PUC

Série: Monografias em Ciência da Computação
Nº 13/77

A SHORT INTRODUCTION TO THE λ-α-β-η CALCULUS WITH APPLICATIONS (FIRST DRAFT)

by

Atendolfo Pereda Bórquez

Roberto Lins de Carvalho

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Marquês de São Vicente 225 — ZC 19

Rio de Janeiro — Brasil

Série: Monografias em Ciência da Computação

Nº 13/77

A SHORT INTRODUCTION TO THE λ-α-β-η CALCULUS WITH APPLICATIONS (FIRST DRAFT)*

by

Atendolfo Pereda Bórquez

Roberto Lins de Carvalho

Series Editor: Michael F. Challis August, 1977

* Work partially supported by the brazilian government agency FINEP.

ABSTRACT: The basic concepts of the lambda calculus are defined. Two applications in the area of programming languages semantics are presented.

KEYWORDS: lambda expressions, programming language semantics

RESUMO: São definidos os conceitos basicos do lambda cálculo. Se apresentam duas aplicações na área da semântica de linguagens de programação.

PALAVRAS CHAVE: expressões lambda, semântica de linguagems de programação.

Contents

1- Introduction

The original motivation for the development of lambda calculus, was to give fundaments to logic. The reason for that necessity was an attempt to precisely define the concept of variable substitution in the predicate calculus. All the initial work was accomplished in the 1920-40 decades, by Church, Schönfinkel and others. Recently, after the works of Dana Scott [11,12,13] there has been a furthes development of the original ideas, mainly because of efforts to obtain a mathematical fondation for the programming language area, particulacy, the study of its semantics.

We can say that today, in that area (semantics of programming languages), there exist two main lines: [5].

a) The compiler oriented view
b) The Interpreter oriented

The second type methods can be subdivided in

b.1) Operational or constructive methods
b.2) The denotational or mathematical methods
b.3) Propositional methods

The knowledge of lambda calculus is important in the understanding of the operational and denotational methods (b.1 and b.2). For the second one, that knowledge is fundamental.

Our aim in these notes is to give the basic concepts of the lambda calculus, trying to explain and motivate through some examples of applications in the programming language area.

2- The lambda calculus

Let's consider the algebraic expression

$$x-y$$

as defining a function depending on one variable. That function can be a function f of the variable x or a function g of the variable y. One way to distinguish these two interpretations is to mark, in some form, the symbol acting the role of a variable in each case. The lambda calculus introduces the special symbol λ (the greek letter lambda) to do this marking, as a way of indicating the binding of an argument with the variable next to it.

According to that notation, the function f can be described as

$$f = \lambda x.(x-y)$$

The notation reinforces the role of x as the variable to be substituted by every argument for the function $\lambda x.(x-y)$, so

$$f(a) = \lambda x.(x-y)(a) = a-y$$

In the case of the function g, we have

$$g = \lambda y.(x-y)$$
$$g(a) = \lambda y.(x-y)(a) = x-a$$

Using these ideas, we have a systematic way to construct, for all the expressions in which the variable x occurs, a notation for a corresponding function of x. This notation can be extended to functions of two or more variables, in which the order of computation is important.

For example, let the following functions h and k be defined (in "conventional" notation) by

$$h(x,y) = x-y$$

and

$$k(y,x) = x-y$$

In our lambda notation the function h is represented by

$$h = \lambda x.\ \lambda y.\ (x-y)$$

and k by.

$$k = \lambda y.\lambda x.(x\text{-}y)$$

The order of computation (i.e. of substitution of variables by arguments) is indicated in each case by the order (from left to right) in which appear the variables adjoined to each 'λ'.

So

$$h(a) = \lambda x.\lambda y.(x\text{-}y)(a) = \lambda y.(a\text{-}y)$$

That is, the result of applying h to the argument a, is a new function of one variable, y.

Using the same argument a for k gives

$$k(a) = \lambda y.\lambda x.(x\text{-}y)\ (a) = \lambda x.(x\text{-}a)$$

we obtain a function depending on x.

The following function of three variables.

$$\lambda x.\lambda y.\lambda z.(x*y+z)$$

is different from the function

$$\lambda y.\lambda z.\lambda x.(x*y+z)$$

That can be seen, applying both functions to the same set of arguments, 2,4,3.

$$\lambda x.\lambda y.\lambda z.(x*y+3)\ 243$$

means, substitute the first argument, 2, for x, the second, 4, for y, and the third, 3, for z. So, the result is

$$2*4+3 = 11$$

The application of the second function give

$$\lambda y.\lambda z.\lambda x.(x*y+z)\ 243$$
$$3*2+4 = 10$$

It is clear from the examples, the binding of each argument with the corresponding variable.

2.1 - Basic Definitions

Initially, we introduce informally (*) the morphology of the lambda calculus.

(*) The formal treatment of this theory appears in the monograph A Review of the Church - Rosser Theorem for the λ-α-β-η Calculus" by R.L. de Carvalho.

