DERIVED SEMANTICS FOR SOME PROGRAMMINCT LANGUAGE CONSTRUCTS

Peter Henderson

Computing Laboratory, University of Newcastle upon Tyne. England.

Some familiar programming language constructs are described and an implementation for a hypothetical computing device is given. The semantics of the object language are described as mappings of the internal states of this device. By compounding the sequence of mappings corresponding to the translation of a source phrase, the semantics of the source phrase can be derived as a complex mapping. The recursive nature of the syntax of the source 1anguage requires an inductive approach to this derivation. If the derived semantics are acceptable then the implementation can be regarded as correct in an informal sense.

1. Introduction

McCarthy and Painter (8) prove the correctness of a compiler for simple arithmetic expressions according to a definition of correctness first defined by McCarthy (7). By means of an abstract analytic syntax for the source language they give a functional description of its semantics. Sinn larly the semantics of an obj ect]anguage are specifled and then the rules governing a translation from the source language to the object 1anguage are defined. It is the correctness of tlus translator which is in question. The proof takes the form of showing that the translated source phrase leaves the value predicted by the functional description of the source language semantics, in the accumulator of the obj ect machine.

There are strong parall els between the semantic description and the translation of a source language. Indeed, semantic drscription is accomplished by translation into a semantic object language which is usually of a functional nature, as opposed to the command structure of the machine language of an object machine. Conversely, the formal description of a translator, mapping the source language into such a procedural 1anguage is an adequate description of the source language semantics, given a description of the object J anguage semantics. This point is made by Virth (10).

In this paper we shall be concerned with an approach to imp]ementati on correc tness very similar to McCarthy and Painter, except that we make no a priori description of the source 1anguage semantics and consequently no definition of correctness. Rather, we shal1 define a procedural object language and a translator mapping the source language into it. We shall derive from these definitions a functiona1 description of the source language semanti cs. Correctness is then involved with the acceptance of this functional description as truly representing the meaning informally attributed to the source language. This is no less formal, except possibly in order of presentation, than the technique of McCarthy and Painter. Where those authors require acceptance of a definition of source language semantics a priori, this author requires the same a postiori.

The mam results of this paper are taken from Henderson (3) where a programming language is presented, analysed and a functional description of its semanti cs derived. The techniques are presented there as a language design tool , since this is how they were developed. As such they are sewrely critical of irrational source language and implementation features. The language ALEPH, which is based on ideas from many current high level languages is quite sophisticated, in that it allows for recursively activated functions and has a block structure within the eonfines of which one may declare storage for arrays. Notably, however, ALEPH does not contain an explicit control transfer (goto) facility. It was possible to derive a semantic description of the 1anguage up to but not including the use of vectors. The probJem here was that of "sha ri ng" of storage whi eh had been identified by Land in *(6)* . In this paper, three of the more interesting results wilJ be described to illustrate the techniques used. There is no doubt that the constructs chosen are the simpler ones derived in Henderson (3). It i s hoped to publish other results, those concerning functions in particuJ ar, elsewhere.

2. Some programming 1anguage constructs

It is not necessary to describe the whole of a programming language in detail in order to discuss some of the constructs available in it. It is, however, judicious to describe a l i t t l e more than is formally necessary in order to cuJ11vato unde rstandi ng. Consider then a programming language in which a program is constructed from phrases ralled expressions. Each expression is defined syntactically as a combinntion of symbols and phrases, some of which may be expressions. We shall not be concerned here with phrases other than express]ons. For reasons of expediency in semantic description we do not distinguish the concept, familiarly known as a statement, in our language. Rather we allow an expression to he evaluated solely for the side-effect it will have. This saves us, for example, having to define separately the concept of conditional statement and conditional expression. In the case of constructs more frequently associated with a side-vffect than a value, such as assignment and iterating statements,

conventional and if possible, convenient values will be designated for them. An assignment statement may yield as its value, the value assigned, for example. This sort of concept is not new, it appears in Wirth (10) and others.

The three constructs we wish to consider can be defined as follows.

