MEMORANDUM NR. 201

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

MAARTEN M. FOKKINGA

SEPTEMBER 1979

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

ONDERAFDELING DER TOEGEPASTE WISKUNDE
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Some self-reproducing Algol 60 and Algol-like programs</td>
<td>3</td>
</tr>
<tr>
<td>3. The Recursion Theorem</td>
<td>7</td>
</tr>
<tr>
<td>4. Applications</td>
<td></td>
</tr>
<tr>
<td>4.1. A self-reproducing Algol 60 program</td>
<td>12</td>
</tr>
<tr>
<td>4.2. A formulation of theorem 3 in HET</td>
<td>14</td>
</tr>
<tr>
<td>4.3. A formulation of theorem 3 in the lambda calculus</td>
<td>16</td>
</tr>
<tr>
<td>4.4. A formulation of theorem 3 in LISP</td>
<td>18</td>
</tr>
<tr>
<td>References</td>
<td>20</td>
</tr>
</tbody>
</table>

## 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 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 formulations 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 MET 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.
Surprisingly, 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 \texttt{begin} and \texttt{end}; 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.

Acknowledgement

We thank Joost Engelfriet for helpful comment and stimulating interest.
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 program, 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 Cl,...,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 Cl,...,Cm (C1 being the quote) have no denotation: they can only be written by standard procedures write1,...,writecm.

The following program, for convention 1, is a structurally improved version of [Pokkinga 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.

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 writeci; write('); ci; write('); writeci end ..., proc tail; begin writeci; 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('
  
  (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 cl: if t=1 then write('DEN1') else write('DEN1'); cl; 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,q1,q, s1);
    repro("", "", "", "");
end
```

```
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,q1,q, s1);
    repro("", "") end; end
```

figure 3: With string parameters.

```
begin proc repro (c_1,...,c_m, c, s_0,...,s_n); string c, c_1,...,c_m, c, s_0,...,s_n;
    write( {lines 1..5:} s_0, c_1,s_1, c_2,s_2,..., s_k-1,c_k,
            {the box: } s_0, c_1,c_1,..., c_1,c_1,..., s_n,
            {last lines:} c_1,..., c_i,..., c_i,s_i,k+1,..., s_n);
    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 C_1,...,C_i are, in that order, the symbols replaced. ")
end
```

figure 4: Generalization of fig. 3.