Definition 2.1

Primitive objects: The symbols $x, x_0, x_1, \ldots, y, \ldots, z, \ldots$ belonging to a certain set of indentifiers, are primitive objects called variables.

Definition 2.2

A lambda expression is either an identifier denoting a variable, and is called a simple expression

or

if Y is a lambda expression and x is a variable, then

$$(\lambda x.\ Y)$$

is a lambda expression. The variable appearing immediately after the 'λ' symbol is called the bound variable, and Y is called the body of the lambda expression.

or

if X and Y are lambda expressions then (XY) is a lambda expression called the composite of the operator X and the operand Y.

If our set of variables is

$$\{x_0, x_1, \ldots, x_n, \ldots\}$$

we can form by example, the following lambda expressions:

x_0 is an identifier, denoting a variable, so, it is a simple expression, and thence a lambda expression.

$(\lambda x_0.x_0)$ in this case, the bound variable is x_0. The body is the lambda expression x_0.

$(\lambda x_1.(\lambda x_0.x_0))$ Bound variable: x_1
Body: $(\lambda x_0.x_0)$

$((\lambda x_0.x_0)(\lambda x_1.x_1))$ Here we have a composite, with the lambda expression $(\lambda x_0.x_0)$ as operator and the lambda expression $(\lambda x_1.x_1)$ as operand.

Definition 2.3

A lambda expression X is a sub expression of another lambda expression Y if:
either X occurs as the body of Y
or X occurs as the operator of Y
or X occurs as the operand of Y
or X is a subexpression of a subexpression of Y.

Example: if $Y \equiv (\lambda y.(y(\lambda x.x)))$

$(y(\lambda x.x))$ is a subexpression of Y (it is its body)

$(\lambda x.x)$ is a subexpression of Y because it is a subexpression of its body.

In the following definitions, the letters W,X,Y,Z represent an arbitrary lambda expressions.

Definition 2.4

An occurrence of a variable x in Y is bound in Y iff x occurs in a Z which is the body of Y, and Y has the form $\lambda x.Z$ or x occurs in a W which is the body of a subexpression $\lambda x.W$ of Y.
If the variable x is not bound, then we say that x is free.
A variable x is free in X (denoted $x \in_f X$) when x occurs free in X.

Example:

$(\lambda x.(xx))$ Both occurrences of x are bound

$(x(\lambda x.(xy)))$ The first occurrence of x is free, the second is bound

$(\lambda z.((\lambda y.(yx))(\lambda x.(xy))))$ z does not occur free or bound.

Definition 2.5

The substitution of each free occurrence of a variable x in Y by Z (denoted [Z/x]Y) is defined by the following rules.

i) $[Z/x]x \equiv Z$ if Y is the lambda expression x

ii) $[Z/x]y \quad y$ for all $y \not\equiv x$

iii) $[Z/x]WX) \equiv (([Z/x]W)([Z/x]X))$

iv) $[Z/x](\lambda x.W) \equiv \lambda x.W$

v) $[Z/x](\lambda y.W) \equiv \begin{cases} (\lambda y.[Z/x]W) & \text{if } y \not\equiv x \text{ and } y \notin_f Z \text{ or } x \notin_f W. \\ (\lambda z.[Z/x][z/y]W) & \text{if } y \not\equiv x \text{ and } y \in_f Z \\ & \text{and } x \in_f W \text{ (z is a new variable).} \end{cases}$

For example, if

$$Z \equiv \lambda z.(zz)$$

$[Z/x]x \equiv \lambda z.(zz)$

$[Z/x]y \equiv y$

$[Z/x](xy) \equiv (([Z/x]x)([Z/x]y))$

$\equiv ((\lambda z\ (zz))y).$

$[Z/x](\lambda x.(xy)) \equiv \lambda x.(xy)$

$[Z/x](\lambda y.(\lambda x(xy))) \equiv (\lambda y.[Z/x](\lambda x.(xy)))$

$\equiv (\lambda y.(\lambda x.(xy)))$

Definition 2.6 : α-rule

A change of the bound variable in a lambda expression X, of the form $\lambda x.Y$, is the replacement

$X = \lambda x.Y = \lambda y.[y/x]Y$ if $y \notin_f Y$

The meaning of the α-rule, or renaming rule, is that the bounds variables are irrelevants, that is, does not matter which is the variable used to indicate correspondence with arguments. For example, the meaning of the functions $\lambda x.(x+2)$ and $\lambda y.(y+2)$ is the same, applying the argument 3, both give the same result, 5. If we apply α-rule to $\lambda x.(x+2)$, with y as new bound variable, we obtain $\lambda y.(y+2)$, i.e.

$$\lambda x.(x+2) = \lambda y.[y/x](x+2) = \lambda y.(y+2)$$

Definition 2.7 β-reduction rule

A lambda expression X of the form $((\lambda x.Y)W)$ contracts to Z iff Z is the result of the replacement

$X = ((\lambda x.YW) = [W/x]Y = Z$

The meaning of the β-rule is the following: $(\lambda x.Y)$ is a function applied to the argument W. The result of the application of the function on W is obtained substituting W for each free ocurrence of x in Y.

