# Expressions that talk about themselves

*Maarten Fokkinga, University of Twente, dept. INF,* fokkinga@cs.utwente.nl

Version of May 6, 1994

**Introduction** Self-reference occurs frequently in theoretical investigations of formal systems. In Predicate Logic self-reference is used in Gödels Incompleteness Theorem: a *formula* is constructed that asserts "I am not formally provable"; in a programming language self-reference manifests itself in the existence of a *program* that writes its own text, a so-called self-reproducing program. Also in Nature self-reference plays a role, in the ability of self-reproduction of the formal system of DNA molecules. Odifreddi [3, page 165–174] discusses these topics and gives good references to the literature.

In this note we explain, independent of any formalism, how to construct an expression in a formal system that "does something with itself". As an application of this procedure we show how to construct a program that writes its own text, lay-out included. (Our self-reproducing Miranda script is, after some optimisation, exactly 32 characters long!)

To appreciate the problems of self-reference, the reader may try and write a self-reproducing program from scratch. She will soon discover that the program tends to be infinitely long, and therefore isn't a program at all. This holds so even when recursion is used in an attempt to avoid the blow-up of the text:

```
program Self;
begin write ("program Self;
              begin write ( ....
                            ....
              end Self.")
end Self.
```

Moreover, some trick seems needed to write special symbols (like quotes). Our method needs no tricks, and works in any programming language, for any way to encode special characters within strings. The technique of self-reference and self-reproduction is by no means new, and very many solutions have been reported throughout the years, but nevertheless I hope the reader will enjoy this exposition.

## 1 How to obtain self-reference

**Notation** Identifiers in *italic font* and math symbols like $($, $)$, $[$, $]$, $/$, $\langle\!\langle$, $\rangle\!\rangle$ are used to denote expressions from the formal system. Expressions and symbols from the formal system itself are written in `type writer font`. We shall assume that $\mathtt{X}$ is a symbol of the formal system (a variable, maybe) that may occur in expressions; we will use it to indicate a place where a replacement has to take place.

**Coding and substitution** In order that an expression can "do something sensibly with itself", the formal system must provide a way to code expressions. For example, expressions might be coded as numerals, or as character sequences. The code of an expression $E$ is denoted $«E»$.

We assume that various operations on codes can be expressed in the formal system. For a programming language this comes as no surprise; for Predicate Logic this is not at all obvious (and we shall not attempt to prove that here). Specifically, we shall need a way to express a kind of substitution. So we assume that there exists a term $subst[\,,\,]$ (with two open places where codes can be put) such that for each code $c$ and expression $E$ the expression $subst[c, «E»]$ (that is, $subst[\,,\,]$ with the two places filled by $c$ and $«E»$) denotes the code of the expression that results from replacing the symbol $\mathtt{X}$ by $c$ in $E$:

$$subst[c, «E»] \quad \approx \quad «[c/\mathtt{X}]E» \quad .$$

The sign $\approx$ is semantic equivalence, meaning that the left-hand side and the right-hand side may be interchanged in all contexts, if only the 'standard meaning' of expressions and terms is considered. So, in Predicate Logic relation $\approx$ is formally provable equivalence, and in a programming language relation $\approx$ is value-equality. Actually, in all uses of $subst$ in the sequel the two arguments are equal. Therefore we will assume that $subst$ has only one open place; the property that we assume to hold of $subst$ thus reads:

$$subst[«E»] \quad \approx \quad «[«E»/\mathtt{X}]E» \quad .$$

Without loss of generality, we may assume that symbol $\mathtt{X}$ does not occur in $subst$.

