TECHNISCHE HOGESCHOOL TWENTE

MEMORANDUM NR. 281

SOME SELF-REPRODUCING ALGOL-LIKE PROGRAMS AND KLEENE'S RECURSION THEOREM FORMULATED IN CONCRETE PROGRAMMING LANGUAGES

MAARTEN M. FOKKINGA

SEPTEMBER 1979

zie ook J Kraus, 1981
Techn Rep.
Selbstreproduzierende Programme
Techn Rep 110, 1981, Tech Univ Dortmund

Department of Applied Mathematics,
Twente University of Technology,
P.O. Box 217,
7500 AE Enschede, The Netherlands

ONDERAFDELING DER TOEGEPASTE WISKUNDE

Contents

Page

Abstract

We give some self-reproducing Algol 60 and Algol-like programs. We show that one of them can be derived quite formally from Kleene's Recursion Theorem. To this end we reformulate the theorem and its proof in terms of manipulations on program texts, wherever possible, rather than on indices of functions. As further applications the theorem is formulated in terms of the language HET, the pure lambda calculus and LISP. We consider our understanding of the decades old proof as the main achievement of the work done.

1. Introduction

It was just for fun that we have ~~some time ago~~ constructed some elegant self-reproducing Algol 60 programs. The programs print out their own text, layout inclusive, and do not use any representation of characters other than strings. When we set out to record them in a short note and were about to write the introduction, we made severe attempts to find some motivation for the interest in self-reproducing programs, but we could only provide the following weak one.

A self-reproducing program is a program the semantics of which is defined in terms of in its syntactic form. (There is also no better proof of its correctness than a test run!). For form~~ulations~~alisms which allow the expression of all partial recursive functions, Kleene's Recursion Theorem of the Theory of Recursive Functions, asserts that such programs exist, and it gives as well a construction for them. The construction requires a coding of programs into manipulatable objects, i.e. into some data type of the language, and a so-called s-m-n function which, given a code of $\lambda x,y.\ f(x,y)$ and a value v, yields a code of $\lambda y.\ f(v,y)$. For Algol-like languages the canonical coding of programs into integers may lead to the manipulation of unfeasably large values. So it is of some interest whether a direct construction of a small size program is possible or not.

[Meertens 1974] claims to have found his self-reproducing HET program by following Kleene's construction as reported in [Van Emde Boas 1974]. However, we failed several times in our attempts to do it in a fully systematic way. Moreover, we got thereby the feeling that also one of our self-reproducing Algol 60 programs should be derivable from the proof of the Recursion Theorem. So we set out to derive at any cost our and Meertens' self-reproducing programs from Kleene's construction.

In [Kleene 1952] a very specific formal language and a very specific coding have been chosen. In [Rogers 1967] no syntax assumption at all has been made; not even an "abstract syntax" is mentioned. In addition, both theories deal with functions on natural numbers only. We however want to deal with programming languages in general, with several data types, and we also want to abstract from the particular coding. We have succeeded in a very systematic transcription of the classical proofs of [Kleene 1952] and [Rogers 1967] into a form which, whenever possible, manipulates program texts rather than their codes.

Surpri~~n~~singly, the resulting proof is intuitively very clear and understandable, whereas the classical proofs, although equally short and correct, seem hard to understand; see e.g. [Van Emde Boas 1974].

The main achievement of our work is that we now do understand the classical proof. We are able, for instance,

- to identify the entity in the abstract construction which takes care of the concrete constraint that quotes within strings have to be written twice (, or have a special denotation),
- to explain why Meerten's program contains occurrences of the so-called universal function whereas our Algol 60 program doesn't do so,
- to see at the abstract level the consequences of peculiar syntax constraints (like the obligation to declare procedures, in Algol 60; the option to embrace within <u>begin</u> and <u>end</u>; and in general the full syntax of the language which might be very different from a pure applicative language),
- to obtain "semantical fixed points" of any data type, whereas the Recursion Theoretic construction only yields (codes of) functions.

In short, we are now able to apply the Recursion Theorem to any language whatsoever. Anyone who does already understand the proof had better stop reading now and derive self-reproducing programs himself.

The remainder of the paper is organized as follows. In section 2 we give some self-reproducing Algol 60 programs*). In section 3 we transcribe Kleene's Recursion Theorem and its proof into a formulation which makes explicit use of a -- very high level -- programming language.
In section 4 we apply the theorem to obtain a program given in section 2, and we give its formulation in HET, the pure lambda calculus and LISP.

<u>Acknowledgement</u>
We thank Joost Engelfriet for helpful comment and stimulating interest.

*) which we invented in an ad hoc way

2. Some self-reproducing Algol 60 and Algol-like programs

We assume that there is no distinction between a left quote and a right quote; however for readability we do use three representations: ‘, ', ’ .

In the construction of the programs the main problem is caused by the appearance of special symbols which within a string, have a special denotation. We treat the following three conventions.

1: Only quotes have a special denotation: they have to be written twice; line feeds and other layout characters within a string are significant.

2: Characters C1,..,Cm (C1 being the quote) have special denotation DEN1,..,DENm . Example: m=2 , the denotation of quotes and line feeds are 'Q' and 'L' . If the line feed has a special denotation then it has no significance within a string -- just as in the remainder of the program text --.

3: Characters C1,..,Cm (C1 being the quote) have no denotation: they can only be written by standard procedures writec1,..,writecm.