```
fig 3 special case for fig 4. Commas may not special symbols.
```

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.

\begin{verbatim}
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] := ' 
    i := i+1; s[i] := ' 
    write(s_0, q, q, q, q, s_1, q, c, q, c, q, s_0, q, c, q, s_1, q, c, q, s_2, q, s_2) end#
    write(s_0, q, q, q, q, s_1, q, c, q, c, q, s_0, q, c, q, s_1, q, c, q, s_2, q, s_2) end

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

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.
\end{verbatim}

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 $\phi_i$; 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 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 $t: \text{Nat} \to \text{Nat}$ there exists a $p: \text{Nat}$ 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 $c$ be 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 $u: c$ is 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 $c$ as 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, $\phi_p$ 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,
_\text{e.g.} as follows._

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
sequent, it may be typed, like Algol 68. There is however some set of values,
(\text{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 Ttxt 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 Ttxt, and only
these, are written with two lowercase letters, like xx, ap1, ap2, with
a possible exception for those ranging over Val (which is \subset Ttxt).
4. We write the expressions for abstraction and application like typed
\text{\lambda}-notation, using \text{\Lambda} for uppercase lambda. These notations thus need
considerable change in order to meet particular syntax rules of concrete
programming languages. Thus, in some cases,
\[ \text{\Lambda} Z : \text{Val. } xx \text{ and } yy[Z] \]
should be written in Algol 60
\[ [Z] ; \text{VALUE } Z ; \text{VAL } Z ; xx' \text{ and } \]
BEGIN VAL PROC P yy' ; P[Z] END
where xx' and yy' are already adapted to the Algol 60 syntax, and
Z : Val.
5. The meta-symbol = , when applied to operands from Ttxt, means
semantical equivalence. Thus Val expressions are coerced to Val and
Val → Val expressions to Val + Val. We use \text{\Xi} 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.

---

\[ \text{How?} \]

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

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 \( \text{code} \) 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 \( \text{cd}, \ \text{ap}, \ \text{ap'} \), \( \text{ld} : \text{Txt} \) satisfying

\[
\begin{align*}
\text{cd}[\overline{\text{x}}] &= \text{code}(\overline{\text{x}}) \quad (= \overline{\text{x}}) , \\
\text{ap}[\overline{\text{x}}, \overline{\text{y}}] &= \overline{\text{xx}[\text{y}]} , \\
\text{ap'}[\overline{\text{x}}, \overline{\text{y}}, z] &= \overline{\text{xx}[\text{y}, z]} , \\
\text{ld}[\overline{\text{x}}, \overline{\text{yy}}] &= \overline{\text{xx} : \text{VAL}. \ \overline{\text{yy}}} .
\end{align*}
\]

Indeed, \( s11 \) may now be expressed by

\[
\text{ss}11 := \overline{\text{x}} \text{, } \text{Y} : \text{VAL}. \ \text{ld}[\overline{\text{z}}, \ \text{ap'}[\overline{\text{x}}, \text{cd}[\overline{\text{y}}], \overline{\text{z}}]] ,
\]

because \( \text{decode}(\text{ss}11[\overline{\text{y}}, \overline{\text{z}}]) = \overline{\text{Z}} : \text{VAL}. \ \overline{\text{xx}[\text{y}, \text{z}]} . \) The appearance of the function \( \text{cd} : \text{Txt} \) 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 \( \text{uu} : \text{Txt} \) satisfying

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

\[
\text{l} \text{Y} : \text{VAL}. \ \text{uu}[\overline{\text{x}}, \overline{\text{y}}] = \text{decode}(\text{x})