**Self-reference** The expressibility of substitution is the key to obtaining self-reference. It allows an expression to *use a code several times* whereas it *occurs only once* in the expression. (In a programming language, there exist alternative means to obtain this effect, namely repetitive constructs like for-loops, list comprehension, and replication of strings [1, 2].) In this way the infinite blow-up as suggested for the self-reproducing program is avoided. Indeed, take $self$ and $code$ as follows:

$$self \quad = \quad subst[\boxed{« \langle \text{the surrounding text with } \mathtt{X} \text{ instead of this box} \rangle »}] \quad ,$$

or, written more formally:

$$\begin{aligned} self \quad &= \quad subst[code] \\ code \quad &= \quad «subst[\mathtt{X}]» \quad . \end{aligned}$$

Then we have that *self* is semantically equivalent to its own code:

$$
\begin{array}{ll}
 & self \\
= & \quad \{ \text{ definition } self \ \} \\
 & subst[code] \\
\approx & \quad \{ \text{ property } subst\text{, definition } code \ \} \\
 & \langle\!\langle [code/\mathtt{X}](subst[\mathtt{X}]) \rangle\!\rangle \\
= & \quad \{ \text{ substitution, } subst \text{ doesn't contain } \mathtt{X} \ \} \\
 & \langle\!\langle subst[code] \rangle\!\rangle \\
= & \quad \{ \text{ definition } self \ \} \\
 & \langle\!\langle self \rangle\!\rangle \quad .
\end{array}
$$

It only remains to adapt this device to other purposes.

**A general purpose self-reference construction** Let $Expr[\,]$ be an expression with an empty place ... where a code has to be put; we write $Expr[c]$ for $Expr$ with the empty place ... filled by $c$. Here are the two main examples:

$$
\begin{array}{lll}
Expr[\,] & \approx & \langle \text{a formula asserting that the formula with code ... is not provable} \rangle, \\
Expr[\,] & \approx & \langle \text{a program that writes the text of the program with code ...} \rangle \quad .
\end{array}
$$

We set out to construct an expression $Self$ that is semantically equivalent to statement $Expr$ about itself:

$$Self \quad \approx \quad Expr[\langle\!\langle Self \rangle\!\rangle] \quad .$$

For the example expressions above we thus get:

$$
\begin{array}{lll}
Self & \approx & \langle \text{a formula asserting that the formula with code } \langle\!\langle Self \rangle\!\rangle \text{ is not provable} \rangle, \\
Self & \approx & \langle \text{a program that writes the text of the program with code } \langle\!\langle Self \rangle\!\rangle \rangle \quad .
\end{array}
$$

Following the construction above of *self*, we take:

$$Self \quad = \quad Expr[subst[\boxed{\langle\!\langle \ \langle \text{the surrounding text with } \mathtt{X} \text{ instead of this box} \rangle \ \rangle\!\rangle}]] \quad .$$

or, written more formally:

$$
\begin{array}{lll}
Self & = & Expr[subst[code]] \\
code & = & \langle\!\langle Expr[subst[\mathtt{X}]] \rangle\!\rangle \quad .
\end{array}
$$

Indeed, assuming that $Expr$ doesn't contain $\mathtt{X}$, we find in exactly the same way as in the calculation above (Yes, try it!!) that $Self \approx Expr[\langle\!\langle Self \rangle\!\rangle]$, as desired.

The attentive reader may have noticed that up to now the coding $\langle\!\langle\ \rangle\!\rangle$ may be trivial. Both the identity and each constant function satisfy all of the assumptions, and the coding need not be invertible. However, if $Expr[c]$ is to do something nontrivial with $c$, it has to be able to retrieve from $c$ the constituent pieces of the expression encoded by $c$. It is only at this point that further properties of codes are relevant. (Thanks to Peter Asveld for this observation.)

(In Lisp, programs and data are the same, namely S-expressions, and it is possible to retrieve the constituent text parts from a program $E$ even if $E$ is *given as a program*. So, in Lisp we can take the identity as coding.)

# 2 Application: a self-reproducing program

Using the above technique for constructing self-referring expressions, we shall construct a self-reproducing program *Self* :

$$Self \quad \approx \quad \langle \text{a program that writes the text of the program with code } \langle\!\langle Self \rangle\!\rangle \rangle \quad .$$

Thus the formal system is a programming language.