The following program, for convention 1, is a structurally improved version of [Fokkinga 1973]. Strings do occur in write commands only. See figure 1.

At the price of a longer text the layout of the resulting program may be improved as follows. The layout of the first six lines may be changed at will; this affects the texts of the box as well, because that consists of the "same text as above...". Thus we can arrange that all write commands of the for loop body appear just below each other: insert a line feed and five more spaces after the second occurrence of "quote;" in the third line. Similarly, for the tail of the program text. With some constraints, we may also insert more program text in front of the for loop and in the tail.

idem for 2nd line of box!

The generalization to convention 2 is straightforward: eliminate (each denotation of) each special symbol from the iteration. See figure 2.

For convention 3 we simply replace the procedure declarations of the previous program by

```
...
proc ci; if t=1 then writeci else
         begin writec1; write(‘); ci; write(’); writec1 end ,...
proc tail; begin writec1; write(‘) end; tail end’) end .
```

```
begin
integer t;
proc quote; if t=1 then write(‘''’) else write (‘''); quote; write(''’);
proc tail; write(‘'') end; tail end’);
for t:=1 step 1 until 2 do
begin write (‘
        Same text as above but for the replacement of each quote
        by:        ’); quote; write(‘      .
       (Hence there appear another 15 write commands within the
        for loop body, 10 of which will print the empty string!)
                              ’) end; tail end
```

figure 1: Strings only occurring in write commands.

```
begin
integer t;
proc c1; if t=1 then write(‘DEN1’) else write(‘DEN1); c1; write(DEN1’);
  :
proc cm; if t=1 then write(‘DENm’) else write(‘DEN1); cm; write(DEN1’);
proc tail; write(‘DEN1) end; tail end’);
for t:=1 step 1 until 2 do
begin write(‘
          same text as above but for the replacement of each special
          symbol Ci by     ’); ci; write(‘
                         ’) end; tail end
```

figure 2: Generalization of fig. 1.

In the following programs, procedure reproduce (an anagram!) is able to write any text, provided it is given as parameter the denotations of strings from which the text can be built. The following mnemonics are used: q for quote, c for comma, s0 and s1 for strings (s0 is the initial part and s1 the tail of the program).
For the first convention see figure 3 below.

It is this program which reminds us of Kleene's construction; we will indeed derive it formally in section 4. Although the comma is not a special symbol, it seems to play a special role: it is treated differently from s0 and s1 .

The generalization to convention 2 is straightforward. See figure 4. The adaptation to convention 3 is left to the reader.

```
begin proc repro(q, c, s0, s1); string q, c, s0, s1;
        write(s0, q,q,q,q, c, q,c,q, c, q,s0,q, c, q,s1,q, s1);
     repro('''', ', ', '
begin proc repro (q, c, s0, s1); string q, c, s0, s1;
        write (s0, q,q,q,q, c, q,c,q, c, q,s0,q, c, q,s1,q, s1);
     repro( ', ') end') end
```

figure 3: With string parameters.

```
begin proc repro (c1,...,cm, c, s0,...,sn); string c1,...,cm, c, s0,...,sn;
        write( {lines 1..5:}  s0,ci1,s1,ci2, .. , sk-1,cik,
                  {the box:   }  s0, c1,c,c1, .. , c1,c,c1, sn,
                  {last lines:}  c1, sk, cik+1, .. , cin, sn);
     repro('DEN1', .. , 'DENm', ', ', ' Same text as outside this box,
but for the replacement of each special symbol by ', ' (possibly introducing
empty strings). It is assumed that Ci1 , .. , Cin are, in that order, the
symbols replaced. ')
end
```

figure 4: Generalization of fig. 3.

fig 3 speciaal geval van fig 4? Comma mag niet speciaal symbool zijn!!

Basically, in procedure reproduce the naming facility of the language is used, so that in the body the string denotation of the whole program can be computed (and printed) without a need for an explicit occurrence of it (which is manifestly impossible). Instead of exploiting the formal parameter naming facility, we can also exploit other language features to achieve the same result. The programs given in figures 5, 6 and 7 thus use multiple constant definition (or multiple variable assignment), respectively arrays of strings and finally substring accessibility (or: strings considered as character arrays). But none of these programs are legal Algol 60 texts. For correction?

```
begin string q, c, s0, s1; q, c, s0, s1 := ‘''’, ‘, ’, ‘
begin string q, c, s0, s1; q, c, s0, s1 := ’, ‘;
      write(s0, q,q,q,q, c, q,c,q, c, q,s0,q, c, q,s1,q, s1) end’;
      write(s0, q,q,q,q, c, q,c,q, c, q,s0,q, c, q,s1,q, s1) end
```

figure 5: With multiple assignment.

```
begin string array s[0:2]; string q, c; integer i;
     i := -1; q := ‘''’; c := ‘;
     i := i+1; s[i] := ’;
     i := i+1; s[i] := ‘begin string array s[0:2]; string q, c; integer i;
     i := -1; q := ’;
     i := i+1; s[i] := ‘; c := ’;
     i := i+1; s[i] := ‘;
     write(s0, q,q,q,q, s1, q,c,q, c, q,s0,q, c, q,s1,q, c, q,s2,q, s2) end’;
     write(s0, q,q,q,q, s1, q,c,q, c, q,s0,q, c, q,s1,q, c, q,s2,q, s2) end
```

figure 6: With arrays of strings (for brevity, subscripts have been used instead of indices).

position: 32

```
begin char array  s[1:102]; s := ‘''begin char array s[1:102]; s := ;
     write(s[2:32], s[1], s[1], s[1], s[2:102]; s[1], s[33:102]) end’;
     write(s[2:32], s[1], s[1], s[1], s[2:102], s[1], s[33:102]) end
```

figure 7: String considered as char array.

begin is 5 char's

The semicolon in s occurs at position 32.

