Technische Hogeschool Twente

MEMORANDUM NR. INF-84-3

A notation for the most general form of repetition.

Maarten M. Fokkinga.

February 1984

Department of Informatics
P.O. Box 217
7500 AE Enschede
The Netherlands.

Onderafdeling der Informatica

Contents

Abstract We present a notation with which all forms of repetition (like the well known while, repeat, Dijkstra's do od, Parnas'~~s~~ it ti, Zahn's events, multi-level exits, and so on) can be written in a very simple and uniform way.

Footnotes have been listed on the last page.

0 Introduction

Since the banishment of the *goto* there has been a continuing stream of proposals for particular forms of repetition. Among these are *repeat-until*, *while-do-until*, *exits* in the middle of a loop, multi-level exits, until-events (Zahn 1974), *do-od* (Dijkstra 1975), and so on. All these proposals are motivated with considerations concerning efficiency (no need for additional boolean variables, no need for extra tests), readability and methodology (Zahn's events), and provably correct program construction (Dijkstra's *do-od*).

Recently this list has been extended with the *it-ti* construct of (Parnas 1983); a generalisation of repetition, embracing both the *if-fi* and *do-od* of (Dijkstra 1975). It is Parnas' paper in the Communications of the ACM, that has prompted us to write this note: another notation for repetition which seems to be the ultimate in combining generality and simplicity. The underlying idea is simple: recursion and name-giving are the canonical way to denote a large class of infinite programs in a finite way. Though this is really folklore for some part of the Computing Science Community, it is apparent from the literature on the subject that the idea is not generally well-known.

In what follows we give a brief exposition of Parnas' proposal in Section 1; this may be skipped by readers already familiar with that construct. In Section 2 we present and motivate our notation; examples are dealt with in Section 3. Then the sections "Can't we improve upon it" and the Conclusion follow.

1 The it-ti and init construct

The *it-ti* construct (1) (Parnas 1983) is a combination of both the *if-fi* and *do-od* of (Dijkstra 1975, 1976). We assume the *if-fi* and *do-od* to be known. An example of the *it-ti* construct is as follows. Assume m and n being constants satisfying $m \leq n$;

$$\{i \leq m\}\ \underline{\text{it}}\ i<n \dashrightarrow i := i+1 \uparrow \square\ m \leq i \dashrightarrow \underline{\text{skip}} \downarrow\ \underline{\text{ti}}\ \{m \leq i \leq n\}$$

The execution proceeds as follows. At least one of the guards must evaluate to *true*, and its fellow command is then obeyed. If several guards evaluate to true, then a nondeterministic choice is made. The delimiter ↑ respectively ↓ indicates whether the repetition is to be continued or terminated. (The example program establishes $m \leq i \leq n$; and any such value of i may be achieved by appropriate nondeterministic choices. In (Fokkinga 1978 a) this property is called "pure": the program is pure with respect to precondition $i \leq m$ and postcondition $m \leq i \leq n$.)

It is not difficult to express if-fi or do-od by means of it-ti. Take all delimiters ↓ and you get if-fi. The do-od can be got by taking all delimiters ↑ and adding the following line:

```
else --> skip ↓
```

An important gain of the it-ti is that both the continuation conditions and the termination conditions have to be expressed separately. Usually one is the negation of the other. The advantage of separately mentioning them has also been formulated by (Fokkinga 1978 b).

Somewhat independent of the it-ti is Parnas' init construct. The expression init is a boolean constant denoting true during the first turn of the repetition, and false in subsequent turns. (2). Using init both duplication of text and redundant tests may be avoided. An example, isomorphic to the one presented by Parnas, reads as follows. The integer variable exp is to be initialized to 2**k where k is a natural number to be read from input. The user gets at most three chances to type in an integral number.

```
n := 0;
it not init cand k ≥ 0  --> exp := 2**k ↓
elif init or n < 3 --> k := read; n := n + 1 ↑
elif true --> skip ↓
ti
{exp initialised iff k ≥ 0}
```

Here and in the sequel we use elif as an abbreviation of " ▯ (not any of the preceding conditions) and", and cand doesn't evaluate its second operand if the outcome is already determined by its first operand ("conditional and").