Definition 2.8 η-rule

If $x \notin_f Y$, then X of the form $(\lambda x.(Yx))$, is replaced by Y, that is $(\lambda x.(Yx)) = Y$

The meaning of the η-rule is that applying $(\lambda x.(Yx))$ or Y to some argument W, is obtained the some result, i.e.

$$((\lambda x.(Yx))W) = YW$$

because x does not occur free in Y.

Any expression of the form $((\lambda x.Y)X)$ is called a β - redex and $[X/x]Y$ is its contractum.
A expression of the form $(x.(Yx))$ is called a η-redex (if x does not occur free in Y).

Definition 2.9

A lambda expression X is said to be in normal form iff X contains no sub-expressions that are redexes. If an expression W is obtained from X by a finite series of applications of the α,β and η rules, we say that X reduces to W (we denote an application of each rule by $\overset{=}{\alpha}>$, $\overset{=}{\beta}>$ and $\overset{=}{\eta}>$).
If the expression W so obtained is in normal form, then W is called a normal form of X.

Examples of application of the α-rule are:

$$(\lambda x.(yx)) \underset{\alpha}{=>} (\lambda z.(yz))$$

$$(\lambda x.(\lambda y.((xy)y))) \overset{=}{\alpha}> (\lambda z.(\lambda y.((zy)y)))$$

Examples of application of the β-rule are:

$$(\lambda x.\ x)y \overset{=}{\beta}> y$$

$$(x((\lambda x.\ x)(\lambda x.(xy)))) \overset{=}{\beta}> (x(\lambda x.(xy)))$$

An example of application of the η-rule is:

$$(\lambda x.((\lambda y.(yy))x)) \underset{\eta}{=>} (\lambda y.(yy))$$

An example of reduction of a lambda expression to normal form is:

$$(\lambda x.((\lambda y.(yx))z))v \xrightarrow{\beta} (\lambda y.(yv)z) \xrightarrow{\beta} zv$$

zv is the normal form of the given lambda expression.

Now, we show an example of a lambda expression without normal form:

$$((\lambda x.((xx)y))(\lambda x.((xx)y))) \xrightarrow{\beta}$$

$$(((\lambda x.((xx)y))(\lambda x.((xx)y)))y) \xrightarrow{\beta}$$

$$((((\lambda x.((xx)y))(\lambda x.((xx)y)))y)y) \text{ ... etc.}$$

so, the application of reduction by β-rules never ends!

In general, there exists more than one way to apply the reduction rules in a lambda expression. In the example

$$((\lambda x.(\lambda y.(yx))z)v)$$

we can apply the β-rule in the following form

$$(\lambda x((\lambda y(yx))z))v \xrightarrow{\beta} (\lambda x.(zx))v \xrightarrow{\beta} zv$$

In this case, both reductions gives the same normal form. These exists lambda expressions where some ways of applications of reductions never arrives to a normal form, and other terminates in a normal form, for example.

$$((\lambda y.(\lambda z.z))((\lambda x.(xx))(\lambda x.(xx))))$$

which is of the general form

$$((\lambda y.Y)W)$$

If we apply β-rule, using W as argument in Y, we obtain Y, that is, $(\lambda z.z)$, because there is not a free ocurrence of y in Y, $(\lambda z.z)$ is in normal form.

If we try to reduce the argument W, before its application, using β-rule, we obtain a never ending sequence of the form.

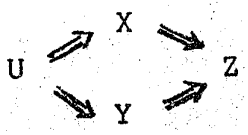
$$((\lambda x.(xx))(\lambda x.(xx))) \xrightarrow{\beta}$$

$$((\lambda x.(xx))(\lambda x.(xx))) \xrightarrow{\beta} \text{ ... etc.}$$

The following theorem says that if a lambda expression is reducible to a normal form, then every order of reduction that terminater yields that same normal form.

2.2 - Church - Rosser Theorem

If U => X and U => Y (=> means reduction using α, β and η rules), then there exists a Z such that X => Z and Y => Z.



The previous concepts are the fundamental ones in the theory of lambda calculus.

We will consider now some interpretations of special lambda expressions, developed in the works of Curry [4], Böhn, Gross [3] and others.