3. The Recursion Theorem

Recursive Function Theory deals with computable functions on the natural numbers. One assumes some formal system in which only and all computable functions on the natural numbers can be expressed. There is some enumeration of all formal expressions, and the meaning of the i-th expression is denoted by ϕ_i ; i might be called the code of the i-th program. For given f , any i with $\phi_i = f$ is called an index of f . The total function s11 is a so-called s-m-n function, satisfying $\phi_{s11(x,y)}(z) = \phi_x(y,z)$; the function u is a so-called universal function satisfying $u(x,y) = \phi_x(y)$. The formal definitions of s11 and u heavily depend on the specific formal system and the chosen enumeration; cfr. [Kleene 1952] p.342. The recursion theorem now reads as follows.

Theorem 1. For any computable $t : \mathrm{Nat} \to \mathrm{Nat}$ there exists a $p : \mathrm{Nat}$ satisfying $\phi_p = \phi_{t(p)}$, i.e. the p-th program is semantically equivalent with the t(p)-th program.

Proof. (1) cfr. [Rogers 1967] p. 180. Let s be a total function satisfying $\phi_{s(x)} = \phi_{\phi_x(x)}$ and let c be some index of $\lambda x.\ t(s(x))$ and let $p = s(c)$. Indeed $\phi_p = \phi_{s(c)} = \phi_{\phi_c(c)} = \phi_{t(s(c))} = \phi_{t(p)}$. We can be slightly more explicit about s : let i be some index of $\lambda x,y.\ u(u(x,x),y)$, then we may define $s := \lambda x.\ s11(i,x)$. but other definitions might exist as well.

(2) cfr. [Rogers 1967] p. 214 exc. 11-4,5. Let $s = \lambda x.\ s11(x,x)$ and $c =$ some index of $\lambda x,y.\ \phi_{t(s(x))}(y)$ and $p = s(c)$. Indeed $\phi_p = \phi_{s(c)} = \phi_{s11(c,c)} = \lambda y.\ \phi_c(c,y) = \lambda y.\ (\lambda x,y.\ \phi_{t(s(x))}(y))(c,y) = \phi_{t(s(c))} = \phi_{t(p)}$. As in (1), we may define c by means of u : $c =$ some index of $\lambda x,y.\ u(t(s(x)),y)$. (End of proof.)

In the above proofs it is crucial that s is total. Indeed, with $s = \lambda x.\ \phi_x(x)$ and $c =$ some index of $\lambda x.\ t(s(x))$ and $p = s(c)$ we may even prove $p = t(p)$, which seems to contradict with the possibility $t = \lambda x.\ x+1$. The paradox however is resolved by noting that s is not total, and in particular s(c) , hence p , is undefined. In a correct definition of s , $\phi_x(x)$ is actually a subprogram of the program with index s(x) .

Note also that t need not be total; p is defined anyway. We should however interpret $\phi_{t(p)}$ as the totally undefined function, when t(p) is not defined. Indeed, ϕ_p is totally undefined in that case.

We will now restate the theorem in a form which is more explicit, but not too specific, about the formal language. We also abstract from the Natural numbers and arithmetic functions as the basic and only objects. We are forced to make some notational conventions and some weak assumptions, e.g. as follows. parentheses occur in the meta-language.

instead of parentheses

F 1. Program texts are written in uppercase and with square brackets ~~only~~.

2. The language need not be untyped, like LISP; with some adaptations in the sequel, it may be typed, like Algol 68. There is however some set of values, (the expressions for which we consider to be of) data type VAL , and for this data type also the data type of functions VAL → VAL is supposed to exist. We denote by Val the set of denotations of data type VAL , and for technical simplicity we consider Val to be a semantic domain as well. Thus the meaning of an expression of type VAL belongs to Val , and the meaning of an expression of type VAL → VAL belongs to Val → Val .

3. We let Txt be the set of "meaningful program text parts". For simplicity you might consider any string of terminal symbols meaningful, but possibly denoting a semantical error. Meta-variables ranging over Txt , and only these, are written with two lowercase letters, like xx, ap1, ap2 , with a possible exception for those ranging over Val {which is ⊆ Txt}.

4. We write the expressions for abstraction and application like typed λ-notation, using L̲ for uppercase lambda. These notations thus need considerable change in order to meet particular syntax rules of concrete programming languages. ~~Thus, in some cases,~~

~~L̲ Z : VAL. xx and yy[z]~~

~~should be written in Algol 60~~

~~[z]; VALUE Z; VAL Z; xx' and~~

~~BEGIN VAL PROC P yy' ; P[z] END~~

~~where xx' and yy' are already adapted to the Algol 60 syntax, and z : Val .~~ is gen A'60

5. The meta-symbol = , when applied to operands from Txt , means semantical equivalence. Thus VAL expressions are coerced to Val and VAL → VAL expressions to Val → Val . We use ≡ to denote syntactic equality. The meta-symbol := is used for definitions; only the right hand side is coerced, namely to the mode of the left hand side.