1. There are certain basic expressions, which we shall not specify, here, which perform primitive computations of a desirable nature. For example

   x:= y :=y + 1 $p^2 + q^2 < r^2$

2. If e_1, e_2, ..., e_k are expressions, then so is

   begin e_1 ; e_2 ; ... ; e_k end

3. If e_1, e_2, e_3 are expressions, then so

   if e_1 then e_2 else e_3

4. If e_1 and e_2 are expressions, then so is

   while e_1 do e_2

To define what each of these constructs means, it is necessary to consider what we mean by value and side—effect. We assume that the computation expressed by a program manipulates the elements of some domain, which elements we shall call values. Each expression, therefore specifies the computation (assuming it terminates) of a value from this domain. We assume this computation takes place in an environment consisting of a collection of variables, each of which may or may not possess a value. When an expression is evaluated it may or may not produce a value. When an expression is evaluated it may alter the values associated with some of these variables. This effect is known as the side-effect (of the evaluation). In particular if the effect is to leave the variables unchanged, then it is said to be a null side-effect. A subset of the domain is distinguished and called zero, where we assume it is possible to test a value for membership of this subset.

We shall see that this predicate is used to construct conditional computations.

1. The value and side-effect of each basic expression is assumed to be defined.

2. The value and side-effect of evaluating

   begin e_1 ; e_2 ; .... ; e_k end

   is just the value and side-effect of evaluating e_k in the environment produced by evaluating e_1,..., e_{k-1} in order from left to right.

3. The value and side-effect of evaluating

   if e then e_2 else e_3

   is the value and side-effect of evaluating one of e_2 or e_3 in the environment produced by evaluating e_1, according to whether the value of e_1 is in zero or not. If the value of e_1 is in zero then e_3 is selected, otherwise e_2 is selected.

4. The side-effect of evaluating

   while e_1 do e_2

   is thp side-effect of evaluating e_1 and e_2 alternately, beginning and ending with e_1, until thr value of e_1 is first in zero. The value of

   while e_1 do e_2

   is the value produced by e_1 when it is last evaluated according to the above scheme, or 0 (which is some element of zero) if e_2 is not evaluated, that is when e_1 yields an element of zero when first evaluated.

We shall not attempt to be more precise about this informal semantic description here. The nature of zero is confusing. To clarify, if the domain is the direct union of integers and booleans then zero may have the single element false. However, we do not wish to confine these results to such a choice of domain.

3. An implementation for a hypothetical machine

Let us consider a computing device (not unlike a current day machine in which core and registers are used in a special way), which includes an accumulator T and a stack P. The names T and F were derived from the "top" element and the "first stacked" element in a single stack structure. The basic operations which this device can perform may be specified simply in terms of the accumulator T and the stack F as follows.

T↑F place the current value of T on top of the stack F, leave T unchanged.

TF replace the value of T by the value at the top of the stack. Remove this value from the stack.

pop F Remove the top value from the stack.

T←O replace the current value of T by 0.

The basic tests which the device can perform include:

(T=O)? test if the current value of T is in zero.

A program for this device is a graph with a single entry and a single exit.

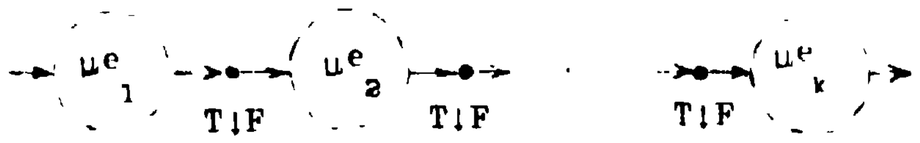
The nodes of the graph are of three kinds.

1. Command: single entry, single exit, labelled with a basic operation.
2. Test: single entry, double exit, labelled with a basic test and with the exits marked YES and NO.
3. Join: double entry, single exit, unlabelled.