\]

for now we may express \( s \) by

\[
\text{ss} := \overline{\text{x}} : \text{VAL}. \ \text{ss}11[\overline{\text{x}}, \overline{\text{Y}} : \text{VAL}. \ \text{uu}[\text{uu}[\overline{\text{x}}, \overline{\text{X}}], \overline{\text{Y}}], \overline{\text{X}}] .
\]

This completes a straightforward transcription. We find

**Theorem 2.** For any \( \text{tt} : \text{Txt} \) of type \( \text{VAL} \rightarrow \text{VAL} \) there exists a \( \text{p} : \text{VAL} \) satisfying \( \text{decode}(\text{p}) = \text{decode}(\text{tt}[\text{p}]) \).

**Proof.** We follow proof (1) of theorem 1. Let \( \text{ss} \) be defined as above and let \( \text{c} := \overline{\text{x}} : \text{VAL}. \ \text{tt}[\text{ss}[\overline{\text{x}}]] \) and \( \text{p} := \text{ss}[\overline{\text{c}}] \). (End of proof.)

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

\[
\begin{align*}
\text{t} : \text{Nat} \rightarrow \text{Nat} \\
\phi_p = \phi_{\text{t}(\text{p})} \text{ and type of } \text{p} \text{ is } \text{Nat} \\
\phi_s(x) = \overline{\text{x}} \text{, } \phi_x(x) \\
\phi_c(x) = \text{t}(\text{s}(\text{x}))
\end{align*}
\]

\[
\text{p} := \text{s}(\text{c}) \rightarrow \text{tt of type } \text{VAL} \rightarrow \text{any type } \text{tp} \\
\text{fp} = \text{tt}[\overline{\text{fp}}] \text{ and type of } \text{fp} \text{ is } \text{tp} \\
\text{ss}[\overline{\text{x}}] = \overline{\text{xx}[\overline{\text{x}}]} \\
\text{cc}[\overline{\text{x}}] = \overline{\text{tt}[\text{ss}[\overline{\text{x}}]]} \\
\overline{\text{fp}} := \text{decode}(\text{ss}[\overline{\text{cc}}]) .
\]
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.

**Theorem 3.** For any \( tt : \text{Txt} \) of type \( \text{VAL} \rightarrow \text{any type tp} \), there exists a program text \( fp : \text{Txt} \) of type \( \text{tp} \) which is semantically equivalent to \( tt[fp] \), i.e. \( fp = tt[fp] \).

**Proof.** 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 \( tt[fp] \). So let 
\[
ss[xx] = xx[xx] \ , \ \text{e.g.} \ \ss := \downarrow x : \text{VAL. ap}[X,cd[X]] \\
cc[xx] = tt(ss[xx]) \ , \ \text{e.g.} \ \cc := \downarrow x : \text{VAL. tt}[ss[x]] \\
fp := \text{decode} \ (ss[cc]) = \cc[cc] 
\]
Indeed, \( fp \equiv \cc[cc] = tt(ss[cc]) = tt[cc[cc]] = tt[fp] \) . (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 \( \phi \ldots \) 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 
\[
ss10 := \downarrow x, y : \text{VAL. ap}[X,cd[y]] \\
\text{for } ss10[xx,y] = xx[y] \ . \ (\text{End of remark.})
\]

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 \( fp \equiv cc[cc] \), 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 not 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 \( cc[cc] \) is written corresponds to the code computed by \( cc \). Also, if an
operator \* is available for function composition, then \texttt{cc} might read \texttt{tt * ss} instead of \( \lambda X : \texttt{VAL. tt[ss[X]]} \). And clearly, having particular definitions of \texttt{ss} and \texttt{ap} and \texttt{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 \texttt{cc}. This is particularly useful if some of \texttt{cd}, \texttt{ss} or \texttt{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 \( fp = tt[fp] \), it seems obvious to choose

\[
\begin{align*}
  \text{Val} & := \text{the set of string denotations} , \\
  \text{VAL} & := \text{STRING} , \\
  \overline{xx} & := \text{the string denotation of} \ xx , \text{ and} \\
  tt & := \text{WRITE} .
\end{align*}
\]

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

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