How? We are ready for a systematic transcription. The enumeration of the formal expressions now is an injective code : Txt → Val . We let decode : Val → Txt be the partially defined inverse of code . For readability we mostly write code(xx) as $\overline{xx}$. So $\phi_x = \phi_{code(xx)}$ can now be written decode(x) = xx .

The functions and value s, s11 and c need all be expressed formally. We might make explicit the dependency on the coding as follows. Assume that code is syntax directed ("the code of a composition equals a composition of the codes of the constituents"), and moreover that the coding can be simulated within the language. That is, there exist cd , ap , ap' , ld : Txt satisfying

$cd[\overline{xx}] = code(\overline{xx}) \quad \{= \overline{\overline{xx}}\}$, *cd[x] = code(x) for all x ∈ Val*

$ap[\overline{xx},y] = \overline{xx[y]}$,

$ap'[\overline{xx},y,z] = \overline{xx[y,z]}$,

$ld[\overline{xx},\overline{yy}] = \overline{\underline{L}\ xx: VAL.\ yy}$.

Indeed, s11 may now be expressed by

$ss11 := \underline{L}\ X, Y: VAL.\ ld[\bar{Z}, ap'[X,cd[Y],\bar{Z}]]$,

because $decode(ss11[\overline{xx},y]) = \underline{L}\ Z: VAL.\ xx[y,Z]$. The appearance of the function cd:Txt was for us rather surprising; it takes however care of doubling the quotes in the self-reproducing Algol 60 program. For an explicit definition of s as suggested in proof (1) of the theorem, we need to assume the existence of some uu:Txt satisfying

$uu[\overline{xx},y] = xx[y]$, or equivalently

$\underline{L}\ Y : VAL.\ uu[x,Y] = decode(x)$

for now we may express s by

$ss := \underline{L}\ X : VAL.\ ss11[\overline{\underline{L}\ X,Y : VAL.\ uu[uu[X,X],Y]},\ X]$.

This completes a straightforward transcription. We find

<u>Theorem 2.</u> For any tt : Txt of type VAL → VAL there exists a p : VAL satisfying decode(p) = decode(tt[p]) .

<u>Proof.</u> We follow proof (1) of theorem 1. Let ss be defined as above and let $c := \overline{\underline{L}\ X : VAL.\ tt[ss[X]]}$ and p := ss[c] . (End of proof.)

why?

However, we are interested in a construction manipulating program texts rather than their codes. Reading $\overline{Txt}$ for Nat , and shifting whereever possible from $\overline{Txt}$ to Txt , we make the following transcription of proof (1) of Theorem 1,

t : Nat → Nat	⟹ tt of type VAL → any type tp
$\phi_p = \phi_{t(p)}$ and type of p is Nat	⟹ $fp = tt[\overline{fp}]$ and type *of* fp is tp,
$\phi_{s(x)} = \phi_{\phi_x(x)}$	⟹ $ss[\overline{xx}] = \overline{xx[\overline{xx}]}$,
$\phi_c(x) = t(s(x))$	⟹ $cc[\overline{xx}] = tt[ss[\overline{xx}]]$,
p := s(c)	⟹ $fp :=\ decode(ss[\overline{cc}])$.

Rather surprisingly, this gives a proof which seems far more understandable than the original proof, and, needless to say, a simpler construction than in the proof of theorem 2.

<u>Theorem 3.</u> For any tt : Txt of type VAL $\rightarrow$ any type tp , there exists a program text fp : Txt of type tp which is semantically equivalent to $\mathrm{tt}[\overline{\mathrm{fp}}]$, i.e. $\mathrm{fp} = \mathrm{tt}[\overline{\mathrm{fp}}]$.

<u>Proof.</u> Let the text fp consist of an application, where the operand is the code of the operator, and the operator first reproduces the code of the whole (by means of ss) and then subjects it to tt , yielding $\mathrm{tt}[\overline{\mathrm{fp}}]$. So let

$\mathrm{ss}[\overline{\mathrm{xx}}] = \overline{\mathrm{xx}[\overline{\mathrm{xx}}]}$, e.g. ss := $\underline{L}$ X : VAL. ap[X,cd[X]]

$\mathrm{cc}[\overline{\mathrm{xx}}] = \mathrm{tt}[\mathrm{ss}[\overline{\mathrm{xx}}]]$, e.g. cc := $\underline{L}$ X : VAL. tt[ss[X]] .

$\mathrm{fp} := \mathrm{decode}\ (\mathrm{ss}[\overline{\mathrm{cc}}]) \equiv \mathrm{cc}[\overline{\mathrm{cc}}]$.

Indeed, $\mathrm{fp} \equiv \mathrm{cc}[\overline{\mathrm{cc}}] = \mathrm{tt}[\mathrm{ss}[\overline{\mathrm{cc}}]] = \mathrm{tt}[\overline{\mathrm{cc}[\overline{\mathrm{cc}}]}] = \mathrm{tt}[\overline{\mathrm{fp}}]$. (End of proof.)

Remark. Suppose we adapt theorem 1 and its proof (2) as follows. Replace t by t' satisfying $t'(x) = \phi_{t(x)}$, and consequently s11 by s10 satisfying $\phi_{s10(x,y)} = \phi_x(y)$. Note that these equalities are "type incorrect" in classical Recursive Function Theory, because $\phi_{..}$ always denotes a function and not a functional (t'), nor a plain value $(\phi_{s10(x,y)})$. In our language however, we did not exclude those data types. Performing now the analogous transcription as above yields exactly the same proof!