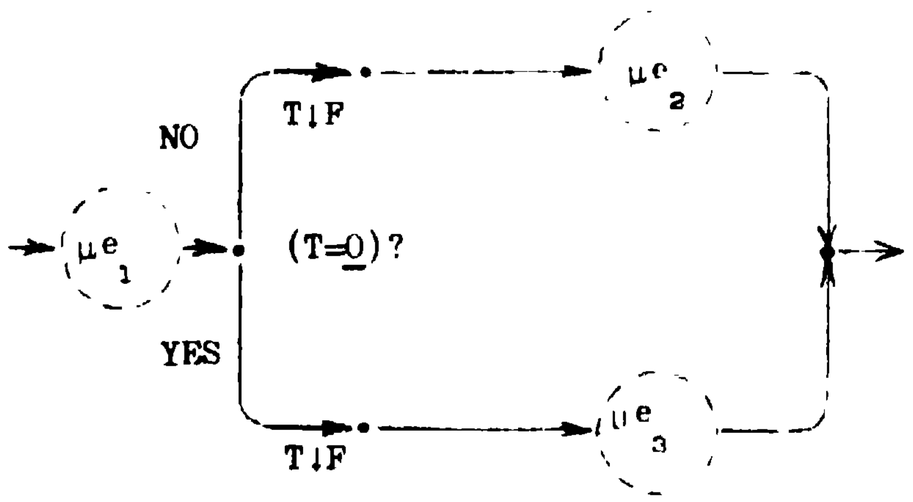
Such a program forms a flow diagram in a fairly obvious sense. We are now in a position to give the translations of our three principal expressions. Square brackets are used throughout merely to indicate structure in a (meta) expression. Source expressions will be used autonomously.

Given an expression e in the source language, we denote by μe its translation to a program for our hypothetical computing device. In displaying this program as a graph we encircle the (unspecified) subgraphs for component expressions with a dotted line.

1. $\mu[\underline{\text{begin}}\ e_1\ ;\ e_2\ ;\ \ldots\ ;\ e_k\ \underline{\text{end}}]$



2. $\mu[\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3]$



3. $\mu[\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2]$

The assumption, which we shall establish inductively in section 5, in that the translation of an expression is a program which:

1. Stacks the current value of T;
2. otherwise leaves F unaltered;

and 3. leaves the value of the translated expression in T.

The program may of course have an effect upon other, as yet unspecified, components of the device.

4. Notation for the model

It is necessary to construct a model of our hypothetical computing device in order to analyse the programs which we have constructed for it. The notation is based on Strachey (1964), Landin (1963) and Curry and Feys (1958) and other papers already cited.

If f is a function and x an argument within the domain of f, we denote by fx the result of applying f to this argument. Application associates to the left, thus $fx_1x_2\ldots x_n$ denotes $(\ldots((fx_1)x_2)\ldots x_n)$. The function "dot" product $f.g$, where f and g are functions over appropriate domains is defined by $\lceil f.g\rceil x = f\lceil gx\rceil$. The function "dot" product has a lower binding power than application and hence $\lceil fx.gy\rceil z$ is interpreted as $\lceil fx\rceil\lceil gyz\rceil$. Note that this product is associative and we shall write $\lceil f.g.h\rceil x$ for $f\lceil g\lceil hx\rceil\rceil$.

By means of the notation of the λ-calculus we may write the definition of f.g as $f.g = \lambda xf[gx]$. In a λ-expression the λ is followed immediately by the bound variable and then the body which extends as far to the right as is consistent with the bracketing. The λ-expression, λxA denotes the function, whose value for argument B, denoted by $[\lambda xA]B$, is obtained by evaluating the expression obtained when B is substituted for all free occurrences of x in A. We shall assume that B is evaluated before substitution and that any clash of variables (if the result of B contains any λ-expressions with free variables) is catered for by renaming of bound variables.

A particular function which we shall have need of is the following

$$Cz = \lambda x\lambda yx \quad \text{if } z \text{ is in } \underline{\text{zero}}$$
$$\lambda x\lambda yy \quad \text{if } z \text{ is not in } \underline{\text{zero}}$$

Note that Cuvw selects v if u is in zero and w if it is not. In general a function f can be defined by a scheme

$$f = \epsilon$$

where ϵ is a combination of λ-expressions and previously defined functions. If, however, ϵ contains an occurrence of f then we denote by

$$Y[\lambda f\epsilon] \quad \text{or simply} \quad Y\lambda f\epsilon$$

the solution of f = e, if it exists. Y is called the paradoxical combinator by Curry and Feys (2) and thp fixed-point operator by Landin *(5)*. Effoe lively it allows us to name the function f independently of the letter used to designate it.

