Paul McJones

Date: August 15, 1972

n (Div. & location
S. mail address): T.J. Watson Research Center
Yorktown Heights, N.Y.

Dept. & Bldg: 440-801

one & Tel. Ext.: 862 - 1464

IBM

Subject: Three algorithms to convert lambda expressions into RED language expressions.

Reference: RJ-1010

To: File

The attached is a brief write up of three algorithms for converting lambda expressions into variable free expressions in the RED language (see "Reduction Languages and Variable free programming" by John Backus, RJ-1010).

P. D. Summers

The trouble with these algorithms is when λ-expressions containing subexpressions with no normal form are translated, the resulting Red expression will always be undefined (since Red semantics decrees "call-by-value" or "bottom-up" evaluation), even if the original λ-expression as a whole does have a normal form. For example YF never has a normal form, but for suitable F and X, YFX does.

965 - IBM - 04

I. Introduction

The three algorithms presented in this memo are adapted from three algorithms for reducing λ-expressions with bound variables into variable-free expressions in combinatorial logic.

Two of the algorithms correspond to algorithms for strong and weak abstraction* and the third, I call "explosive-abstraction" because of the size of the expressions generated.

All of the algorithms start with wf λ-expression and syntactically transform it into a wff of combinatorial logic, containing the combinators S, K, and I defined below. The correctness of the Algorithms have been demonstrated (See for example Curry and Feys-Chap.6) and I will only demonstrate the correctness of the defined RED operators corresponding to S, K, I. The algorithms will be presented informally but with enough detail so that implementation is straight forward.

II. Definitions & Notation

A. A λ-expression is

1. simple, x, where x is an atom or
2. an applications, $\langle P\ Q \rangle$, where P is the operator and Q is the operand. (One or more blanks seperates operator from operand. An underline

*I have taken the terms "weak" and "strong abstraction" from a private correspondence from Prof. J.A. Robinson, Syracuse University. I have been unable to locate their use in the open literature. Curry and Feys discuss a variety of abstraction algorithms (Chap. 6-Combinatory Logic, Vol. 1. But do not use these names.

will be used to denote a blank when needed eg. ⟨A B⟩. The operator and operand must be a wf λ-expression.

3. an abstraction, λx.B, where x is the bound variable and B is the body, which must be a wf lambda expression

4. Only expressions satisfying 1,2,3.

Examples: x

⟨A B⟩

λx.⟨F x⟩

⟨λx.⟨x x⟩ λx.⟨x x⟩⟩

λx.λy.⟨A ⟨λz.⟨x z⟩ B⟩⟩

B. Definition of RED expressions

The definition of a RED expression is that used in RJ-1010 for RED/1. (Note that application mean the same in both languages.) Symbols not defined in this memo are defined in RJ-1010, eg. □.

C. Functions for the algorithms.

simp [e] = T iff e is simple

app [e] = T iff e is an application

opr [e] = P if e ≡ P Q

= □ otherwise

opd [e] = Q if e ≡ P Q

= □ otherwise

abs [e] = T iff e is an abstraction

bdy [e] = B if e ≡ λx.B

= □ otherwise

vbl [e] = x if e ≡ λx.B

= □ otherwise

965 - IBM - 04

eql $[e_1;e_2]$ = T iff $e_1 = e_2$

cont $[e_1;e_2]$ = T iff e_1 is part of e_2

eg. cont $[x;\lambda x.\langle B\ x\rangle]$ = T

cont $[\langle A\ B\rangle;\ \lambda x.\langle A\ Q\rangle]$ = F

cont $[x;xyz]$ = F

~ Boolean "not"

∧ Boolean "and"

D. Red operators.

X_i	Y_i (Assume X_i, Y_i) $\in$ ∇)
I	ϕ
K	(ADJ C)
S	(UNION (ADJ UNION) UNION (ADJ (ADJ SS)) (ADJ ADJ))
SS	*(APP 2APP TRPRS (SEC DBL) (FST TL))

S,K,I correspond to the combinatorial constants

defined as follows:

a. $\langle I\ X\rangle \rightarrow X$

b. $\langle\langle K\ X\rangle\ Y\rangle \rightarrow X$

c. $\langle\langle\langle S\ X\rangle\ Y\rangle\ Z\rangle \rightarrow \langle\langle X\ Z\rangle\langle Y\ Z\rangle\rangle$

Justification for definition.

a. I is the identity operator .: ϕ is the corresponding RED operator having that property.

b. K is the constancy operator. It may be defined as being the operator, such that when applied to anything, yields an operator K', K' applied to a second operand yields the first operand.