Indeed, s10 is represented by

ss10 := $\underline{L}$ X,Y : VAL. ap[X,cd[Y]] ,

for $\mathrm{ss10}[\overline{\mathrm{xx}},\mathrm{y}] = \overline{\mathrm{xx}[\mathrm{y}]}$. (End of remark.)

One final and important remark is in order; it might be called the key to the understanding of the construction. The text of cc is completely irrelevant; it is only required that by invocation of cc first the code of $\mathrm{fp} \equiv \mathrm{cc}[\overline{\mathrm{cc}}]$, whatever way this application actually is written, is computed and then the result is subject to tt . Hence, for a translation of the construction into a concrete language like Algol 68, we may represent some brackets by BEGIN, END and others by [,] , and we may declare some functions explicitly and leave the routine texts of the others still at operator positions, and we may write some applications in one way and others in another way. That is, we need <u>not</u> translate the programs from the abstract language in a uniform way into the concrete language. The appearance of ss in the definition of fp guarantees however that the way the application $\mathrm{cc}[\overline{\mathrm{cc}}]$ is written corresponds to the code computed by cc . Also, if an operator * is available for function composition, then cc might read tt * ss instead of $\underline{L}$ X : VAL. tt[ss[X]] . And clearly, having particular definitions of ss and ap and cd available, we may apply the body replacement rule (which affects the syntax but not the semantics) and replace texts by semantically equivalent ones as many times as we like in order to get a simple text for cc . This is particularly useful if some of cd, ss or tt do not easily translate into the concrete programming language, but their body with suitable substitutions do.

4. Applications

4.1. A self-reproducing Algol 60 program

Knowing that by construction we will find some program fp satisfying fp = tt[$\overline{\text{fp}}$] , it seems obvious to choose

Val := the set of string denotations ,

VAL := STRING ,

$\overline{\text{xx}}$:= the string denotation of xx , and

tt := WRITE .

Remembering notational convention 4 of section 3, we find for ap

$$\text{ap}[\overline{\text{xx}},\overline{\text{yy}}] = \overline{\text{BEGIN PROC P}}\,\hat{}\,\overline{\text{xx}}\,\hat{}\,\overline{\text{; P[}}\,\hat{}\,\overline{\text{yy}}\,\hat{}\,\overline{\text{] END}}$$

where it is assumed that ^ is a string concatenation operator. So for ss we find, in pseudo Algol,

$$\text{ss} := \lfloor\ \text{X : STRING.}\ \overline{\text{BEGIN PROC P}}\,\hat{}\,\text{X}\,\hat{}\,\overline{\text{; P[}}\,\hat{}\,\text{cd[X]}\,\hat{}\,\overline{\text{] END}}\ .$$

Applying the body replacement rule for the occurrence of ss[X] in cc , we find the following theorem. Let

$$\text{cc} := \text{[X]; STRING X; WRITE[}\ \overline{\text{BEGIN PROC P}}\,\hat{}\,\text{X}\,\hat{}\,\overline{\text{; P[}}\,\hat{}\,\text{cd[X]}\,\hat{}\,\overline{\text{] END}}\text{]}\ .$$

fp := BEGIN PROC P cc ; P [$\overline{\text{cc}}$] END ,

then fp = WRITE[$\overline{\text{fp}}$] .

It will cause no problem to eliminate the concatenation operator: WRITE[x^y] = WRITE[x,y] = WRITE[x]; WRITE[y] . But how should we refine cd[X] to legal Algol 60? Recall that cd[$\overline{\text{xx}}$] = $\overline{\overline{\text{xx}}}$; so quotes have to be replaced by their denotation (double quotes) -- and in general each special symbol has to be replaced by its denotation, but we will not pursue the general case -- and the whole has to be embraced by another pair of quotes. Well then, replace in the definition of fp the occurrence of $\overline{\text{cc}}$ by ($\overline{\text{cc1}}$,..,$\overline{\text{ccn}}$) : Txt , where cc1,..,ccn is the sequence of successive maximal quote free parts of cc . It is now easy to express $\overline{\overline{\text{cc1}},..,\overline{\text{ccn}}}$, for this equals $\overline{\text{'cc1',..,'ccn'}}$ = q^$\overline{\text{cc1}}$^qcq^..^qcq^$\overline{\text{ccn}}$^q , provided q = $\overline{\text{‘}}$ = $\overline{\text{’}}$ and qcq = $\overline{\text{’, ‘}}$ (recall that left and right quote are supposed to be equal; if not, the construction becomes slightly more complicated). Thus splitting $\overline{\text{cc}}$, and simultaneously also STRING variable C , yields

C of X

cc := [C1,..,Cn]; STRING C1,..,Cn;

$$\text{WRITE[}\ \overline{\text{BEGIN PROC P}}\text{, C1, q,..,q, Cn, }\overline{\text{; P[}}\ ,$$
$$\text{q, C1, qcq,..,qcq, Cn, q,}\overline{\text{] END}}\ \text{]}$$

fp := BEGIN PROC P cc ; P[$\overline{\text{cc1}}$,..,$\overline{\text{ccn}}$] END .