Finally, let us introduce some notation for stacks, Λ denotes the empty stack and if S is thp stack with the n elements $S_1, S_2, \ldots S_n$ we vritc

$$S = (S_1, (S_2, \ldots (S_n, \Lambda)\ldots))$$

We denote by S^+ the top element *S* and by S^- the stack remaining when S is removed.

$$(S_0,S)^+ = S_0$$
$$(S_0,S)^- = S$$
$$(S^+,S^-) = S$$

When we consider the hypothetical computing device, there are various elements which can alter as the result of executing a command. The accumulator and the stack can alter, since ail the basic commands introduced here refer to them directly. The rest of the elements that can alter we shall call collectively the (source) environment. The environment is effec(lvely a data structure which includes information about the values of all currently declared variables. If we denote the (source) environment by v we can consider the object—environment *F* to be a triple

$$F = [t,f,v]$$

where t and f denote the current values of the accumulator and the stack respectively. The notion F is what McCarthy (7) refers to as a state vector. What we have done is simply to impart to it a little more structure.

The semantics of the object langua ge can now be defined as mappings on the object-environment as follows.

$$[t,f,v] \xrightarrow{T^{\uparrow}F} [t,(t,f),v]$$
$$[t,f,v] \xrightarrow{T\downarrow F} [f^+,f^-,v]$$
$$[t,f,v] \xrightarrow{popF} [t,f^-,v]$$
$$[t,f,v] \xrightarrow{T\leftarrow 0} [0,f,v]$$

By compounding the effect of these basic object-environment mappings we can derive the mapping corresponding to a program for our hypothetical computing device.

5. Derivation of semantics for the three constructs

The recursive nature of the syntax given for the expressions in section 2 necessitates an inductive approach to derivation for an expression. Given an expression e constructed from expressions $e_1, e_2, \ldots, e_k$ we shall establish a lemma stating that e had a certain property under the assumption that $e_1, e_2, \ldots e_k$ have that property. In Henderson (3) by an argument using structural induction, it is shown that for a certain choice of expression types, that such a lemma is sufficient to establish the property absolutely for e. The induction principle used is similar to that required by McCarthy and Painter and explained fully by Burstall (1).

The property which concerns us in this paper we shall call property A.

property A: the expression e is said to possess property A, written A[e], If the result of evaluating e in the object-environment F = [t,f,v] is the object-environment F' = [t',(t,f),'] where t' = gv and v' = hv for some functions g and h, determined by the expression e.

We see that in particular the original value of t is saved and the new values of the accumulator and environment depend only on the old value of the environment. (Note we mean source environment). The function g is called the value function associated with e and h is the associated side-effect function.

We shall assume that there exist functions cr and ill such that given o, as in the definition of property A, we have

$$\mu e = g \quad \Vdash e = h$$

Thus we can write

[t,f,v] UP [Wev, (t,f), Illevl since A[e] to show the object-environment mapping due to an expression with property A. Note that e and Ie are the functions which determine the semantics of e. We are now in a position to establish the lemmas which we should use in an argument of correctness of the implementation of the three expressions introduced in section 2.