where it is assumed that \( \overline{\cdot} \) is a string concatenation operator. So for \( ss \) we find, in pseudo Algol,

\[
ss := \overline{\underline{\underline{\text{X}}}} \ := \text{STRING. BEGIN PROC} \ P \overline{X} ; P[\overline{\text{cd}[X]}] \text{END} .
\]

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

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

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

then \( fp = \text{WRITE}[fp] \).

It will cause no problem to eliminate the concatenation operator:

\[
\text{WRITE}[\overline{xy}] = \text{WRITE}[x,y] = \text{WRITE}[x] ; \text{WRITE}[y] . \text{But how should we refine}
\]

\( \text{cd}[X] \) to legal Algol 60? Recall that \( \text{cd}[\overline{xx}] = \overline{xx} \); 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 \( \overline{cc} \) by \( (\overline{ccl}, \ldots, \overline{ccn}) \) :Txt , where \( ccl, \ldots, ccn \) is the sequence of successive maximal quote free parts of \( cc \). It is now easy to express \( \overline{ccl}, \ldots, \overline{ccn} \), for this equals \( 'ccl', \ldots,'ccn' = q\overline{ccl}q\overline{cq}{q}_{q} \ldots q\overline{cq}q \overline{ccn}q \), provided \( q = q = q \) and \( q = q = q \) (recall that left and right quote are supposed to be equal; if not, the construction becomes slightly more complicated). Thus splitting \( cc \), and simultaneously also string variable \( C \), yields

\[
cc := [C1, \ldots, Cn] ; \text{STRING} C1, \ldots, Cn ;
\]

\[
\text{WRITE[ BEGIN PROC} P, C1, q, \ldots, q, Cn, \overline{P} ,
\]

\[
q, C1, q\overline{cq}, \ldots, q\overline{cq}, Cn, q, \overline{P} \text{END} ]
\]

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

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 \( q\overline{cq} \) are
quote free! Two solutions suggest themselves: either
1. \( q \) and \( q\text{cq} \) are formal parameter identifiers and \( \text{?} \) and \( \text{?}\text{, }\text{!} \) are
   passed as actual parameter (this yields almost literally the program of
   section 2), or
2. within the body of \( \text{cc} \) the procedure declarations
   \[
   \text{PROC } q; \text{ WRITE[?]; } \text{PROC } q\text{cq}; \text{ WRITE[?}\text{, }\text{!];}
   \]
   are inserted, and each occurrence of \( q \) is replaced by \( ?; q; \text{ WRITE[} \)
   and each occurrence of \( q\text{cq} \) by \( ?; q\text{cq}; \text{ WRITE[} \).

Remark. The above trick of splitting \( \text{cc} \) can be described in abstract
terms as well. Indeed, let the comma be the separator of sequences and take
\( \text{xx} := \) the sequence of string denotations of the successive maximal
quote free parts of \( \text{xx} \),
\( \text{Val} := \) sequences of quote free string denotations,
\( \text{VAL} := \) sequences of \text{STRING} expressions.
So, \text{VAL} variable \( \mathbf{X} \) translates to \text{STRING} variable sequence \( \mathbf{X}_1, \ldots, \mathbf{X}_n \).
The functions \( \text{cd}, \text{ap}, \text{ss} \) and \( \text{tt} \) can not yet be formulated in legal
\text{Algol} \text{60}, but their result for the particular applications in which they
occur in \( \text{cc} \) can. So we apply the technique described in the last lines of
section 3.

\[
\text{"cd[X1,\ldots,Xn]" := (ee, X1, cm, \ldots, xm, Xn, ee)}
\]
assuming that both \( \text{ee} \) and \( \text{cm} \) are quote free, and \( \text{ee} = \) empty string
denotation, and \( \text{cm} = \) . So indeed
\[
\text{("cd[X1,\ldots,Xn]" where } \mathbf{X} = \text{xx} \text{) = xx .}
\]
Further
\[
\text{"ss[X1,\ldots,Xn]" := BEGIN PROC P + (X1,\ldots,Xn) + P[ + \text{cd[X1,\ldots,Xn]} + ] END}
\]
assuming \( +: (\text{Txt of type VAL}) \ast (\text{Txt of type VAL}) \), defined by
\[