2.3 - Integer number representation

The usual way of representing non negative integers by means of lambda expressions is to associate with every number n the operator $\underline{n}$, such that, applied to any parameter f and g it iterates n times the application of f to g. That is

$$\underline{n}\ fg \equiv (\underbrace{f(f(f\ldots(fg)\ldots)))}_{n \text{ times}}$$

this is satisfied by the lambda expression

$$(\lambda x(\lambda y(\underbrace{x(x(x\ldots(xy)\ldots)))))}_{n \text{ times}}$$

For example

$$\underline{0} \equiv (\lambda x(\lambda y.(y)))$$

$$\underline{1} \quad (\lambda x(\lambda y(xy)))$$

$$\underline{2} \quad (\lambda x(\lambda y(x(xy))))$$

Now, having the representation of the positive integers, the following step is to define some of the elementary operations on numbers. Let $\underline{m}$, $\underline{n}$ be the representations of any two numbers. The sum of these two numbers is obtained using the lambda expression:

$$(((\lambda x(\lambda y(\lambda z(\lambda t((xz)\ ((yz)t))))))\underline{m})\underline{n})$$

For example, adding 2 plus 1, in this representation is obtained by

$$(((\lambda x(\lambda y(\lambda z(\lambda t((xz)((yz)t))))))\underline{2})\underline{1})$$

$$\overset{\beta}{\Rightarrow} ((\lambda y(\lambda z(\lambda t((\underline{2}z)((yz)t)))))\underline{1})$$

$$\overset{\beta}{\Rightarrow} (\lambda z(\lambda t((\underline{2}z)((\underline{1}z)t))))$$

$$\equiv (\lambda z(\lambda t(((\lambda x(\lambda y(x(xy))))z)(((\lambda x(\lambda y(xy)))z)t))))$$

$$\overset{\beta}{\Rightarrow} (\lambda z(\lambda t((\lambda y(z(zy)))((\lambda y(zy))t))))$$

$$\overset{\beta}{\Rightarrow} (\lambda z(\lambda t((\lambda y(z(zy)))(zt))))$$

$$\overset{\beta}{\Rightarrow} (\lambda z(\lambda t(z(z(zt)))))$$

$$\overset{\alpha}{\Rightarrow} (\lambda x(\lambda y(x(x(xy))))) \equiv \underline{3}$$

The result is the representation of 3.

The product of twonumbers can be defined as

$$(((\lambda x(\lambda y(\lambda z(x(yz)))))\underline{m})\underline{n})$$

Now, applying the representation $\underline{m}$ of a number to a second representation $\underline{n}$, consists in elevating the latter to an exponent equal to the first, that is

$$(\underline{m}\ \underline{n}) \equiv \underline{n}^{\underline{m}}$$

2.4 - Representation of logical constants and logical operators.

The logical constants can be chosen to be two-argument functions [16] each of which selects one of its arguments depending on its value: 'true' or 'false'. This choice is motivated by programming language expressions of the form:

$$\underline{\text{if}}\ \text{logical expression}\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2;$$

depending on the logical value of the logical expression, the statement S_1, will be selected, if 'true', or S_2, if 'false'. Then, the logical constant 'true' must act as a selector for the first element, S_1, in a list of two elements S_1 and S_2, 'false' will select the second element, S_2.

So, the representation for 'true' is

$$(\lambda x(\lambda y(x)))$$

and for 'false'

$(\lambda x(\lambda y(y)))$

The logical operations and, or and not are represented by the lambda expressions:

not $\equiv (\lambda x((x\text{'false'})\text{'true'}))$

and $\equiv (\lambda x(\lambda y((xy)\text{'false'})))$

or $\equiv (\lambda x(\lambda y((x\text{'true'})y)))$

3- Applications

The lambda calculus has been applied in different forms, by authors such as Böhm [2], Landin [6,7,8,9] Strachey [14,15] and others in their efforts to explain the meaning of the various aspects of programming language constructs.

In recent years, thanks to the work of Scott, a model for the lambda calculus has been evolved, so, in this way, new and solid mathematical aparatus is available for the investigation of programming language semantics.

In this section we revise, in an informal manner, two works in this direction. We will examine the SECD machine of Landin [7,9] and a recent work by K.Abdali [1].

3.1 - The SECD Machine

One of the points of view used to define the semantics of programming languages is the interpreter oriented view. In it, the meaning is given in terms of the transformations that are specified by the syntactically valid programs in the language. Specifically, there must be:

- a formal description of the universe of discourse for the language.
- a set of rules that indicate the way in which the basic expressions in the language can be combined and giving the outcome of each combination as a function of its component terms.

One of the methods of the interpreter oriented view is the operational or constructive method.The operational method defines a language giving:

- The definition of an abstract "machine state" S that contains the essential information about the progress of the process represented for each program in the language.
- Specification of the effect of the constructs in the language on the states; that is, the specification of a transition function, from one state to another.

In 1964, Landin developed the SECD machine, as a method to define the meaning of the evaluation of "Applicative expressions" [7]. Two years later [8], he published a new version of the machine, called the "Sharing Machine", used to define the semantics of ALGOL-60, but this time using the Sharing Machine as an interpreter for the "code" generated by a "compiler" translating from ALGOL-60 to applicative expressions of a more complex type.