In order that n , the number of quote free parts of cc , is well defined (and less than infinity), it is required that both q and qcq are quote free! Two solutions suggest themselves: either

1. q and qcq are formal parameter identifiers and $\overline{'}$ and $\overline{',\ '}$ are passed as actual parameter (this yields almost literally the program of section 2), or
2. within the body of cc the procedure declarations
   PROC Q; WRITE[$\overline{'}$]; PROC QCQ; WRITE[$\overline{',\ '}$];
   are inserted, and each occurrence of q is replaced by]; Q; WRITE[and each occurrence of qcq by]; QCQ; WRITE[.

Remark. The above trick of splitting $\overline{cc}$ can be described in abstract terms as well. Indeed, let the comma be the separator of sequences and take

$\overline{xx}$:= the sequence of string denotations of the successive maximal quote free parts of xx ,

Val := sequences of quote free string denotations,

VAL := sequences of STRING expressions.

So, VAL variable X translates to STRING variable sequence X1,..,Xn . The functions cd, ap, ss and tt can not yet be formulated in legal Algol 60, but their result for the particular applications in which they occur in cc can. So we apply the technique described in the last lines of section 3.

"cd[X1,..,Xn]" := (ee, X1, cm,..,xm, Xn, ee)

assuming that both ee and cm are quote free, and ee = empty string denotation , and cm = $\overline{,}$. So indeed

("cd[X1,..,Xn]" where X = $\overline{xx}$) = $\overline{\overline{xx}}$.

Further

"ss[X1,..,Xn]" := $\overline{\text{BEGIN PROC P}}$ + (X1,..,Xn) + $\overline{;\ \text{P[}}$ + "cd[X1,..,Xn]" + $\overline{\text{] END}}$

assuming + : (Txt of type VAL)→(Txt of type VAL), defined by

(xx1,..,xxn) + (yy1,..,yym) = (xx1,..,xxn^yy1,..,yym)

so that $\overline{xx}$ + $\overline{yy}$ = $\overline{xx\ yy}$, hence ("ss[X1,..,Xn]" where X = $\overline{xx}$) = $\overline{\text{BEGIN PROC P xx ; P[}\overline{xx}\text{] END}}$, as required.

Now define

"tt[Y1,..,Yn]" := BEGIN PROC Q; WRITE[$\overline{'}$]; PROC C; WRITE[$\overline{,}$];
WRITE[Y1]; Q; ... ; Q; WRITE[Yn]
<but WRITE[cm] replaced by C>
END

so that ("tt[Y1,..,Yn]" where Y = $\overline{yy}$) = WRITE["string denotation of yy"].

Hence, with

cc := [X0,..,Xn]; STRING X0,..,Xn;
"tt[Y0,..,Yn]" where Y = "ss[X0,..,Xn]" ,

fp := BEGIN PROC P cc ; P[$\overline{cc}$] END .

we only need to eliminate four occurrences of ^ from the WRITE commands in order to obtain a legal Algol 60 self-reproducing program. (End of remark.)

4.2. A formulation of theorem 3 in HET

The Heel Eenvoudige Taal (Very Simple Language (but Highly Encouraging Tricks)) HET might be called an "imperative version of the pure applicative lambda calculus". In order to be self-contained we provide the following brief description; more information can be found in [Meertens 1974].

The evaluation of a program text is from left to right, and each elementary expression, henceforth called "object" (viz. a "word" of letters, a "special symbol", or a possibly empty "list" of objects)is interpreted as a function. There is an initially empty anonymous, stack, on (the uppermost part of) which each function finds its arguments and leaves its results. Besides (there is a memory in which) any value can be (re)associated as "the value of" any other value; it will appear that only and all objects can come forward as value.

Each word and list is a 0-ary function, yielding itself as result {on the top of the stack}.

Here follow the special symbols and their meaning:

- ; is the 1-ary function with no result {it throws away the top of the stack and has no effect on the memory},
- ↓ is a 2-ary function; it reassociates its 2nd argument as the value of its 1st (= top most) argument, yielding the value assigned,
- ↑ is a 1-ary function, it yields the value of its argument,
- \+ is a 2-ary function; it yields the list obtained from its 2nd argument, which must be a list, by inserting its 1st (= top most) argument as a new, first list element;
- / is a 1-ary function, such that + / is semantically equivalent to (the identical stack transformation).
- ! is a (1+n)-ary function, for varying n. It has the effect of the evaluation, as a subprogram,of its 1st argument, stripped of its outermost parentheses if it is a list; the subprogram evaluation may take some n more arguments and leave some or none results.

Our abstract programming language translates easily to HET. Let zz be an identifier, xx and yy texts, and let the prime denote the translation into HET. We find

$(xx[yy])' := yy'\ xx'$,

$(\underline{L}\ zz{:}VAL.\ xx)' := zz \downarrow ;\ xx'$, and

$zz' := zz \uparrow$ for any applied occurrence of zz .

From now onwards we write only HET texts.

For a simple formulation of Theorem 3 we obviously try to code program texts by enclosing them within brackets:

Val := the lists ,

$\overline{xx} := [xx]$.

With this convention cd and ap should satisfy

$\overline{xx}\ cd = \overline{\overline{xx}} \equiv [\overline{xx}]$, and

$\overline{yy}\ \overline{xx}\ ap = \overline{yy\ xx} \equiv [yy\ xx]$.