Lemma 1 If the expressions $e_1, e_2, \ldots, e_k$ possess property A then so does the expression

begin e_1 ; e_2 ; ... ; e_k end

proof We shall prove by induction on the index k, that if

$$\beta_k = \mu[\text{begin } e_1 ; e_2 ; \ldots ; e_k \text{ end}]$$

then

$$[t,f,v] \xrightarrow{\beta_k} [[\mu e_k . \Vdash e_{k-1} . \ldots . \Vdash e_1]v, (t,f), [\Vdash e_k . \ldots . \Vdash e_1]v]$$

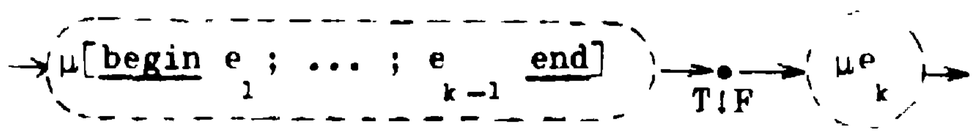
For if k = 1 we have $\mu[\text{begin } e_1 \text{ end}] = \mu e_1$ and hence

$$[t,f,v] \xrightarrow{\beta_1} [\mu e_1 v, (t,f), \Vdash e_1 v]$$

since A[e_1]

which establishes our hypothesis for k = 1.

The translation β_k for k>1, given in section 3 can be written



which is

Hence

$\lceil t,f,v\rceil \xrightarrow{\beta_{k-1}} [\lceil \varphi e_{k-1} . \psi e_{k-2} . \ldots . \|e_1\rceil v,$

$(t,f), \lceil \psi e_{k-1} . \ldots . \psi e_1\rceil v]$

by the inductive hypothesis

$\xrightarrow{T\downarrow F} [t,f,[\|e_{k-1} . \ldots . \psi e_1]v]$ by defn.

$\xrightarrow{\mu e_k} \lceil \varphi e_k \lceil [\|e_{k-1} . \ldots . \psi e_1]v],$

$(t,f), \|e_k \lceil \psi e_{k-1} . \ldots . \|e_1\rceil v]]$ since $A\lceil e_k\rceil$

$= [\lceil \varphi e_k . \|e_{k-1} . \ldots . \|e_1\rceil v,$

$(t,f), \lceil \|e_k . \ldots . \|e_1\rceil v]$

This completes the induction and shows that begin e_1 ; ... ; e_k end possesses property A with

φ[begin e_1 ; ... ; e_k end]

$= \varphi e_k . \|e_{k-1} . \ldots . \|e_1$

and $\|$[begin e_1 ; ... ; e_k end]

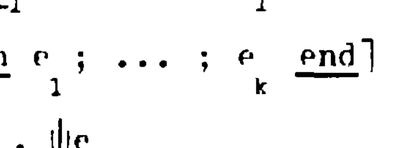
$= \|e_k . \ldots . \|e_1$ ||

lemma 2 If the expressions e_1, e_2, and e_3 possess property A, then so does the expression

if e_1 then e_2 else e_3

proof First we consider the derivation connected with the following diagram, for some e which has property A.

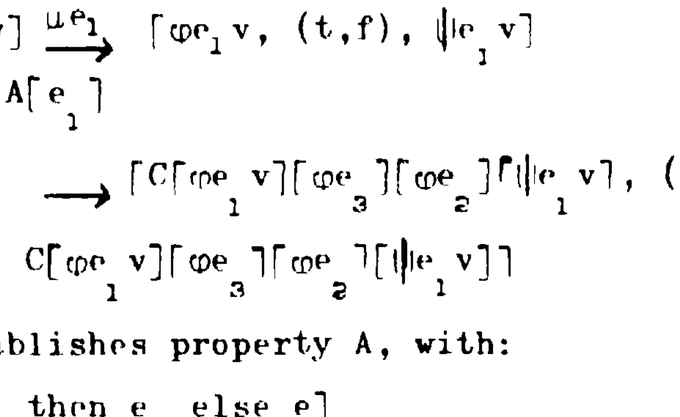
The diagram corresponds to the form of each arm of the conditional expression. The initial operation, TIF is used to remove the redundant value of the tested expression. This context allows us to assume that F is not empty.



$\lceil t,f,v\rceil \xrightarrow{T\downarrow F} \lceil f^+, f^-, v\rceil$ assume $f \neq \Lambda$

$\xrightarrow{\mu e} \lceil \varphi ev, f, \|ev\rceil$ since A[e]

Now in the program



the condition selects between two functions of the same form, thus

$\lceil t,f,v\rceil \longrightarrow \lceil Ct[\varphi e_3]\lceil \varphi e_2\rceil v,\ f,\ Ct[\|e_3]\lceil \|e_2\rceil v]$

for the value of the accumulator since this suggests that both e_2 and e_3 are evaluated.