(xx1,\ldots,xxn) + (yy1,\ldots,ym) = (xx1,\ldots,xxn,yy1,\ldots,ym)
\]
so that \( \text{xx + yy = xx yy} \), hence \( \text{("ss[X1,\ldots,Xn]" where } \mathbf{X} = \text{xx} \text{) =}
\)
\[
\text{BEGIN PROC P xx ; P[ xx ] END , as required.}
\]
Now define
\[
\text{"tt[Y1,\ldots,Yn]" := BEGIN PROC Q; WRITE[7]; PROC C; WRITE[,] ;}
\]
\[
\text{WRITE[Y1]; Q; \ldots ; Q; WRITE[Yn]}
\]
\[
\text{<but WRITE[cm] replaced by C>}
\]
\[
\text{END}
\]
so that \( \text{("tt[Y1,\ldots,Yn]" where } \mathbf{Y} = \text{yy} \text{) = WRITE["string denotation of yy"]} \).
Hence, with
\[
\text{cc := [X0,\ldots,Xn]; STRING X0,\ldots,Xn;}
\]
\[
\text{"tt[Y0,\ldots,Yn]" where } \mathbf{Y} = \text{ss[X0,\ldots,Xn]} \text{ ,}
\]
\[
\text{fn := BEGIN PROC P cc ; P[ cc ] END .}
\]
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,

- $\uparrow$ is a 1-ary function, it yields the value of its argument,

- $\downarrow$ 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
\[(xx[yy])' := yy' xx' \ ,\]
\[(\_zz:VAL. \ xx)' := zz \uparrow \ ; \ xx' \ , \text{ and} \]
\[zz' := zz \uparrow \text{ for any applied occurrence of } zz \ .\]

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:

\[Val := \text{the lists} , \]
\[xx := [xx] .\]

With this convention \(cd\) and \(ap\) should satisfy
\[xx \ cd = xx \equiv [xx] , \text{ and} \]
\[yy \ xx \ ap = yy xx \equiv [yy xx] .\]

So we may define
\[cd := X \uparrow ; [ ] X \uparrow + , \]
but a simple definition if \(ap\) is problematic; for instance
\[ap := X \uparrow ; Y \uparrow ; X \uparrow Y \uparrow \text{ "strip of its brackets and } + \text{ it"} .\]

However we can easily express the required result of the particular application in which \(ap\) appears in \(ss\), for in general
\[yy \ cd \ xx \ ap = yy xx \ ap = yy xx \equiv [yy xx] = xx yy + .\]

So, in the definition
\[ss := (\_X:VAL. \ ap[X, cd[X]])' \equiv X \uparrow ; X \uparrow cd X \uparrow ap\]
we replace \(X \uparrow cd X \uparrow ap\) by \(X \uparrow X \uparrow + .\)

This definition is also suggested by the original requirement
\[xx \ ss = xx xx \equiv [xx xx] = [xx] xx + = xx xx + .\]

We thus find the following theorem. Let
\[ss := X \uparrow ; X \uparrow X \uparrow + \text{ or shorter } X \uparrow X \uparrow + , \]
\[cc := X \uparrow ; X \uparrow ss tt \text{ or shorter } ss tt , \]
\[fp := \text{decode}(cc \ ss) \equiv cc \ cc ,\]
then \(fp = \overline{fp} \ tt .\) Fully written out \(fp \equiv [X \uparrow X \uparrow + tt] X \uparrow X \uparrow tt .\)

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 \(y \overline{xx} ! = y \overline{xx} \)
and even $\bar{xx} = xx$, and we thought that

either $\phi_{\bar{xx}}$ had been transcripted systematically in $\bar{xx} uu \equiv \bar{xx}！$

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

$\bar{yy} xx \equiv [yy \; xx！]$.

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 $\bar{fp}$, viz. $\bar{cc} \; \bar{cc}！$ instead of $\bar{cc} \; \bar{cc}$.

Indeed

$ss := X \; \! ; \! [!] X \; \! + \! X \; \! + \!$

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

$cc := ss \; tt$,

$fp := \text{decode} \; (cc \; ss) \equiv \bar{cc} \; \bar{cc}！$,

we find

$fp \equiv \bar{x} ; \; [!] X \; \! + \! X \; \! + \! tt] \bar{x} ; \; [!] X \; \! + \! X \; \! + \! tt]！ = \bar{fp} \; tt$.

4.3. A formulation of theorem 3 in the lambda calculus

Let us take $\text{Txt}$ 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 $= \beta$ and is the reflexive and transitive closure of the well-known conversion rules, viz.

$\alpha: [\bar{xx} \; ee = \bar{yy}$ replace$(xx, ee, yy)$

$\beta: [\bar{xx} \; ee] \; aa = \text{replace}(xx, ee, aa)$

$\Pi: [\bar{xx} \; ee[xx]] = ee$

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 $\bar{yy}$).