The applicative expressions (AE) have as abstract syntax, the syntax of the lambda expressions (λE), so:

as AE is

either simple and is an identifier

or is a λE and has a bound variable which is an identifier and a body which is an AE.

or is a Compound and has an operator which is an AE and an operand which is an AE.

All the AEs have a "value", which is either a number, or a function, etc. More precisely an AE X has a value relative to some information which gives a value for each free identifier in X. That background information will be called the Environment relative to which the evaluation is conducted.

The Environment is considered as a function that associates with each identifier a value.

The value of X relative to the environment E will be denoted:

$$\underline{val}\ (E)(X)$$

The machine uses the environment for assigning values to identifiers.

Consider the expression $(\lambda x.(ax-b))c$

for the machine to be able to evaluate ax-b, the environment must contain the values of a,b and x. The value of x can only be supplied at the moment when the function is applied to an argument. In the machine, the value of an expression such as $(\lambda x.(ax-b))$ is represented by the expression itself, together with the environment including the values of a,b,-. This collection of information is called a closure.

The closure is formed by:

> a control part which is an expression list, a bound variable (an identifier), and an environment part which is a list of pairs "identifier-object".

The machine works using a stack. In order to evaluate an AE, in a certain environment, the operand is evaluated first and its value is loaded onto the stack. Then the operator is evaluated, and a closure is loaded on to the stack. The closure is then applied to its argument.

The structure of the "state" of the machine for evaluating an AE is the following:

- A Stack,S — which is an object-list
  Environment,E — a list of pairs identifier-object
  Control,C — an expression-list
  Dump,D — a state.

The SECD initial state is:

the stack S with the null list ('()'),
the environment E with the null list,
the control C with the AE to be evaluated,
the dump D with an initial dump Do.

To define the transition function of the machine we will use the following predicates, functions, and selectors on lists :

predicates:

null(arg) : test if arg is null
id(arg) : test if arg is an identifier
λexp(arg) : test if arg is a lambda expression
eq(alg_1,arg_2): test if arg_1 is equal to arg_2
comp(arg) : test if arg is a compound
closure(arg) : test if arg is a closure

functions:

apply(arg_1,arg_2) : apply the (primitive function denoted by) arg_1 to arg_2.
constclosure(arg_1,arg_2,arg_3) : construct a closure with the arguments arg_1,arg_2 and arg_3.
h(arg): obtain the head of arg
t(arg): obtain the tail of arg.
(arg_1:arg_2): append arg_1 to arg_2

Selectors:

envp(arg) : selects the environment part of a clasure.
rand(arg) : selects the operand part of arg
orator(arg): selects the operator part of arg
body(arg) : selects the body part of a lambda expression
bound(arg) : selects the bound variable of a lambda expression.

The transition function is

```
Transform (S,E,C,D) =
        if null(C) then
           (((h(S)):S'),E',C',D') where (S',E',C',D')=D
        if id(h(C)) then
           (((val(E)(h(C))):S),E,t(C),D)
        if λexp(h(C)) then
           (((constclosure(body(h(C)),boundv(h(C)),E):S),E,t(C),D)
```

```
if eq(h(C),'apply')and closure(h(S)) then
   ('()',((boundv(h(S)),h(t(S))):envp(hS)),body(hS),
                  (t(t(S)),E,t(C),D))

if eq(h(C)),'apply') and ¬closure(h(S))then
   ((apply(h(S),h(t(S))):t(S)),E,t(C),D)

if comp(h(C))then
   (S,E,(rand(h(C)):(rator(h(C)):('apply':t(C)))),D)
```

The ending conditions are:
the result (if any) in S,
the environment with the null list,
the control with the null list,
and the dump with Do.

Now, we give an example of evaluation using the SECD machine. Let the AE be

(((λf.(λx.(fx)))square)2)

In Fig.1 is presented the successive states of the SECD machine during the computation. We use the letter A to represent the 'apply' metasymbol, '()' is the null list.

Following the conventions of the original work the head of the stack will be at the right side, and the head of the control at the left side.

State	Stack	Environment	Control	Dump.
1	( )	( )	(((λf.(λx(fx)))sq)2)	Do
2	( )	( )	2,((λf(λx(fx)))sq),A	Do
3	2	( )	((λf(λx(fx)))sq),A	Do
4	2	( )	sq,(λf(λx(fx))),A,A	Do
5	2,sq	( )	(λf(λx(λx(fx))),A,A	Do
6	2,sq,[λx(fx),f,( )]	( )	A,A	Do
7	( )	(f=sq)	λx(fx)	(2,( ),A,Do)
8	[(fx),x,(f=sq)	(f=sq)	( )	(2,( ),A,Do)
9	2,[(fx),x,(f=sq)]	( )	A	Do
10	( )	(x=2, f=sq)	(fx)	(( ),( ),( ),Do)
11	( )	(x=2, f=sq)	x,f,A	(( ),( ),( ),Do)
12	2	(x=2, f=sq)	f,A	(( ),( ),( ),Do)
13	2,sq	(x=2, f=sq)	A	(( ),( ),( ),Do)
14	4	(x=2, f=sq)	( )	(( ),( ),( ),Do)
15	4	( )	( )	Do

