lambda_string(L,S) :- application(L,F,A),
    internal_lambda_string(F,FS),
    internal_lambda_string(A,AS),
    atom_concat_list(['(', FS, ' ', AS, ')'], S).

internal_lambda_string(L,L) :- name(L).

execute_command([compute, L], OD, OD) :-
    evaluate(L, R),
    internal_lambda_string(R, RS),
    write(RS),
    nl.
execute_command(['define', N, L], OD, [[N,L]|OD]).
execute_command(['execute', L], OD,OD) :-
    internal_lambda_string(L,LS),
    write('Executing '), write(LS), nl,
    desugar_all(L,OD,R),
    write(R), nl,
    internal_lambda_string(R,RS),
    write(RS), nl,
    evaluate(R,Z),
    internal_lambda_string(Z,ZS),
    write(RS), nl, write('\t=>\t'), nl, write(ZS), nl,
    % The neat thing in lambda calculus that we associate meaning with
    % weird sentences, like (lambda x x) is true. But when we're done
    % calculating a lambda sentence, it would be nice to express
    % it in terms of some previously-defined macro (e.g., "true") rather
    % than the raw lambda sentence (lambda x x). My hacky implementation
    % is invoked here.
    write('Reversing'), nl,
    resugar_all(Z,OD,FINAL),
    write('Done reversing: '),
    internal_lambda_string(FINAL,FINAL_STRING),
    write(FINAL_STRING), nl.
execute_command(['alpha', L], D, D) :-
    internal_lambda_string(L,LS),
    write('Renaming '), write(LS), nl,
    desugar_all(L,D,R),
```

```
    internal_lambda_string(R,RS),
    alpha_reduction(R,A),
    internal_lambda_string(A,AS),
    write(RS), nl, write('\t=>\t'), nl, write(AS), nl.

execute_command(['print'], [],[]) :- nl.
execute_command(['print'], [E|D],[E|D]) :-
    internal_lambda_string(E,S),
    write(S), nl,
    execute_command(['print'], D, D).
execute_command(['halt'],_,_) :- halt.

main_loop(OD) :-
    read_s(S),
    parse(S,[L]), % I strip out the other list immediately. Why not.
    execute_command(L,OD,ND),
    main_loop(ND).
main_loop(OD) :- write('Parse error'), nl, main_loop(OD).

% :- initialization main.
main :- main_loop([]), halt.
```

⟨*lambda.pl*⟩≡

⟨*Initial Lambda Calculus Definitions*⟩

⟨*Beta Reduction*⟩

⟨*Replace Predicate*⟩

⟨*Evaluate*⟩

⟨*Alpha Reduction*⟩

⟨*Sugaring*⟩
⟨*Resugaring*⟩

⟨*Main Loop*⟩

⟨*Evaluate.tests*⟩≡

```
evaluate([[lambda, x, x], [lambda, x, x]], R), R == [lambda, x, x].
evaluate([[lambda, x, [x, x]], [lambda, x, x]], R), R == [lambda, x, x].
```

A Complete Code Listings

A.1 lambda.pl

```
expression(L) :- name(L).
expression(L) :- function(L).
expression(L) :- application(L).

function([lambda, V, B]) :- name(V), expression(B).

application([E1, E2]) :- expression(E1), expression(E2).

name(X) :- not(is_list(X)), ground(X), X \= lambda.
function([lambda, V, B], V, B) :- function([lambda, V, B]).

application([E1, E2], E1, E2) :- application([E1, E2]).

beta_reduction(F,A,R) :- function(F,V,B), replace(V,B,A,R).

replace(V,V,A,A) :- name(V).
replace(V,W,_,W) :- name(W), V \= W.
replace(V,F,_,F) :- function(F,V,_).
replace(V,F,A,R) :- function(F,W,B), W \= V,
    replace(V,B,A,S),
    function(R,W,S).
replace(V,P,A,R) :- application(P,E1,E2),
    replace(V,E1,A,R1),
    replace(V,E2,A,R2),
    application(R,R1,R2).

evaluate(L, L) :- name(L).
evaluate(L, L) :- function(L).
evaluate(L, L) :- application(L, E1, E2), name(E1).
evaluate(L, R) :- application(L, E1, E2), function(E1),
    beta_reduction(E1, E2, R1),
    evaluate(R1, R).
evaluate(L, R) :- application(L, E1, E2), application(E1),
    evaluate(E1, R1),
    application(S, R1, E2),
    evaluate(S, R).

alpha_reduction(L, L) :- name(L).
alpha_reduction(L, R) :- application(L, E1, E2),
    alpha_reduction(E1, R1),
    alpha_reduction(E2, R2),
    application(R, R1, R2).
```

```prolog
alpha_reduction(L, R) :- function(L),
    gensym(alpha_,X),
    beta_reduction(L, X, BB),
    alpha_reduction(BB, ABB),
    function(R, X, ABB).

canon(L, R) :- reset_gensym(alpha_), alpha_reduction(L, R).

isomorphic(A, B) :- canon(A, C), canon(B, C).

desugar(L,[N,E],R) :-
    function(F,N,L),
    beta_reduction(F,E,R).

desugar_all(L,[],L).
desugar_all(L,[M|T],R) :- desugar(L,M,S), desugar_all(S,T,R).
resugar(L,[N,E],N) :- isomorphic(L, E).
resugar_all(L, [], L).
resugar_all(L, [M|T], R) :- resugar(L, M, S), resugar_all(S, T, R).