$\langle(ADJ\ C)\ \ X\rangle \rightarrow (C\ X)$

$\langle(C\ X)\ \ Y\rangle \rightarrow X$

965 - IBM - 04

c. Similarly for S

1. ⟨(UNION (ADJ UNION) UNION (ADJ (ADJ SS)) (ADJ ADJ)) X⟩ → (UNION (ADJ SS) ADJ X)

2. ⟨(UNION (ADJ SS) ADJ X) Y⟩ → (SS X Y)

3. ⟨(SS X Y) Z⟩ → ⟨⟨X Z⟩⟨Y Z⟩⟩

∴ ⟨⟨⟨S X⟩Y⟩Z⟩ → ⟨⟨X Z⟩⟨Y Z⟩⟩

for the RED operator S

III. Algorithms

A. "Explosive-Abstraction"

This algorithm is the simplest of the three, but, unfortunately, it generates the largest expressions. The algorithm is presented as a recursive function σ_0, that has as input a representation of a lambda expression, and produces, as its value, the equivalent RED expression.

σ_0 [e] = [simp [e] → e;

app [e] → ⟨σ_0 [opd [e]] _ σ_1 [opd [e]]⟩;

abs [e] →

[abs [bdy [e]] → σ_0 [λ vbl [e] . σ_0 [bdy [e]]];

eql [vbl [e] ; bdy [e]] → I;

simp [bdy [e]] → ⟨K _ bdy [e]⟩;

T → ⟨⟨S _ σ_0 [λ vbl [e] . opr[bdy [e]]]⟩ _

σ_0 [λ vbl [e] . opd[bdy [e]]]⟩

T → □

Where juxtiposition of values represent some appropriate composition of entities to create expressions.

B. Weak Abstraction

The algorithm is based on the process called weak abstraction. It

produces shorter expressions than A but at the cost of being slightly more complicated. The conventions are as before except this algorithm is written as the function σ_1.

σ_1 [e] = [simp [e] → e;

app[e] → ⟨σ_1 [opr [e]] _σ_1 [opd [e]]⟩;

abs [e] →

[cont [λ; bdy [e]] → σ_1 [λ vbl [e].σ_1 [bdy [e]]];

[eql [vbl [e]; bdy [e]] → I

~ cont [vbl [e]; bdy [e]] →

⟨K _ bdy [e]⟩

T → ⟨⟨S _ σ_1 [λ vbl [e] . opr [bdy [e]]]⟩ _

σ_1 [λ vbl [e]. pod [bdy [e]]]⟩

T → □

C. Strong Abstraction

This algorithm σ_2, yields simpler expressions than σ_1 but is more complicated.

σ_2 [e] = [simp [e] → e;

app [e] → ⟨σ_2 [opr [e]]_σ_2 [opd [e]]⟩;

abs [e] →

cont [λ;bdy [e]] → σ_2 [λ vbl[e].σ_2[bdy [e]]];

eql[vbl[e]; bdy [e]] → I

~cont[vbl[e]; bdy [e]] →⟨K_bdy [e]⟩

eql [vbl[e] opd [bdy [e]]]∧

~cont [vbl[e]; opr[bdy[e]]] → opr[bdy[e]]

T → ⟨⟨S_σ_2[λ vbl [e].opr[bdy[e]]]⟩_

σ_2[λ vbl[e]. opd[bdy[e]]]⟩

T → □

965 - IBM - 04