2 The all-embracing notation

Before reflecting on notations for repetition, let us first have a look at programs in general. In principle a program is built from

assignments: x := e

by means of the compositions

sequencing: prog0; prog1

choice: <u>if</u> b0 --> prog0 [] ... [] bn --> progn <u>fi</u>

(possibly with the abbreviations <u>elif</u>, <u>else</u>, <u>cand</u> and so on, as mentioned in Section 1). With an obvious notation there is this great disadvantage: the computation invoked by a program is at most as long as the program text itself. Therefore we introduce two notations in order to overcome this. The most important is a notation to denote infinite programs (so with possibly infinite computations) in a finite way. The appropriate means is recursion. The notation

```
idf :: prog
```

(called a <u>recursive construct</u>) denotes the infinite program that is obtained by repeatedly replacing the identifier idf in prog by this very same construct. Let expr[x/expr'] denote the substitution of expr' for all occurences of x in expr. Then we have by definition that idf::prog is semantically equivalent to

prog[idf / idf :: prog]

and that the infinite program equals

prog[idf / prog[idf / prog[idf / ...]]].

Here is a simple example: summation of 1 through n.

```
s := 0; i := 0;
sumFurther ::
    if i = n --> skip
    [] i < n --> i := i + 1; s := s + i; sumFurther
    fi
```

This text denotes the infinite program

```
s := 0; i := 0;
if i = n --> skip
[] i < n --> i := 1; s := s + i;
             if i = n --> skip
```

```
                    ▯ i < n --> i := i + 1; s := s + i;
                                if i = n --> skip
                                ▯ i < n --> i := i + 1; s := s + i;
                                                .
                                                .
                                                .
      fi          fi          fi
```

Accidentally, all possibly invoked computations are finite; had we written $i \neq n$ instead of $i < n$, then the computation invoked for negative values of n would have been infinite and nonterminating.

It is almost trivial to represent the do-od and it-ti in this notation. Schematically it looks as follows.

```
      do ... ▯ b --> prog ▯ ... od                becomes
do :: if ... ▯ b --> prog; do ▯ ... ▯ else --> skip fi

      it ... ▯ b --> prog ↑ ▯  ... ▯ b' --> prog' ↓ ▯  ... ti   becomes
it :: if ... ▯ b --> prog; it ▯ ... ▯ b' --> prog' ▯ ... fi
```

But there are more possibilities. Terminating and restarting of repetitions on higher levels is possible; examples are given in the next section. Also the proof rules do not essentially differ from conventional rules for repetition. In order to prove {P}idf::prog{Q} it is sufficient to prove {P}prog{Q} assuming that for each occurrence of idf in prog, {P}idf{Q} holds. In such a proof (P,Q) is called the invariant pair; in the above example an invariant pair is (s=1+2+...+i, s=1+2+...+n).

Actually the recursive construct is a parameterless recursive procedure which is written in-place without being given a name in a preceding definition. In case of tail recursion or last action recursion (that is, idf occurs only as a very last command in prog) the recursive construct can be implemented exactly as a conventional repetitive construct, see (Knuth 1974), (Er 1983). Actually (Sussman 1982) shows that such an implementation needs <u>no</u> specific optimization phases in the compilation process!

The second part of our notation of programs is name-giving. The notation

```
def idf = prog in prog'
```

denotes the program which is obtained from prog' by substituting prog for each occurrence of idf: prog'[idf/prog]. Please note that the notation does not mean that first prog is executed and afterwards prog'. In contrary, in prog' idf is "called-by-name"!

3 Examples

Because of the simple representation of do-od by means of our recursive construct, we may quote all of (Dijkstra 1976) as an illustration of the use of our notation. That is quite a bit, (and for some that will do). The transformation of it-ti into our notation is even more simpler. In addition we present five more examples.

Example 0 Multi-level exit
It is requested to print the 1st, 3rd, 6th, 10th, 15th ... number of the input, up to end-of-file (eof).

```
i:=0;
restInput ::
  ({the i-th number has just been printed}
    var j:=i;
    skipJNumbers ::
         if eof --> skip {multi-level exit!}
         elif j=0 --> print(read); i:=i+1; restInput
         elif {j>0} --> read; j:=j-1; skipJNumbers
         fi
  )
```

Another example of multi-level exits arises naturally when programming a linea~~i~~r search over a two-dimensional matrix; cfr. the next example.

