Programming in circular arithmetics

Andrzej Blikle
February 19th, 2026

1 An intuitive description of the problem to solve

We are given a circular scale with divisions labelled from 0 to 99, and a pointer fixed at the center of the circle that may be turned clockwise or anticlockwise by any number of integers. Assume that a sequence of integers describes a history of pointer rotations such that positive numbers denote clockwise rotations, and negative numbers anticlockwise. The programming task is to write a program, which, given an initial position of the pointer and a history of its rotations, will compute the number of passages of the pointer by 0 and the number of rotations that terminated at 0. We assume that each rotation, except the first, starts where the previous rotation ended.

Eric Wastl originally formulated this problem at a site https://adventofcode.com/2025/day/1. Thank you to Tomek Gieorgijewski for sharing this information.

2 Building a mathematical model

Whenever we intend to describe a phenomenon by means of a program, we have, in the first place, to build an appropriate mathematical model of that phenomenon. By a *model* we mean in this case a many-sorted algebra, i.e.:

1. a many-sorted universe, i.e., a family of domains,
2. some functions (constructors) between these domains.

The constructors of a model may be defined either explicitly, i.e., algorithmically, or implicitly, i.e., by axioms. We usually prove additional lemmas about constructors, which may be more handy in program derivation than the definitions or axioms themselves. In this example, we will first describe our model in **MetaSoft.**

Once we have a model, we translate its definitions, axioms, and lemmas into **Lingua-T** to store them in the repository. Based on it, we derive our program. In practice, the development of the triad:

(model, repository, programs)

is not built one by one in this order. Since it may be hard to predict in advance all domains and functions, which we shall need (especially the latter), the three components of the triad are built layer-by-layer, adding new functions and possibly also new domains, when necessary. We shall proceed in our example in this way as well.

To begin with, we introduce four domains:

pos	: Position	= {0,…,99}	positions of the pointer
rot	: Rotation	= Integer	rotation of the pointer[1]
int	: Integer		integers as defined in **Lingua-V**
tra	: Trace	= Rotation^{c*}	trace of rotations (may be empty)
boo	: Boole	= {tt, ff}	

The first two functions that we introduce describe single steps in the rotation of the pointer:

[1] Formally we could have introduced just Integers as one of our domains, but we introduce Rotations to help reading our programs.

```
suc : Position ⟼ Position                    successor: the next position in a clockwise rotation
suc.pos = if pos < 99 then pos+1 else 0 fi

pre : Position ⟼ Position                    predecessor: the next position in an anticlockwise rotation
pre.pos = if pos > 0 then pos-1 else 99 fi
```

In a general case, a rotation of a pointer may be described by a certain number of full rotations of 100 steps, plus a residuum of less than 100 steps. To decompose rotations in this way, we introduce two further functions: one computes the integer division by 100, and the second — the residuum of such division:

```
div100 : Rotation ⟼ Integer                  the integer part of division by 100 (the number of full rotations)
div100.rot =
   if rot = 0
      then 0
      else
         if rot > 0
            then
               if rot < 100
                  then rot
                  else 1 + div100.(rot - 100)     # rot ≥ 100
               fi
            else                                  # rot < 0
               if rot > -100
                  then rot
                  else 1 + div100.(rot + 100)
               fi
         fi
   fi

res100 : Rotation ⟼ {0, 1, … ,99}            the residuum of integer division by 100
res100.rot = rot – div100.rot*100
```

Since we intend to build a correct metaprogram that counts the number of passes by zero and the number of stops at zero, we must be able to formulate appropriate postconditions that describe these facts. For this purpose, we define functions that compute these numbers. Since we assume that each rotation starts where the last has ended, we also need a function that computes the last position of the pointer.

```
Last : Position x Rotation ⟼ Position        the last position of the pointer after rotation rot
Last.(pos, rot) =
   if rot ≥ 0
      then
         if pos + res100.rot > 99
            then pos + res100.rot – 100
            else pos + res.rot
         fi
      else # rot < 0
         if pos + res100.rot < 0
            then pos + res100.rot + 100
            else pos + res100.rot
         fi
   fi
```