So we may define

$cd := X \downarrow ;\ [\ ]\ X \uparrow +$,

but a simple definition if ap is problematic; for instance

$ap := X \downarrow ;\ Y \downarrow ;\ X \uparrow Y \uparrow$ "strip of its brackets and + it" .

However we can easily express the required result of the particular application in which ap appears in ss, for in general

$\overline{yy}\ cd\ \overline{xx}\ ap = \overline{\overline{yy}}\ \overline{xx}\ ap = \overline{\overline{yy}\ xx} \equiv [\overline{yy}\ xx] = \overline{xx}\ \overline{yy}\ +$.

So, in the definition

$ss := (\underline{L}\ X{:}VAL.\ ap[X,cd[X]])' \equiv X \downarrow ;\ X \uparrow cd\ X \uparrow ap$

we replace $X \uparrow cd\ X \uparrow ap$ by $X \uparrow X \uparrow +$.

This definition is also suggested by the original requirement

$\overline{xx}\ ss = \overline{\overline{xx}\ xx} \equiv [\overline{xx}\ xx] = [xx]\ \overline{xx}\ + = \overline{xx}\ \overline{xx}\ +$.

We thus find the following theorem. Let

$ss := X \downarrow ;\ X \uparrow X \uparrow +$ or shorter $X \downarrow X \uparrow +$,

$cc := X \downarrow ;\ X \uparrow ss\ tt$ or shorter $ss\ tt$,

$fp :=$ decode $(cc\ ss) \equiv \overline{cc}\ cc$,

then $fp = \overline{fp}\ tt$. Fully written out $fp \equiv [X \downarrow X \uparrow + tt\ ]\ X \downarrow X \uparrow tt$.

Taking tt empty we find the self-reproducing HET program $fp = \overline{fp}$.

[Meertens 1974] sets out to find some list p satisfying $p! = p$. Of course, we find the solution $p := \overline{fp}$ as $\overline{fp}\ ! = fp = \overline{fp}$. Our list $\overline{fp}$ is shorter than the list found by Meertens, but more remarkably, in our program fp there is no occurrence of the symbol ! . We have been seriously mislead by the occurrences of ! in Meertens solution. For it happens that ! equals the universal function uu , because $y\ \overline{xx}\ ! = y\ xx$

and even $\overline{xx}$! = xx , and we thought that

either $\phi_{\overline{xx}}$ had been transcripted systematically in $\overline{xx}$ uu ≡ $\overline{xx}$!

or he had chosen a different coding, viz. something satisfying

$\overline{\overline{yy}\ xx} \equiv [\overline{yy}\ \overline{xx}\ !]$.

Neither of these assumptions enabled us to reinvent his solution. In retrospect the occurrences of ! may be blamed on a less simple way to write the main application of fp , viz. $\overline{cc}$ $\overline{cc}$! instead of $\overline{cc}$ cc . Indeed

ss := X ↓ ; [!] X ↑ + X ↑ +

satisfies $\overline{xx}$ ss = $\overline{\overline{xx}\ \overline{xx}!}$, and with

cc := ss tt ,

fp := decode ($\overline{cc}$ ss) ≡ $\overline{cc}$ $\overline{cc}$! ,

we find

fp ≡ [X ↓ ; [!] X ↑ + X ↑ + tt][X ↓ ; [!] X ↑ + X ↑ + tt] ! = $\overline{fp}$ tt .

4.3. A formulation of theorem 3 in the lambda calculus

Let us take Txt as the usual lambda calculus expressions. We consider each expression fully bracketed, but will omit some brackets for readability. The meaning, semantics, of an expression xx is defined to be the set of all expressions being equivalent to xx on account of the following equivalence relation. The relation is written as = and is the reflexive and transitive closure of the well-known conversion rules, viz.

α: $\underline{L}$ xx. ee = $\underline{L}$ yy. replace(xx,ee,yy)

β: [$\underline{L}$ xx. ee] aa = replace(xx,ee,aa)

η: $\underline{L}$ xx. ee[xx] = ee

for any identifiers xx and yy and expressions ee and aa . Replace(xx,ee,aa) denotes the expression obtained from ee by replacing each free occurrence of identifier xx by the expression, possibly identifier, aa ; it is required that for any free identifier yy of aa , the free occurrences of xx do not occur in subexpressions of ee which are abstractions with respect to yy (i.e. fall in the scope of some $\underline{L}$ yy).

Naturally we take as code of xx some representative of its equivalence class; the simplest choice is xx itself. So

Val := Txt ,

$\overline{xx}$:= xx .

We thus find the following theorem. Let

ss := $\underline{L}$ X. X[X]

so that indeed $ss[\overline{xx}] = \overline{xx[\overline{xx}]}$,

$cc := \underline{L}\ X.\ tt[ss[X]]$ or shorter $\underline{L}\ X.\ tt[X[X]]$,

$fp := decode\ (ss[\overline{cc}])$

$\equiv [\underline{L}\ X.\ tt[X[X]]]\ [\ \underline{L}\ X.\ tt[X[X]]]$

then $fp = tt[fp]$.

Surprisingly, or not?, we find that the above construction of fp coincides with the construction via the paradoxical combinator

$yy := \underline{L}\ T.\ [\underline{L}\ X.\ T[X[X]]]\ [\underline{L}\ X.\ T[X[X]]]$.

Indeed, $fp \equiv$ "β-rule applied to yy[tt]" .

Warning. One should not try to eliminate (by means of rule β) all abstractions at operator positions in yy[tt] . (End of warning.)