Finally, for the whole program as shown in section 3 we have

$[t,f,v] \xrightarrow{\mu e_1} \lceil \varphi e_1 v,\ (t,f),\ \|e_1 v]$

since $A\lceil e_1\rceil$

$\longrightarrow \lceil C\lceil \varphi e_1 v\rceil[\varphi e_3][\varphi e_2]\lceil \|e_1 v\rceil,\ (t,f),$

$C[\varphi e_1 v][\varphi e_3]\lceil \varphi e_2\rceil[\|e_1 v]]$

which establishes property A, with:

φ[if e_1 then e_2 else e_3]

$= \lambda v C\lceil \varphi e_1 v\rceil[\varphi e_3][\varphi e_2]\lceil \|e_1 v\rceil$

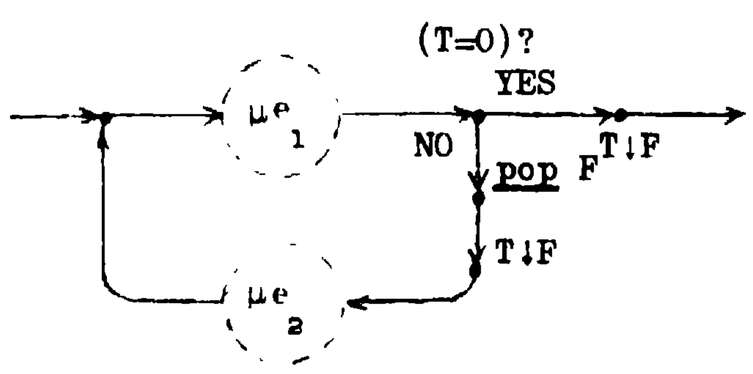
$\|$[if e_1 then e_2 else e_3]

$= \lambda v C\lceil \varphi e_1 v\rceil[\psi e_3]\lceil \|e_2\rceil[\|e_1 v]$ ||

lemma 3 If the expressions e_1 and e_2 possess property A, then so does the expression

while e_1 do e_2

proof Consider the program:

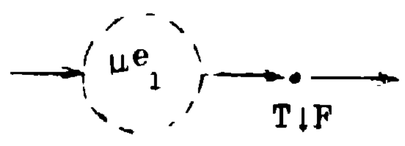


we postulate that there exist functions α, β with the property that the object-environment mapping of this program (if it terminates) is

$\lceil t,f,v\rceil \longrightarrow [\alpha tv,\ f, \beta v]$

We postulate that there exist functions α and p constructively by induction on k, the number of times the condition in the loop proves false.

Suppose k = 0, then the above program corresponds to



which has

$$\lceil t,f,v\rceil \xrightarrow{\mu e_1} \lceil \varphi e_1 v,\ (t,f),\psi e_1 v\rceil \xrightarrow{T\downarrow F} \lceil t,f,\|e_1 v\rceil$$

Thus the hypothesis is true for k = 0, wit

$$\alpha tv = t$$

$$\beta v = \psi e_1 v$$

Suppose now the hypothesis is true in case we should go round the loop k - 1 times, and that in this case we have $\alpha = \alpha'$ and $\beta = \beta'$, then if the condition should prove false k times, we have the derivation.

$$\lceil t,f,v\rceil \xrightarrow{\mu e_1} \lceil \varphi e_1 v,(t,f),\|e_1 v\rceil \quad \text{since } A\lceil e_1\rceil$$

$$\xrightarrow{\underline{pop}\ F} \lceil \varphi e_1 v,f,\psi e_1 v\rceil$$

$$\xrightarrow{T\downarrow F} \lceil f^+,f^-,\psi e_1 v\rceil$$

$$\xrightarrow{\mu e_2} \lceil \varphi e_2\lceil\|e_1 v\rceil,\ f,\ \psi e_2\lceil\|e_1 v\rceil\rceil$$