Fig. nº1 - States of the SECD during the evaluation of (((λf(λx(fx)))sq)2)

In the initial state, the AE is in the control list, with the head to the left. To create a new state, the head of the control is tested, and happens to be a compund, with operator

$$((\lambda f(\lambda x(fx)))sq)$$

and operand 2

So, the new control list is obtained by appending the operand of the head of control, 2, to the operator, that is, the AE above, to the metasymbol "apply", to the tail of the previous control C, which in this case, is the empty list. The rest of the components of the state remain the same.

Another point of interest is the change from state 5 to state 6. In state 5, the head of the control list is a lambda expression. So, a new state is created with, a new stack, where the closure

$$[\lambda x(fx), f, ( )]$$

was appended to the previous stack S. The closure is formed by

$\lambda x(fx)$, the body of the head of the previous control C.

f the bound variable of the head of the previous control C.

and ( ) the empty list, thatis, the previous environment.

The environment and the dump remain the same, the control list is now the tail of the previous control list, i.e., 'apply', 'apply', ( ).

Certain well formed expressions of the lambda calculus are not computed completely by the SECD machine [10]. For example, if the following compound is submitted as input:

$$((\lambda x(\lambda y(x(xy))))(\lambda x(\lambda y(x(xy)))))$$

That is, the representation of ($\underline{2}$ $\underline{2}$), (it must give 2^2), the machine goes through the following states.

S	E	C	D
( ) ( ) [(λy(x(xy))),x,( )]	( ) ( ) ( )	((λx(λy(x(xy))))(λx(λy(x(xy))))) (λx(λy(x(xy)))),(λx(λy(x(xy)))),A (λx(λy(x(xy)))),A	Do Do Do
[(λy(x(xy))),x,( )], [(λy(x(xy))),x,( )]	( )	A	Do
( )	(x=[(λy(x(xy))),x,( )])	(λy(x(xy)))	((),(),(),Do)
[(x(xy)),y,(x=[(λy(x(xy))),x,()])]	(x=[(λy(x(xy))),x,( )])	( )	((),(),(),Do)
"	( )	( )	Do

Figura 2.

The machine arrives to a final state, but without the result in the stack.

This situation is obviated by modifying the SECD machine by adding two new components:

O: an output symbol string

j: unique name counter

The output symbol string is built-up by concatenation during a computation, the unique name counter generates new bound variables b_j. Thus is obtained the so called modified SECD, MSECD.

The state of the MSECD is defined to be the values of its six components

$$(S,E,C;D,O,j)$$

3.2 - The lambda calculus model of programming languages of S.K. Abdali

The work of S.K. Abdali [1] presents a correspondence between the constructs of programming languages and the lambda calculus. The correspondence is obtained using functional interpretations of these constructs.

The concepts of program variable and assignment are accounted for in terms of the concepts of mathematical variables and substitution respectively. Using pure lambda calculus, all its properties, in particular the Church-Rosser property, are valid in the model. In the model, the programs are translated into lambda-expressions, not interpreted by a lambda calculus interpreter (note the difference with SECD).

Thus, program semantics are completely reduced to the lambda calculus semantics.

The language modelled is ALGOL-60.

The general idea behind the representation is the following:

Consider a given statement S of a program. Let $(v_1,\ldots,v_n)$ be the environment of S,i.e., the values at the moment of execution of S, of the n declared and active variables of the program, and let F denote the section of the program following S and extending all the way to the program end. This F is called the "program remainder" of S. The two parts of the program, one

consisting of F alone, and the other composed of S and F together may be interpreted as two functions ϕ and ϕ' respectively, of the arguments $v_1,\ldots,v_n$:

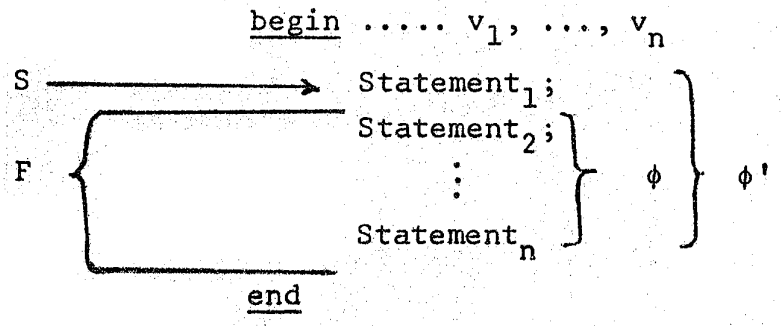


Fig.3.