Since in all programming languages a text can be written if it is first encoded as a string (a character sequence) and then subject to `write`, say, we choose the string encoding as the coding $\langle\!\langle\ \rangle\!\rangle$. Thus, writing is an inverse of coding, that is, for any text $x$:

$$\texttt{write}\ (\langle\!\langle x \rangle\!\rangle) \quad \text{yields} \quad x \quad .$$

For definiteness, let us assume that a text is coded as a string in the following way:

$$\begin{array}{lcl} \langle\!\langle x \rangle\!\rangle & = & \text{text } x \text{ with the replacement of each quote, newline, and backslash} \\ & & \text{by } \texttt{\textbackslash q},\ \texttt{\textbackslash n}, \text{ and } \texttt{\textbackslash b}, \text{ respectively,} \\ & & \text{and the whole enclosed by a quote } \texttt{"} \text{ at the left and at the right} \quad . \end{array}$$

For example:

$$\begin{array}{lcl} x & = & \texttt{aaa"bbb}\langle newline \rangle\texttt{ccc} \\ \langle\!\langle x \rangle\!\rangle & = & \texttt{"aaa\textbackslash qbbb\textbackslash nccc"} \\ \langle\!\langle\langle\!\langle x \rangle\!\rangle\rangle\!\rangle & = & \texttt{"\textbackslash qaaa\textbackslash bqbbb\textbackslash bnccc\textbackslash q"} \quad . \end{array}$$

We call the quote, backslash, and newline *the special symbols*, since these, and only these, undergo a change when subject to coding. In the sequel it will be obvious how to adapt our construction to alternative ways of encoding special symbols; for example, encoding a quote by doubling it, or encoding a quote by `\"` and a backslash by `\\`. Later on we shall use the concatenation $+\!\!+$ of strings; it is specified by the following property:

$$\langle\!\langle x \rangle\!\rangle +\!\!+ \langle\!\langle y \rangle\!\rangle \quad = \quad \langle\!\langle x\, y \rangle\!\rangle \quad .$$

In the right-hand side the texts $x$ and $y$ are juxtapositioned to form one long text.

**The program** Following the general purpose construction of a self-referring *Self*, we take:

$$\begin{array}{lcl} \mathit{Expr}[\,] & = & \texttt{program p; ...... ; begin write (...) end p.} \\ \mathit{Self} & = & \mathit{Expr}[\mathit{subst}[\mathit{code}]] \\ \mathit{code} & = & \langle\!\langle \mathit{Expr}[\mathit{subst}[\texttt{X}]] \rangle\!\rangle \quad . \end{array}$$

It remains to express in the programming language the expression $\mathit{subst}[...]$. For this we choose to use a procedure call `S (`*args*`, ...)`, where procedure `S` is declared within *Expr* in the declaration part of program `p`, and *args* is a collection of auxiliary arguments that we might need:

```
procedure S (parms; c: string): string;
(*  pre: c = «E»;  result: «[c/X]E»  *)
begin . . . end S;  .
```

In order to synthesise a definition for procedure `S`, we try and express in programming language terms its result for an argument $\mathit{code} = \langle\!\langle E \rangle\!\rangle$ where $E = E'\,\texttt{X}\,E''$ and `X` doesn't occur in $E', E''$:

$$\begin{array}{ll} & \text{result of } \texttt{S (}\mathit{args}\texttt{, }\mathit{code}\texttt{)} \\ = & \\ & \mathit{subst}[\mathit{code}] \\ = & \\ & \mathit{subst}[\langle\!\langle E \rangle\!\rangle] \\ = & \\ & \langle\!\langle [\mathit{code}/\texttt{X}]\,(E'\,\texttt{X}\,E'') \rangle\!\rangle \\ = & \\ & \langle\!\langle E'\ \mathit{code}\ E'' \rangle\!\rangle \\ = & \\ & \langle\!\langle E' \rangle\!\rangle \mathbin{+\!\!+} \langle\!\langle \mathit{code} \rangle\!\rangle \mathbin{+\!\!+} \langle\!\langle E'' \rangle\!\rangle \\ (*) \quad = & \qquad \{ \text{ defining } \mathit{firstpart} \text{ and } \mathit{lastpart} \text{ as specified below } \} \\ & \mathit{firstpart}\ \mathit{code} \mathbin{+\!\!+} \langle\!\langle \mathit{code} \rangle\!\rangle \mathbin{+\!\!+} \mathit{lastpart}\ \mathit{code} \quad . \end{array}$$