$$\longrightarrow \lceil \alpha'\lceil \varphi e_2\lceil\|e_1 v\rceil\rceil\lceil\|e_2\lceil\|e_1 v\rceil\rceil,$$

$$f,\ \beta'\lceil\|e_2\lceil\|e_1 v\rceil\rceil\rceil$$

by the inductive hypo thesis, since condition proves false after k - 1 mae loops. This establishes the hypothesis and rv and p have the form

$$\alpha tv = \alpha'\lceil \varphi e_2\lceil\|e_1 v\rceil\rceil\lceil\|e_2\lceil\|e_1 v\rceil\rceil$$

$$\beta v = \beta'\lceil\|e_2\lceil\psi e_1 v\rceil\rceil$$

However, in order to get a definitional form for α and p we must consider the partial derivations chosen by the condition in the loop.

i) $\lceil t,f,v\rceil \xrightarrow{T\downarrow F} \lceil f^+,f^-,v\rceil$ for the YES branch

ii) $\lceil t,f,v\rceil \xrightarrow{\underline{pop}\ F} \lceil t,f^-,v\rceil$

$$\xrightarrow{T\downarrow F} \lceil f^{-+},f^{--},v\rceil$$

$$\xrightarrow{\mu e_2} \lceil \varphi e_2 v,f,\|e_2 v\rceil \quad \text{since } A\lceil e_2\rceil$$

$$\longrightarrow \lceil \alpha\lceil \varphi e_2 v\rceil\lceil\psi e_2 v\rceil,\ f,\ \beta\lceil\psi e_2 v\rceil\rceil$$

for the NO branch

Now for the whole diagram, we have

$$\lceil t,f,v\rceil \xrightarrow{\mu e_1} \lceil \varphi e_1 v,\ (t,f),\|e_1 v\rceil,\ \text{since } A\lceil e_1\rceil$$

$$\longrightarrow \lceil C\lceil \varphi e_1 v\rceil\lceil\alpha t v\rceil\lceil\alpha v\lceil \varphi e_2 v\rceil\|e_2 v\rceil\rceil\|e_1 v\rceil\rceil$$

$$f,C\lceil \varphi e_1 v\rceil\lceil\beta v\rceil\beta v\lceil\|e_1 v\rceil\|e_2 v\rceil\rceil$$

and hence

$$\alpha tv = C\lceil \varphi e_1 v\rceil\lceil\alpha t v\rceil\lceil\alpha v\lceil \varphi e_2 v\rceil\rceil\lceil\|e_2 v\rceil\rceil\|e_1 v\rceil$$

$$\beta v = C\lceil \varphi e_1 v\rceil\lceil\beta v\rceil\lceil\beta v\lceil\|e_2 v\rceil\rceil\lceil\psi e_1 v\rceil$$

Thus

$$\alpha = Y\lambda\alpha\lambda t\lambda v C\lceil \varphi e_1 v\rceil\lceil\lambda v t\rceil\lceil\lambda v\alpha\lceil \varphi e_2 v\rceil\rceil\lceil\psi e_2 v\rceil\rceil\lceil\psi e_1 v\rceil$$

$$\beta = Y\lambda\beta\lambda v C\lceil \varphi e_1 v\rceil\lceil\lambda v v\rceil\lceil\beta.\|e_2\rceil\lceil\|e_1 v\rceil$$

Now to establish property A, we have simply to construct a derivation for the whole program given in section 3.

$$\lceil t,f,v\rceil \xrightarrow{T\uparrow F} \lceil t,(t,f),v\rceil$$

$$\xrightarrow{T\leftarrow \underline{0}} \lceil \underline{0},(t,f),v\rceil$$

$$\longrightarrow \lceil \alpha\underline{0}v,(t,f),\beta v\rceil$$

which establishes property A. We have

$$\varphi\lceil \underline{while}\ e_1\ \underline{do}\ e_2\rceil =$$

$$\lceil Y\lambda\alpha\lambda t\lambda v C\lceil \varphi e_1 v\rceil\lceil\lambda v t\rceil\lceil\lambda v\alpha\lceil \varphi e_2 v\rceil\rceil\lceil\psi e_2 v\rceil\rceil\lceil\psi e_1 v\rceil\rceil\underline{0}$$

$$\|\lceil \underline{while}\ e_1\ \underline{do}\ e_2\rceil =$$

$$Y\lambda\beta\lambda v C\lceil \varphi e_1 v\rceil\lceil\lambda v v\rceil\lceil\beta.\|e_2\rceil\lceil\|e_1 v\rceil$$