Our functions that compute the numbers of passes and stops are defined as follows:

```
NoPass : Position x Rotation ⟼ Integer       the number of passes over zero during one rotation
NoPass.(pos, rot)  =
   if rot = 0
```

```
        then
           if pos = 0 then 1 else 0 fi
        else
           if rot > 0
              then
                 if pos = 0 then 1 + NoPass.(suc.pos, rot-1) else NoPass.(suc.pos, rot-1) fi
              else # rot < 0
                 if pos = 0 then 1 + NoPass.(pre.pos, rot+1) else NoPass.(pre.pos, rot+1) fi
     fi
```

```
  NoPassTra : Position x Trace ⟼ Integer      the number of stops at zero during the execution of a trace
  NoPassTra.(pos, tra) =
     if empty.tra
        then
           0
        else
           NoPass.(pos, head.tra) + NoPassTra.(Last.(pos, head.tra), tail.tra)
     fi
```

```
  NoEnd : Position x Rotation ⟼ Integer      the number of stops at zero in one rotation (it may be 0 or 1)
  NoEnd.(pos, rot) =
     if rot = 0
        then
           if pos = 0 then 1 else 0 fi
        else
           if rot > 0
              then
                 NoEnd.(suc.pos, rot-1)
              else # rot < 0
                 NoEnd(pre.pos, rot+1)
           fi
     fi
```

```
  NoEndTra : Position x Trace ⟼ Integer      the number of stops at zero in the execution of a trace
  NoEndTra.(pos, tra) =
     if empty.tra
        then
           0
        else
           NoEnd.(pos, head.tra) + NoEndTra.(Last.(pos, head.tra), tail.tra)
     fi
```

Traces in our model will be represented by lists. We will need four functions on lists:

```
  head    : Trace ⟼ Rotation | Error     takes the first element of the trace
  tail    : Trace ⟼ Trace | Error        removes the first element of the trace
  length  : Trace ⟼ Integer              returns the length of the trace
  empty   : Trace ⟼ {tt, ff}             checks if a trace is empty
```

We do not need to define them, since they are available in **Lingua-V.**

3 Building a formalized theory

So far, we have defined some preliminaries of our mathematical model. Let's call it *circular arithmetics*. The definitions of its functions have been written in **MetaSoft.** Now we have to build a formalized theory, such that circular arithmetics is one of its models (cf. Sec. 10 of [2]). To do that, we have to reformulate our definitions into **Lingua-T**.

To begin with, we assume that the repository already contains all axioms for the operations on integers that we have used, i.e., +, -, <, ≤, >, ≥, and =. We recall that they should be axioms characterizing operations on typed integers, as defined by the semantics of **Lingua-V** (Sec. 4.3 in [2]), rather than abstract integers.

Now, we have to express in **Lingua-T** the definitions of all functions defined in Sec. 2. Before we do that, we have to enrich our language with the symbols of these functions, which means that we have to add the following typed patterns to the repository of temporary schemes (see [1] for explanation):

```
(vex, suc(vex))
(vex, pre(vex))
(vex, div100(vex))
(vex, res100(vex))
(vex, last(vex1, vex2))
(vex, NoPass(vex1, vex2))
(vex, NoPassTra(vex1, vex2))
(vex, NoEnd(vex1, vex2))
(vex, NoEndTra(vex1, vex2))
(vex, Lead(vex1, vex2))                                    definition in Sec. 4
```

Having enriched **Lingua-T,** we can express in it the definitions of our functions, e.g.:

```
vex is integer ⇨
    suc(vex) = if vex < 99 then vex + 1 else 0 fi
```

or

```
(vex1 is integer) and-k (vex2 is list(integer)) ⇨
    NoPassTra.(vex1, vex2) =
        if empty.vex2
            then
                0
            else
                NoPass.(vex1, head.vex2) + NoPassTra.(Last.(vex1, head.vex2), tail.vex2)
        fi
```

In both cases, our axioms are metaimplications whose premises describe the signatures of the defined functions. Note in this place that the equational definitions of our functions can't be axioms, also because they are not formulas, but terms.