Notice that procedure `S` has to do an "extra" coding in the middle part of its result. For step $(*)$ operations *firstpart* and *lastpart* should satisfy:

$$\begin{array}{lcl} \mathit{firstpart}\,\langle\!\langle E'\,\texttt{X}\,E'' \rangle\!\rangle & = & \langle\!\langle E' \rangle\!\rangle \\ \mathit{lastpart}\,\langle\!\langle E'\,\texttt{X}\,E'' \rangle\!\rangle & = & \langle\!\langle E'' \rangle\!\rangle \quad . \end{array}$$

If it is possible to define these operations and $\langle\!\langle\ \rangle\!\rangle$ and $\mathbin{+\!\!+}$ (as is the case in most languages), their definitions can be placed before procedure `S`, in the declaration part of program `p`. (In Miranda, the pseudo-function `show` is equal to the coding $\langle\!\langle\ \rangle\!\rangle$.) However, we shall show in a while how to eliminate these operations; so they are not needed in the end. Having the operations available, it is easy to complete the definition of procedure `S`, and the complete program thus reads:

```
program p;
```

```
auxiliary procedures
procedure S (parms; c: string): string;
begin return firstpart c ++ «c» ++ lastpart c
end S;
begin write (S (args, «⟨the surrounding text with X instead of this box⟩»))
end p.
```

As said before, argument *args* may contain auxiliary data to ease the task of procedure `S`.

**Optimisation** It is possible to eliminate the need for operations ++, *firstpart*, *lastpart*, and procedures that implement the coding « ». We achieve this by an extreme form of *eager evaluation*; for several operations the time of evaluation is shifted from run-time through compile-time to construction-time of the program. (The extreme opposite is *lazy evaluation*, where the evaluation of operations is delayed as long as possible.)

First, instead of programming the disassembling of *code* within procedure `S`, *we ourselves* disassemble *code* and write the parts as separate arguments to `S`. This eliminates the need for disassembling procedures like *firstpart*. It also eliminates the need for symbol `X`; its sole purpose was to indicate one point where a replacement had to occur, and *we* know at which point.

Second, the need for the assembling operation ++ is eliminated by exploiting the following equivalence:

```
write («x» ++ «y»)   ≈   write («x»); write («y»)   .
```

So, instead of passing a concatenation of several codes to a write procedure, procedure `S` *itself* can perform write actions on the codes separately.

Together these simplifications also enable an easy way to implement the coding « » that `S` has to perform: we ourselves split *code* at each special symbol, and for each splitting point *we ourselves* know what special code has to be written and where, and so we let procedure `S` do just that.

To work out this idea in an illustrative way, we shall first assume that *the above program doesn't contain special symbols outside the box.* The optimisation then gives the following text:

```
program p;
auxiliary procedures
procedure S (parms; c0,c1: string): string;
begin write (c0);
      quote; write (c0); qcomma; write (c1); quote;  (* <-- the box *)
      write (c1)
end S;
```

```
begin S (args, " ⟨the text surrounding this box
               with ", " instead of this box⟩ ")
end p.
```

Here, *quote* and *qcomma* have to be defined in such a way that their effect is the writing of a quote `"` and a quoted comma `", "`, respectively. Notice that the box stands for *two* arguments, $code_0$ and $code_1$ say; the former *code* is equal to $code_0 +\!\!+ \langle\!\langle \mathtt{X} \rangle\!\rangle +\!\!+ code_1$.

There are several ways to define *quote* and *qcomma*.