Remark. The above result does not prove that in any model of the lambda calculus yy yieldsthe minimal fixed point. And assuming that yy does yield the minimal fixed point -- which is true, see [Scott 1976] --, we may not conclude that the construction given in the proof of theorem 3, gives the minimal fixed point as well -- which indeed is not true, see [Rogers 1967] p. 196 --.

4.4. A formulation of theorem 3 in LISP

Presumably each LISP programmer has taken the trouble once in his life to construct a self-reproducing program. We consider it worthwhile to give the formulation of theorem 3 in LISP, thus obtaining self-reproducing programs and the like without any trouble.

It will appear that we only need a few atomic symbols with a fixed meaning, viz. CONS, LAMBDA, NIL and QUOTE, abbreviated to C , L, N and Q respectively. Thus the construction holds even for pure LISP, as defined in section 1 of [McCarthy 1962].

We define a LISP program to be an S-expression xx , the meaning of which is given by eval(xx, N) . (Recall that the formal programming language is written with capitals and square brackets only; parentheses occur in the meta-language.) Clearly, the sole objects manipulatable by LISP functions are S-expressions and expressions for S-expressions are S-expressions as well.

Thus we find

VAL := the set of S-expressions ,

Val := the set of denotations of VAL values = quoted S-expressions.

It seems obvious to try the simplest possible coding (note that by definition $\overline{xx}$ should belong to Val):

$\overline{xx}$:= the denotation of the S-expression xx ≡ [Q xx].

Now cd and ap should satisfy

[cd $\overline{xx}$] = $\overline{\overline{xx}}$ ≡ $\overline{[Q\ xx]}$,

[ap $\overline{xx}$ $\overline{yy}$]= $\overline{[xx\ yy]}$,

so we may define

cd := [L [X] [C $\bar{Q}$ [C X $\bar{N}$]]] ,

ap := [L [X Y] [C X [C Y $\bar{N}$]]] .

We thus have found a definition for ss :

ss := [L [X] [ap X [cd X]]] .

By the body replacement rule this may be simplified to

ss := [L [X] [C X [C [C $\bar{Q}$ [C X $\bar{N}$]] $\bar{N}$]]]

which we could have found more directly from the original requirement

[ss $\overline{xx}$] = $\overline{[xx\ \overline{xx}]}$ ≡ $\overline{[xx\ [Q\ xx]]}$.

We thus find the following theorem. Let

cc := [L [X][tt [ss X]]] , or simplified

cc := [L [X][tt [C X [C [C $\bar{Q}$ [C X $\bar{N}$]] $\bar{N}$]]]] , and

fp := decode ([ss $\overline{cc}$]) ≡ [cc [Q cc]] ,

Then fp = [tt [Q fp]] .

Taking tt := [L[X] X] (and simplifying cc) we find a self-reproducing program fp = [Q fp] .

We might adopt another notion of LISP program and require a program to be a pair fn aa where fn is an S-expression and aa is a list of S-expressions, and the meaning is given by evalquote(fn, aa) . Actually this is the form in which programs have to be supplied to the[McCarthy 1962] LISP interpreter. Now the denotation of an S-expression xx as an argument in the list aa equals xx and not [Q xx] . Thus

Val := the set of S-expressions .

The theorem will read fp = tt $[\overline{\text{fp}}]$ where fp is a pair. We choose to supply tt with a single argument, and thus define

$$\overline{\text{xx}} := \begin{cases} [\text{xx}] & \text{for pairs xx} \\ \text{xx} & \text{for S-expressions xx .} \end{cases}$$

The requirement for ss now reads

$$\text{ss } [\overline{\text{xx}}] = \overline{\text{xx } [\overline{\text{xx}}]} \equiv [\text{xx } [\text{xx}]] \quad \text{for S-expressions xx .}$$

So we may define

$$\text{ss} := [L[X]\ [C\ X\ [C\ [C\ X\ \bar{N}]\ \bar{N}]]] \quad ,$$

and find the following theorem. Let

$$\text{cc} := [L[X]\ [\text{tt}\ [C\ X\ [C\ [C\ X\ \bar{N}]\ \bar{N}]]]] \ ,$$

$$\text{fp} := \text{decode}(\text{ss}[\overline{\text{cc}}]) \equiv \text{cc}\ [\text{cc}]$$

then fp = tt [[fp]] .

Quite remarkably, QUOTE occurs only in $\bar{N} \equiv [Q\ N]$; in full LISP we may even take $\bar{N} \equiv N$, so that QUOTE does not occur at all in the self-reproducing program fp .

References

Van Emde Boas, P.: Abstract resource bounded classes. Thesis, Mathematical Centre, Amsterdam (1974).

Fokkinga, M.M.: A Self-Reproducing Algol 60 Program. Algol Bulletin 35 (1973) 24-62.

Kleene, S.C.: Introduction to Meta-Mathematics. North-Holland, Amsterdam, 1952.

McCarthy, J. et al: LISP 1.5. Programmers Manual. The M.I.T. Press, Cambridge, Massachusetts (1962).

Meertens, L.: De heel eenvoudige taal HET. Mathematical Centre Syllabus 25 (1974), Mathematical Centre, Amsterdam.

Rogers, H.R.: Theory of recursive functions and effective computability, McGraw-Hill (1967).

Scott, D.: Data types as lattices. SIAM J. Comp. 5 (1976), 3, 522-587.