% This (I think) takes a list of atoms like ['(', 'lambda, 'x', 'x', ')']
% and turns it into the internal prolog represention [lambda, x, x].
% __The tricky part was nesting parenthesis and matching them.__
parse(['('|T], [M|L], R) :- parse(T,M,[')'|S]), parse(S,L,R).
parse([')'|T], [], [')'|T]).
parse([N|T], [N|L], R) :- N \= ')', N \= '(', parse(T,L,R).
parse([],[],[]).
parse(S,L) :- parse(S,L,[]).


% This is the main I/O driver: it reads from stdin and tokenizes
% things into "names" or parenthesis. That's it.
read_s(S) :- get_char(C), read_s(C,S).
read_s(C,S)     :- whitespace(C), get_char(D),      read_s(D, S).
read_s(C,[C|S]) :- parens(C),     get_char(D),      read_s(D, S).
read_s(C,[N|S]) :- name_char(C),  read_name(C,N,D), read_s(D, S).
read_s(C,[])    :- end_of_line(C).

whitespace(' ').
parens('(').
parens(')').
name_char(C) :- char_code(C,N), N >= 65, N =< 90. % upper case
name_char(C) :- char_code(C,N), N >= 97, N =< 122. % lower case.
name_char(C) :- char_code(C,N), N == 95.
name_char(C) :- char_code(C,N), N >= 48, N =< 57.
end_of_line('\n').
```

```
read_name(C,N,E) :-
    get_char(D),
    read_name_chars(S,D,E),
    atom_chars(N,[C|S]).
read_name_chars([C|S], C, E) :-
    name_char(C), !, % needed for io
    get_char(D),
    read_name_chars(S, D, E).
read_name_chars([],C,C) :- not(name_char(C)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is just raw output helpers, designed to output the result in a way
% that's compatible with our input-parser.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

atom_concat_list([],'').
atom_concat_list([A|T],S) :- atom_concat_list(T,R), atom_concat(A,R,S).

% this creates a "pretty printer" for lambda expressions, translating
% from the internal prolog representation to a more scheme-like parens
% syntax.
internal_lambda_string(L,S) :- function(L,V,B),
    internal_lambda_string(V,VS),
    internal_lambda_string(B,BS),
    atom_concat_list(['(lambda ', VS, ' ', BS, ')'],S).

internal_lambda_string(L,S) :- application(L,F,A),
    internal_lambda_string(F,FS),
    internal_lambda_string(A,AS),
    atom_concat_list(['(', FS, ' ', AS, ')'], S).

internal_lambda_string(L,L) :- name(L).

execute_command([compute, L], OD, OD) :-
    evaluate(L, R),
    internal_lambda_string(R, RS),
    write(RS),
    nl.
execute_command(['define', N, L], OD, [[N,L]|OD]).
execute_command(['execute', L], OD,OD) :-
    internal_lambda_string(L,LS),
    write('Executing '), write(LS), nl,
    desugar_all(L,OD,R),
    write(R), nl,
    internal_lambda_string(R,RS),
```

```
    write(RS), nl,
    evaluate(R,Z),
    internal_lambda_string(Z,ZS),
    write(RS), nl, write('\t=>\t'), nl, write(ZS), nl,
    % The neat thing in lambda calculus that we associate meaning with
    % weird sentences, like (lambda x x) is true. But when we're done
    % calculating a lambda sentence, it would be nice to express
    % it in terms of some previously-defined macro (e.g., "true") rather
    % than the raw lambda sentence (lambda x x). My hacky implementation
    % is invoked here.
    write('Reversing'), nl,
    resugar_all(Z,OD,FINAL),
    write('Done reversing: '),
    internal_lambda_string(FINAL,FINAL_STRING),
    write(FINAL_STRING), nl.
execute_command(['alpha', L], D, D) :-
    internal_lambda_string(L,LS),
    write('Renaming '), write(LS), nl,
    desugar_all(L,D,R),
    internal_lambda_string(R,RS),
    alpha_reduction(R,A),
    internal_lambda_string(A,AS),
    write(RS), nl, write('\t=>\t'), nl, write(AS), nl.

execute_command(['print'], [],[]) :- nl.
execute_command(['print'], [E|D],[E|D]) :-
    internal_lambda_string(E,S),
    write(S), nl,
    execute_command(['print'], D, D).
execute_command(['halt'],_,_) :- halt.

main_loop(OD) :-
    read_s(S),
    parse(S,[L]), % I strip out the other list immediately. Why not.
    execute_command(L,OD,ND),
    main_loop(ND).
main_loop(OD) :- write('Parse error'), nl, main_loop(OD).

% :- initialization main.
main :- main_loop([]), halt.
```