The simplest way is to write the quote and quoted comma using the ASCII code. The ASCII codes do *not* contain special symbols, and if program `p` is written as one line, our simplifying assumption is true. Then we are readily done, and the *auxiliary procedures* and parameters *parms* and arguments *args* are not needed at all. However, we refrain from using this specific coding property. Instead, we strive for a generally applicable solution.

Another way is to *name* the quote and quoted comma in some way or another, and to use the names instead of the symbols themselves to write them. One option is to name them as constants, another option is to name *and write* them in a procedure, and yet another option is to name them via the argument-parameter binding of *args* and *parms*. Quite arbitrarily we choose the latter option. So we take:

```
args   =  ", ", "\q",...
parms  =  c,q: string,...   .
```

Then *quote* is: `write (q)`, and *qcomma* is: `write (q); write (c); write (q)`. However, this *args* does contain special symbols, and our simplifying assumption (no special symbols outside the box) is definitely invalid. So, now we really have to split the text surrounding the box at all special symbols, taking care that `S` writes the special symbols at the splitting points. We also have to include a naming for the backslash and newline. For brevity we say `write (` $x$`,` $y$`,...)` instead of `write (`$x$`); write (`$y$`); ...`. Thus we obtain our final program:

```
program p;
procedure S (c,q,bs,nl: string; c0,c1,...,c_{n+1}: string);
begin write (c0,sym_1,c1,...,sym_k,c_k);
      write (q,c0,q,c,q,c1,q,c,...,c,q,c_{n+1},q);     (* <-- the box *)
      write (c_{k+1},sym_{k+1},...,sym_n,c_{n+1})
end S;
begin S (", ", "\q", "\b", "\n",
         " ⟨the text surrounding this box with ", "
           instead of this box and of every special symbol⟩ ")
end p.
```

Here each $sym_i$ stands for `q`, `bs`, or `nl` according to the special symbol at which the $i$th splitting occurs; $k$ is the number of special symbols preceding the box, and $n$ is the total number of special symbols outside the box. Note that all $sym_i$ are texts without special symbols; were this not the case, we would have a hard time!

The box now stands for a series of codes, $code_0, code_1, \ldots, code_{n+1}$ say. The reader may wish to check that, assumed the program is written as one long line, we have:

$$\begin{array}{lcl} code_0 & = & \texttt{"}\langle\text{the text of program } \texttt{p} \text{ up to, excluding, the first quote}\rangle\texttt{"} \\ k, n & = & 11, 11 \\ code_{12} & = & \texttt{") end p."} \quad , \end{array}$$

and further:

| $sym_1$ | $code_1$ | $sym_2$ | $code_2$ | $sym_3$ | $code_3$ | $sym_4$ | $code_4$ | $sym_5$ | $code_5$ | $sym_6$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `q` | `", "` | `q` | `", "` | `q` | `""` | `bs` | `"q"` | `q` | `", "` | `q` |

| $code_6$ | $sym_7$ | $code_7$ | $sym_8$ | $code_8$ | $sym_9$ | $code_9$ | $sym_{10}$ | $code_{10}$ | $sym_{11}$ | $code_{11}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `""` | `bs` | `"b"` | `q` | `", "` | `q` | `""` | `bs` | `"n"` | `q` | `", "` |

Needless to say that further optimisation is quite well possible. Several codes are equal, some are empty, and since there are no newlines, parameter `nl` is nowhere used.

**Execise** Write a program that writes "`Yes, that was me.`" or "`Rubbish!`", if it reads its own text on a given file, or not, respectively.

# References


[1] M.M. Fokkinga. A self-reproducing Algol 60 program. *Algol Bulletin*, 35:24–26, 1973.

[2] M.M. Fokkinga. Some self-reproducing Algol-like programs and Kleene's recursion theorem formulated in concrete programming languages. Technical Report TW-memo 281, University of Twente, Enschede, Netherlands, September 1979.

[3] P. Odifreddi. *Classical Recursion Theory*, vol 125 of *Studies in Logic*. North-Holland, Amsterdam, 1989.