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

$Val := \text{Txt}$,

$xx := xx$.

We thus find the following theorem. Let

$ss := \bar{x} \; X \; X[x]$
so that indeed $ss[xx] = xx[xx]$,

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

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

$$= [\downarrow X. tt[X[X]]] [\downarrow X. tt[X[X]]]$$

then $fp = tt[fp]$.

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

$$yy := \downarrow T. [\downarrow X. T[X[X]]] [\downarrow X. T[X[X]]].$$

Indeed, $fp \equiv "\beta\text{-rule applied to } yy[tt]"$.

Warning. One should not try to eliminate (by means of rule $\beta$) 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 $yy$ yields the 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 \textit{C}, \textit{L}, \textit{N} and \textit{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 \textit{xx}, the meaning of which is given by \textit{eval}(\textit{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

\[
\text{VAL} := \text{the set of S-expressions},
\]
\[
\text{Val} := \text{the set of denotations of VAL values = quoted S-expressions}.
\]

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

\[
\textit{xx} := \text{the denotation of the S-expression } \textit{xx} \equiv [\textit{Q} \textit{xx}].
\]

Now \textit{cd} and \textit{ap} should satisfy

\[
[\textit{cd} \textit{xx}] = \textit{xx} \equiv [\textit{Q} \textit{xx}] ,
\]
\[
[\textit{ap} \textit{xx} \textit{yy}] = [\textit{xx} \textit{yy}] ,
\]

so we may define

\[
\textit{cd} := [\textbf{L} \ [\textbf{X}] \ [\textbf{C} \ \textbf{Q} \ [\textbf{C} \ \textbf{X} \ \textbf{N}]])
\]
\[
\textit{ap} := [\textbf{L} \ [\textbf{X} \ \textbf{Y}] \ [\textbf{C} \ \textbf{X} \ [\textbf{C} \ \textbf{Y} \ \textbf{N}]])
\]

We thus have found a definition for \textit{ss}:

\[
\textit{ss} := [\textbf{L} \ [\textbf{X}] \ [\textit{ap} \ \textit{X} \ [\textit{cd} \ \textit{X}]])
\]

By the body replacement rule this may be simplified to

\[
\textit{ss} := [\textbf{L} \ [\textbf{X}] \ [\textbf{C} \ \textbf{X} \ [\textbf{C} \ \textbf{Q} \ [\textbf{C} \ \textbf{X} \ \textbf{N}] \ \textbf{N}])]
\]

which we could have found more directly from the original requirement

\[
[\textit{ss} \textit{xx}] = [\textit{xx} \textit{xx}] \equiv [\textit{xx} [\textit{Q} \textit{xx}]] .
\]

We thus find the following theorem. Let

\[
\textit{cc} := [\textbf{L} \ [\textbf{X}] [\textit{tt} [\textit{ss} \textit{X}]]) , \text{or simplified}
\]
\[
\textit{cc} := [\textbf{L} \ [\textbf{X}] [\textit{tt} [\textbf{C} \ \textbf{X} [\textbf{C} \ \textbf{Q} \ [\textbf{C} \ \textbf{X} \ \textbf{N}] \ \textbf{N}]]) , \text{and}
\]
\[
\textit{fp} := \text{decode } ([\textit{ss} \textit{cc}]) \equiv [\textit{cc} [\textit{Q} \textit{cc}]) ,
\]

Then \textit{fp} = [\textit{tt} [\textit{Q} \textit{fp}]].
Taking \( tt := [[[x] x]] \) (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 \( \text{evalquote}(fn, aa) \). Actually this is the form in which programs have to be supplied to the\footnote{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 \( \text{Val} := \text{the set of S-expressions} \).

The theorem will read \( fp = tt \ [fp] \) where \( fp \) is a pair. We choose to supply \( tt \) with a single argument, and thus define

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

The requirement for \( ss \) now reads

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

So we may define

\[
ss := [[[x] [C X [C \ [C X \ N] \ N]]]],
\]

and find the following theorem. Let \( cc := [[[x] [tt \ [C X [C \ [C X \ N] \ N]]]]] \), \( fp := \text{decode}(ss[cc]) \equiv cc \ [cc] \)
then \( fp = tt \ [\overline{[fp]}] \).

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