The effect of interposing S in the program is to transform ϕ into ϕ'. σ is taken as the representation of S where the function operator σ is given by

$$(i)\quad (\sigma(\phi))(v_1,\ldots,v_n) = \phi'(v_1,\ldots,v_n)$$

which accomplishes the transformation. The equation (i) may be written as a reduction in lambda calculus,

$$(ii)\quad \sigma\phi\ v_1\ldots v_n \Rightarrow \phi'\ v_1\ldots v_n$$

If the right-hand-side of (ii) in can be expressed in terms of ϕ, v_1, ..., v_n, and possibly some constants, then (ii) can be taken as the definition of σ as a function of formal arguments ϕ, v_1, ..., v_n.

The domains of the arguments v_1, ..., v_n are the values of the corresponding program variables; the domain of ϕ consists of the program remainder considered as a function.

If a statement S is modelled by the lambda expression σ then the execution of S is simulated by the reduction of the lambda expression

$$\sigma\ \hat{\phi}\hat{v}_1\ldots\hat{v}_n$$

in which the symbols $\hat{v}_1\ldots\hat{v}_n$ denote the lambda calculus representations of the values of the corresponding variables immediately prior to the execution of S, and $\hat{\phi}$ denotes the representation of the program remainder of S.

So, the key step in modelling a programming language is to define a suitable equation of the form (ii) for each construct in it.

By example, the following is the representation rule for the assignment statement:

$$\{v_i := e\} \text{ in the environment } (v_1,\ldots,v_n) \equiv$$
$$(\lambda\phi(\lambda v_1(\ldots(\lambda v_n(\ldots(\phi v_1)\ldots v_{i-1})\{e\})v_{i+1})\ldots V_n))\ldots)))$$

using a shorter notation

$$= \lambda\phi\lambda v_1\ldots\lambda v_n.\phi v_1\ldots v_{i-1}\{e\}v_{i+1}\ldots v_n$$

So, the translation of the following program

```
                begin integer    x,y;
1.-                   x:= 2;
2.-                   y:= x+19;
                        begin  integer  z,y;
3.-                             y:= x-5
                      end
                end
```

to lambda calculus notation is:

1.- $\lambda\phi.\lambda x\lambda y.\phi\underline{2}y$

2.- $\lambda\phi\ \lambda x\lambda y.\phi x(+x\underline{19})$

3.- $\lambda\phi\lambda z\lambda y^1\lambda x\lambda y^0.\phi z(-x\underline{5})xy^0$

here, $\underline{n}$, with n integer, denotes the lambda calculus representation of that integer. A prefixed expression like (+x $\underline{19}$) denotes the lambda calculus representation of that expression. The indices are used to denote different variables using the same identifier.

The representation rule of the compound statement is:

$$\{\underline{\text{begin}}\ S_1;S_2;\ldots;S_n\ \underline{\text{end}}\} \equiv \lambda\phi.\{S_1\}(\{S_2\}(\ldots(\{S_n\}\phi)\ldots))$$

where the S_i are statements.

The following are two examples of compound statement representations.

i) begin

x:=2; $S_1 \equiv \lambda\phi\lambda x\lambda y.\phi\ \underline{2}y$

y:=x+3; $S_2 \equiv \lambda\phi\lambda x\lambda y.\phi x(+x\underline{3})$

x:=y+x $S_3 \equiv \lambda\phi\lambda x\lambda y.\phi(+yx)y$

end

{begin x:=2;y:=x+3;x:=y+x end} $\equiv \lambda\phi.S_1(S_2(S_3\phi))$

ii) begin

y:=5; $S_1' \equiv \lambda\phi\lambda x\lambda y.\phi\ x\underline{5}$

x:=y+2 $S_2' \equiv \lambda\phi\lambda x\lambda y.\phi\ (+y\underline{2})y$

end

{begin y:=5; x:=y+2 end} $\equiv \lambda\phi.\ S_1'(S_2'\phi)$

Using both previous examples we can show another interesting point in the model, i.e. the derivation of equivalence of programs.

The equivalence of the compound statements (i) and (ii) is derived by showing that the lambda expressions of (i) and (ii) reduce to the same normal form. (This normal form represents the"simplest" version)

Applying the lambda calculus reduction rules to the representation of (i) we obtain :

$\lambda\phi.\lambda\phi\lambda x\lambda y.\phi\ \underline{2}y\ (S_2(S_3\phi))$

$\overset{\beta}{>}$ $\lambda\phi\lambda x\lambda y.(S_2(S_3\phi))\ \underline{2}y$

$\equiv$ $\lambda\phi\lambda x\lambda y\lambda\phi\lambda x\lambda y.\phi\ x(+x\ \underline{3})(S_3\phi)\ \underline{2}y$

$\overset{\beta}{>}$ $\lambda\phi\lambda x\lambda y\lambda x\lambda y.(S_3\phi)x(+x\ \underline{3})\ \underline{2}y$