Example 1 Linea~~i~~r search
The program below is the direct trans~~ition~~lation of a scheme often found. Actually the scheme is a simple application of Zahn's event facilities; see also the next example.

```
i:=1;
```

```
continueSearch ::
    def found = print ("x occurs at position ", i)
    ;   notFound = print ("x doesnot occur")
    in if i=n+1  --> notFound
      elif a[i]=x --> found
      elif {a[i]≠ x} --> i:=i+1; continueSearch
       fi
```

Example 2 Zahn's events

The reader may consult (Zahn 1974) for the motivation and explanation of the events construct, as well as for a nontrivial example. Also (Knuth 1974) strongly supports it and gives various remarkable applicatons. The construct allows ~~to~~ escape from the middle of a command sequence in a loop. We must admit that a direct transliteration of the scheme

```
---; if B0 --> SL0 ▯  B1 --> SL1 ▯  B --> . fi;  SL .
```

requires SL to be named and invoked twice (or we could duplicate SL):

```
loop ::
    def tail = SL; loop
    in  ---; if B0 --> SL0; tail ▯ B1 --> SL1; tail  ▯ B --> skip fi
```

Knuth dislikes that name-giving (page 271 of (Knuth 1974)). But name-giving might have its benefits too, for readability. Thus we would write his example 6g (which uses Zahn's events) as follows.

```
{The set of triples (A[t],L[t],R[t]) represents the nodes of a binary
 tree; nil is represented by zero. The binary tree t is to be
 printed in pre-order.}
 stack S; S := empty;
 printTAndS::
     def LtProcessed = print(A[t]); t:=R[t]; printTAndS
     in if t ≠ 0 -->  gotoLeftMostLeaf::
                          if L[t] = 0  -->  skip
                          ▯  L[t] ≠ 0  -->  S<==t; t:=L[t];
                                            gotoLeftMostLeaf
                          fi;
                      LtProcessed
        elif nonempty(S) -->  t<==S; LtProcessed
```

```
        elif true --> skip
        fi
```

Example 3 More than normal iteration

In this illustration we exploit proper recursion. In integer variable m we count the number of moves required for the towers of Hanoi of height n. We annotate the program with a semi-formal correctness proof.

```
m := 0;
hanoi {pre n=n0, m=m0; post m=m0+2**n0-1, n=n0}::
      if n=0 --> skip {m=m0=m0+2**n0-1 qed}
      □  n>0 --> n := n-1; {n=n0-1, m=m0}
                 hanoi; {n=n0-1, m=m0+2**(n0-1)-1}
                 m := m+1; {m=m0+2**(n0-1)}
                 hanoi; {n=n0-1, m=m0+2**(n0-1)+2**(n0-1)-1}
                 n := n+1 {n=n0, m=m0+2**n0-1 qed}
      fi
```

Example 4 The init construct

We now present in our notation the example which motivated the use of init, see Section 1. The consecutive n := 0 and n := n+1 have been combined into n := 1 ; apart from that the execution of this program has exactly the same tests and assignments as the program from Section 1.
performs

```
def try = k:=read
in
   ( try; n := 1;
     completeTrial ::
         if k ≥ 0 --> exp := 2**k
       elif n<3 --> try; n := n+1; completeTrial
       elif {n=3} --> skip
         fi
   )
```

The duplication of text, namely the three occurrences of the identifier try, is comparable with the two occurrences of init in the original example. In addition there is now no need for cand, and "readability" has surely not got worse. As said before, Parnas' example is isomorphic to the one above, see fig. 12b in (Parnas 1983).

4 Can't we improve upon it?

If we generalize the recursive construct to allow for multiple mutual recursion, then there is no problem in transliterating arbitrary flow charts, or transition diagrams (Reynolds 1978), into our notation. Basically the program becomes a multiple recursive construct, with components uniquely corresponding to the labels and calling each other as a last action; see (Knuth and Floyd 1971). Thus no additional variables or computation steps are introduced. (One should of course not expect that this improves the program in any respect.) (It is folklore that multiple recursion can be expressed in single recursion in exchange for a duplication of text.)

Nevertheless programs denoted with the recursive construct have a very regular, periodical structure. More complex infinite programs are concievable and sometimes needed as well. We think that the computations invoked by such programs can not be named "repetitions". Despite it we present a notation for them by way of curiosity.