Regarding functions on traces, i.e., on lists — head, tail, length, and empty — we do not need to define them, since they are primitive concepts in **Lingua-V.** Still, we will need some lemmas about them to support the derivation of our programs. To begin with, we formulate the following lemma that describes functions empty and tail:

Lemma 1

```
pre ide is-v list(integer) and-k (not empty.ide) :
    while not empty(ide)
        do
            ide:= tail(ide)
        od
post empty(ide)
```

Of course, formally, we have to prove it on the grounds of the semantics of **Lingua-T**.

We also introduce a series of lemmas concerning the previously defined functions. For the reader's convenience, we formulate each lemma in two versions: in **MetaSoft,** which is more intuitive, and in **Lingua-T.**

Lemma 2 *If* rot = 0, *then for every position* pos

```
NoPass.(pos, rot)  = if pos = 0 then 1 else 0 fi
NoEnd.(pos, rot)   = if pos = 0 then 1 else 0 fi
```

Last.(pos, rot) = pos

(vex1 **is** integer) **and-k** (vex2 **is** integer) **and-k** (vex2 = 0) ⇨
NoPass.(vex1, vex2) = **if** vex1= 0 **then** 1 **else** 0 **fi** **and-k**
NoEnd.(vex1, vex2) = **if** vex1= 0 **then** 1 **else** 0 **fi** **and-k**
Last.(vex1, vex2) = vex1

Lemma 3 *If* rot > 0, *then for every position* pos,

NoPass.(pos, rot) = div100(rot) + NoPass.(pos, res100.rot)
NoEnd.(pos, rot) = NoEnd.(pos, res100.rot)
Last.(pos, rot) = **if** pos + res100.rot > 99 **then** pos + res100.rot – 100 **else** pos + res.rot **fi**

(vex1 **is** integer) **and-k** (vex2 **is** integer) **and-k** (vex2 > 0) ⇨
NoPass.(vex1, vex2) = div100(vex2) + NoPass.(vex1, res100(vex2)) **end-k**
NoEnd.(vex1, vex2) = NoEnd.(vex1, res100(vex2)) **end-k**
Last.(vex1, vex2) = **if** vex1 + res100(vex2) > 99 **then** vex1 + res100(vex2) – 100 **else** vex1 + res.vex2 **fi**

Lemma 4 *If* rot > 0, *then for every position* pos,

NoPass.(pos, res100.rot) = **if** pos = 0 **then** 1 **else if** pos + res100.rot > 99 **then** 1 **else** 0 **fi fi**
NoEnd.(pos, res100.rot) = **if** pos + res100.rot = 100 **then** 1 **else** 0 **fi**

(vex1 **is** integer) **and-k** (vex2 **is** integer) **and-k** (vex2 > 0) ⇨
NoPass.(vex1, res.100.vex2) = **if** vex1 = 0 **then** 1 **else if** vex1 + res100.vex2 > 99 **then** 1 **else** 0 **fi fi**
NoEnd.(vex1, res100.vex2) = **if** vex1 + res100.vex2 = 100 **then** 1 **else** 0 **fi**

Lemma 5 *If* rot < 0, *then for every position* pos,

NoPass.(pos, rot) = div100(- rot) + NoPass.(pos, res.rot)
NoEnd.(pos, rot) = NoEnd.(pos, res100.(rot))
Last.(pos, rot) = **if** pos + res100.rot < 0 **then** pos+res100.rot+100 **else** pos+res100.rot **fi**