$\overset{\beta}{>}$ $\lambda\phi\lambda x\lambda y\lambda y.(S_3\phi)\ \underline{2}(+\underline{2}\ \underline{3})y$

$\equiv$ $\lambda\phi\lambda x\lambda y\lambda y.(S_3\phi)\ \underline{2}\ \underline{5}y$

$\overset{\eta}{>}$ $\lambda\phi\lambda x\lambda y.(S_3\phi)\ \underline{2}\ \underline{5}$

$\equiv$ $\lambda\phi\lambda x\lambda y\lambda\phi\lambda x\lambda y.\phi(+yx)y\phi\ \underline{2}\ \underline{5}$

$\overset{\beta}{>}$ $\lambda\phi\lambda x\lambda y\lambda x\lambda y.\phi(+yx)y\ \underline{2}\ \underline{5}$

$\overset{\beta}{>}$ $\lambda\phi\lambda x\lambda y\lambda y.\phi(+y\ \underline{2})y\ \underline{5}$

$\overset{\beta}{>}$ $\lambda\phi\lambda x\lambda y.\phi(+\underline{5}\ \underline{2})\ \underline{5}$

$\equiv$ $\lambda\phi\lambda x\lambda y.\phi\ \underline{7}\ \underline{5}$

applying the rules of reduction to the representation of (ii)

$\lambda\phi\lambda\phi\lambda x\lambda y.\phi\ x\underline{5}\ (S_2^1\phi)$

$\overset{\beta}{=}$ $\lambda\phi\lambda x\lambda y.(S_2^1\phi)\ x\underline{5}$

$\equiv$ $\lambda\phi\lambda x\lambda y\lambda\phi\lambda x\lambda y.\phi(+y\underline{2})y\ \phi\ x\underline{5}$

$\overset{\beta}{=>}$ $\lambda\phi\lambda x\lambda y\lambda x\lambda y.\phi(+y\ \underline{2})\ yx\ \underline{5}$

$\overset{\eta}{=>}$ $\lambda\phi\lambda x\lambda y\lambda y.\phi(+y\ \underline{2})y\ \underline{5}$

$\overset{\beta}{=>}$ $\lambda\phi\lambda x\lambda y.\phi(+\underline{5}\ \underline{2})\ \underline{5}$

$\equiv$ $\lambda\phi\lambda x\lambda y.\phi\ \underline{7}\ \underline{5}$

so, both compound statements have the same "meaning", and a "minimal" equivalent compound statement is

```
begin
    x:=7
    y:=5
end
```

The examples given before, are for the simplest statements. In his work, Abdali presents in addition, reduction rules for blocks, conditional statements, input-output, programs, iteration, jumps and procedures.

As was observed, the potential of the model in studying the properties of programs is very interesting - convergence , correctness, and equivalence and in performing useful program transformations - such as program simplification, and optimization.

References

[1] Abdali S.K. "A lambda Calculus model of Program - ming languages", part I and II. Journal of Computer languages vol.1, 1976.

[2] Böhm C. "The CUCH as a formal and description language" in F.L.D.L. Steel (Ed.) North-Holland P.C., 1966.

[3] Böhm C., Gross.W. "Introduction to the CUCH" in Automata Theory, Caianiello (Ed). Academic Press, 1966.

[4] Curry H., Feys R. "Combinatory Logic", North-Holland P.C., 1958.

[5] Donahue J.D. "Complementary definitions of Programming language semantics". Lecture Notes in Computer Science, vol 42, Springer V., 1976.

[6] Landin P.J. "A correspondence between ALGOL-60 and Church's lambda notation"., C.A.C.M, vol.8, nº 2, vol.8, nº 3, 1965.

[7] Landin P.J. "The mechanical evaluation of expressions". Computer J.,6, 1964.

[8] Landin P.J. "A formal description of ALGOL-60" in F.L.D.L. Steel (Ed.). North-Holland P.C., 1966.

[9] Landin P.J. "A λ-calculus approach" in Advances in Programming, Fox (Ed.), Pergamon Press, 1966.

[10] Mc Gowan C.L. "The correctness of a modified SECD machine" Second ACM Symposium on theory of Computing, 1970.

[11] Scott D. "The lattice of Flow-diagrams" in Semantics of Algorithmic languages, Engeler (Ed), Springer notes in Mathematics, vol. 188, 1971.

[12] Scott D. "Mathematical concepts in programming language semantics" Proceedings SJCC 40, AFIP press, 1972

[13] Scott D., Strachey C. "Toward a mathematical semantics for computer languages", in Computer and Automata, Fox (Ed.), John Wiley, 1972.

[14] Strachey C. "Toward a formal semantics" in F.L.D.L. Steel (Ed.), North-Holland EC., 1966.

[15] Strachey C. "The varieties of programming language" Technical monography PRG-10, Oxford U.C.L., march-1973.

[16] Wegner P. "Programming languages, Information Structures and machine Organization". McGraw Hill, 1971.