5. Conclusions

In each of the above lemmas we have derived a value f unc tion and n side-effect function , for the expression concerned, in terms of the value and side-effect functions of the component expressions. These functions are in a form which this author has found convenient for manipulative purposes. The formulation would be called 'applicative' by Curry and Feys (2) (in a lose sense). They are not, however, in a digestible form for the defmition of langunge in an educational sense. In Henderson (3), the result on the while expression is analysed and alternative forms given. In particular the side-effect function is shown to be equivalent to that defined by Hoare (4).

The requirement for formal semantic description, in the author's view, is most essential in the area of program correctness. Depending upon whether the correctness proof is the duty of the programmer or the machine, the semantic metalanguage must be so oriented. The meta language used here is only nominally oriented towards mechanical handling but this is an area in which the author is currently experimenting.

To summarise, ve have defined a language informally and then an implementation for it. The semantics of the object language and the definition of a translator into the object language specify the semantics of the source language in a procedural way. By analysing the translation of the source phrase it has been possible to derive a functional description of the source language semantics. Correctness of the implementation involves acceptance that the informal interpretation of the derived semantic functions corresponds to the meaning informally attributed to the phrases when they were defined.

A final point worth noting is that the results presented here were developed in the context of a particular implementation. Yet they turned out to be remarkably independent of many of the design features of that implementation. This fact has enabled us to investigate just three of the constructs here without concern for the others. It is heartening to note that the demands of the proofs rationalised the language design to a point where it was easy to describe to both man and machine.

Acknowledgements

This research, which was carried out in the Computing Laboratory of the University of Newcastle upon Tyne from 1967 to 1970, WAS supported initially by the Science Research Council and subsequently by the University. The author is extremely grateful to Mr. M.J. Elphick of the Laboratory for the time he devoted to reading and discussing this material

1 Burstall, R.M. (1969) 'Proving properties of programs by structural induction'. Computer Journal. Vol 12, p. 41.

Curry, H.F. and Feys, R. (1958) 'Combinatory Logic'. Vol. 1. North Holland, Amsterdam.

Henderson, P. (1970) 'Design and Semantic-Analysis of a programming language and its compiler'. Ph.D. Thesis, University of Newcastle upon Tyne.

Hoare, C.R. (1969). 'An axiomatic basis for computer programming'. CACM, Vol. 12 Nov. 1969, p. 576.

5 Landin, P.J. (1963) 'The mechanical evaluation of expressions'. Computer Journal, Vol. 6, p. 308. 1963.

6. Landin, P.J. (1965) 'A correspondence between Algol and Church's λ-calcuius'. CACM Vol. 8, 1965, p.89 + p.158.

7. McCarthy, J. 'Towards a mathematical science of computation'. IFIP Proc. (1962) p. 21.

8. McCarthy, J. and Painter, J. (1967) 'Correctness of a Compiler for Arithmetic expressions'. AMS Symp. in Appl. Math 19, 1967.

9. Strachey, C. (1964) 'Towards a formal semantics' in 'Formal language description languages for computer programming'. Ed. T.H. Steel. Publ. North Holland. 1964 IFIP Conf. Proc. Baden.

10. Wirth, N. (1966) 'Euler, a generalisation of Algol and its formal definition' CACM, Vol. 9, p. 13.

Typographical note: In the iterated fuinction "dot" product like $\cdot_{1} \ldots \cdot_{n}$ the "dot" should not be confused with the ellipses. A succession of three dots is used consistently as ellipses, a single dot must therefore be interpreted as "dot" product.