To this end we introduce paramaterized programs:

(fct x. prog)

is an anonymous parameterized program. The so-called application (fct x. prog)(a) is by definition equivalent to

def x=a in prog

The identifier x may belong to an arbitrary syntactic category; e.g. x may be an expression identifier or a program identifier and so on. (Tennent 1981) shows that the notations (fct x. prog) and (fct x. prog)(a) arise from the Principles of Abstraction and Qualification. In this way we actually get at our disposal the (possibly typed) lambda calculus. We do not pursue this topic here; the interested reader is referred to (Landin 1966), (Reynolds 1981) and (Fokkinga 1983).

5 Conclusion

We have considered programs as compositions of assignments by means of sequencing and choice, and a notation has been presented to denote a class of infinite programs. Apparently all forms of repetitions treated in the literature appear to be expressible very simply in this notation.

However we do not claim that this work contains anything original or surprising. For example, (Hehner 1979) arrives at a similar notation, but from quite another viewpoint. Our recursive construct is also a notational variant of the mu-construct of (de Bakker and de Roever 1973); it might even be said that it is just a disguised application of the fixed point combinator of the lambda calculus. Notwithstanding all that it really is disappointing that still notations are being proposed for only particular and sometimes peculiar forms of repetition.

References

de Bakker, J.W. and de Roever, W.P.: A calculus for recursive program schemes. In Proc. 1st ICALP (ed M.Nivat), North-Holland, Amsterdam (1973) pp 167-196.

Dijkstra, E.W.: Guarded commands, nondeterminacy and the formal derivation of programs. Comm ACM 18 (1975) 8, pp 453-457.

Dijkstra, E.W.: A discipline of programming. Prentice Hall, Englewood Cliffs N.J., (1976).

Er, M.C.: Optimizing procedure calls and returns. Software-Practice and Experience 13 (1983) 10, pp 921-940.

Fokkinga, M.M.: Another difference between recursive refinement and repetition. Internal report, Twente Univ. of Technology, (1978a).

Fokkinga, M.M.: A note on the pragmatics of the repetitive construct. Internal report, Twente Univ. of Technology (1978b).

Fokkinga, M.M.: Structuur van Programmeertalen. (In Dutch). Lecture notes, Twente Univ. of Technology (1983).

Hehner, E.C.: do considered od - a contribution to programming methodology. Acta Informatica 4 (1979) 11, pp 287-305.

Knuth, D.E., Floyd, R.W.: Notes on avoiding goto statements. Information Processing Letters 1 (1977) 1, pp 23-31, 177.

Knuth, D.E.: Structured programming with goto statements. Computing Surveys 6 (1974) 4, pp 261-301.

Landin, P.J.: The next 700 programming languages. Comm ACM 9 (1966) 3, pp 157-166.

Parnas, D.L.: A generalized control structure and its formal definition.

Comm ACM 26 (1983) 8, pp 572-581.

Reynolds, J.C.: Programming with transition diagrams. In Programming Methodology (ed. D. Gries). Springer-Verlag, Berlin etc, (1978) pp 153-165.

Reynolds, J.C.: The essence of Algol. In Algorithmic Languages (eds. J.W. deBakker, J.C. vanVliet). North Holland, Amsterdam, (1981) pp 345-372.

Sussman, G.J.: Notes on Lisp. In Functonal programming and its applications (eds. J. Darlington, P. Henderson, D. Turner). Cambridge Univ Press (1982)

Tennent, R.D.: Principles of programming Languages. Prentice-Hall, Englewood Cliffs N.J. (1982).

Zahn, C.: A. control statement for natural topdown structured programming. In Proc. Symposium on Programming (ed B. Robinet). LNCS 19 (1974) pp 170-180

---

(1) We only consider iteration over select-lists, and write the select-separator as a bar [] so as to stress the correspondence with Dijkstra's guarded commands. Parnas also mentions iteration over jury lists, but apart from the definition he doesn't discuss them at all (That is "left to future research").

---

(2) One might be tempted to introduce the dual of <u>init</u>, namely <u>last</u>. However this imposes serious problems on the implementer, as shown by

<u>it</u> <u>last</u> --> <u>skip</u> ↑ [] <u>not</u> <u>last</u> --> <u>skip</u> ↓ <u>ti</u>