(vex1 **is** integer) **and-k** (vex2 **is** integer) **and-k** (vex2 < 0) ⇨
NoPass.(vex1, vex2) = div100(- vex2) + NoPass.(vex1, res100(vex2))
NoEnd.(vex1, vex2) = NoEnd.(vex1, res.(vex2))
Last.(vex1, vex2) = **if** vex1 + res100(vex2) < 0 **then** vex1+res100(vex2)+100 **else** vex1+res100(vex2) **fi**

Lemma 6 *If* -99 ≤ rot ≤ 0, *i.e.,* res.rot = rot, *then for every position* pos,

NoPass.(pos, res100.rot) = **if** pos = 0 **then** 1 **else if** pos + res100.rot < 99 **then** 1 else 0 **fi fi**
NoEnd.(pos, res100.rot) = **if** pos + res100.rot = 0 **then** 1 **else** 0 **fi**

(vex1 **is** integer) **and-k** (vex2 **is** integer) **and-k** (-99 ≤ vex2 ≤ 0) ⇨
NoPass.(vex1, res100(vex2)) = **if** vex1 = 0 **then** 1 **else if** vex1 + res100(vex2) < 99 **then** 1 else 0 **fi fi**
NoEnd.(vex1, res100(vex2)) = **if** vex1 + res100(vex2) = 0 **then** 1 **else** 0 **fi**

We also recall Lemmas 9.4.3-1 and 9.4.3-2 from Sec. 9.4.3 of [2], which we will need. We formulate these lemmas as implicative formulas in **Lingua-T,** and we recall (Sec. 10.7.3 of [1]) that they give rise to corresponding inference rules.

Lemma 7

pre (con1 **and-k** vex) : sin1 **post** con2
pre (con1 **and-k** (**not-k** vex)): sin2 **post** con2
con1 ⇨ (vex **or-k** (**not-k** vex))
---
pre con1 : **if** vex **then** sin1 **else** sin2 **fi post** con2

Lemma 8

(1) **pre** (con3 **and-k** vex) : sin **post** con3
(2) con3 **insures LR of asr** vex **rsa** ; sin
(3) con1 ⇨ con3
(4) con3 ⇨ (vex **or-k (not-k** vex**)**)
(5) con3 **and-k (not-k** vex**)** ⇨ con2

---

pre con1 **: asr** vex **rsa** ; **while** vex **do** sin **od post** con2

4 The derivation of the program

We shall build our program in two phases:

- the derivation of the code that executes a single rotation,
- the deviation of a loop that executes a trace, rotation-by-rotation.

In the derivation of the program, we will generate, derive, or prove a series of concrete, i.e., grounded, formulas in **Lingua**-T, most frequently metaprograms, but also a few metaimplications. All these formulas will be formally lemmas, but in this section, we are not labelling them in this way, to distinguish them from lemmas of Sec. 3. In our metaprograms, we shall use the following variables:

- rot — the current rotation to be executed,
- pos — the starting position of the current rotation,
- posPro — the starting position of the whole program (constant),
- zp — the number of passes by zero in the current rotation
- ze — the number of endings at zero in the current rotation
- ac_zp — the accumulated number of passes by zero,
- ac_ze — the accumulated number of endings by zero,
- step — the current number of executed rotations,
- lp — the last position after current rotation,
- tra — the remaining trace to be executed,
- traPro — the initial trace of the program (constant).

To make our future programs look compact, we introduce the following parameters in the repository of final parameters (they won't be modified):

```
basic     :: (rot, pos, posPro, zp, ze, ac_zp, ac_ze, lp, step is-v integer)  and-k
             (tra, traPro is-v list(integer)                                  and-k
             (0 ≤ pos, posPro ≤ 99)                                           and-k
             step ≤ length(traPro)

initial   :: basic      and-k
             step = 0   and-k
             zp = 0     and-k
             ze = 0

finalRot  :: zp = NoPass.(pos, rot)     and-k
             ze = NoEnd.(pos, rot)      and-k
             lp = Last.(pos, rot)

finalPro  :: zp = NoPassTra(posPro, traPro)  and-k
             ze = NoEndTra.(posPro, traPro)
```

finalPro is the intended postcondition of our target program. basic summarises the information about the types and ranges of variables. It will be perpetual in all our programs (see Sec. 9.3.6 of [1]) derived later. Our first two concrete metaprograms may be derived from Lemma 2:

(1) **pre** basic **and-k** rot = 0 **and-k** pos = 0:
    zp := 1; ze := 1; lp := pos

post basic **and-k** finalRot

(2) **pre** basic **and-k** rot = 0 **and-k** pos ≠ 0:
zp := 0; ze := 0; lp := pos
post basic **and-k** finalRot

The derivation of each of them requires a few steps, which we leave to our readers. **Hint**: consult Sec. 9.4.3.2 of [1], and also the example in Sec. 10.3 of [1]. Now, we use Lemma 7 to sequentially combine these two programs. In this step we apply the following substitutions:

basic **and-k** rot = 0 → prc
pos = 0 → vex
basic **and-k** finalRot → poc
zp := 1; ze := 1; lp := pos → sin1
zp := 0; ze := 0; lp := pos → sin2

To apply Lemma 7, we have to prove

(3) basic **and-k** rot = 0 ⇨ (pos = 0 **or-k** (**not-k** pos = 0))

which essentially means that if basic is satisfied, then pos = 0 evaluates cleanly. From (1), (2), and (3), using Lemma 6, we derive:

(4) **pre** basic **and-k** rot = 0 **:**
if pos = 0 **then** zp := 1; ze := 1; lp = pos **else** zp := 0; ze := 0; lp := pos **fi**
post basic **and-k** finalRot

From Lemma 3 and Lemma 4, we derive:

(5) **pre** basic **and-k** rot > 0 :
zp := div100(rot) + **if** pos = 0 **then** 1 **else if** pos + res100(rot) > 99 **then** 1 **else** 0 **fi fi**;
ze := **if** pos + res100(rot) = 100 **then** 1 else 0 **fi**
lp := **if** pos + res100(rot) > 99 **then** pos + res100(rot) – 100 **else** pos + res100(rot) **fi**
post basic **and-k** finalRot

From Lemma 5 and Lemma 6, we derive:

(6) **pre** basic **and-k** rot < 0 :
zp := div100(- rot) + **if** pos = 0 **then** 1
else
if pos + res100(rot) < 99
then 1
else 0
fi
fi;
ze := **if** pos + res.rot = 100
then 1
else 0
fi
lp := **if** pos + res100(rot) < 0
then
pos + res100(rot) + 100
else
pos + res100(rot)
fi
post basic **and-k** finalRot

We introduce three more parameters, this time pointing to specinstructions:

sin of (4) → **if** pos = 0 specified instruction of program (4)
then zp := 1; ze := 1; lp = pos

```
                    else zp := 0; ze := 0; lp := pos fi
sin of (5) → zp := div100(rot) + if pos = 0
                                 then 1
                                 else if pos + res100(rot) > 99 then 1 else 0 fi fi;
             ze := if pos + res100(rot) = 100 then 1 else 0 fi
             lp  := if pos + res100(rot) > 99 then pos + res100(rot) – 100 else pos + res100(rot) fi
sin of (6) → zp := div100(- rot) + if pos = 0 then 1 else if pos + res100(rot) < 99 then 1 else 0 fi fi;
             ze := if pos + res.rot = 100 then 1 else 0 fi
             lp  := if pos + res100(rot) < 0 then pos + res100(rot) + 100 else pos + res100(rot) fi
```

To put (5) and (6) together using Lemma 7, we substitute

```
basic and-k rot ≠ 0   → prc
rot > 0               → vex
basic and-k finalRot  → poc
sin of (5)            → sin1
sin of (6)            → sin2
```

This time, to apply Lemma 7, we have to prove

(7) basic **and-k** (rot ≠ 0) ⇨ ((rot > 0) **or-k** (**not-k** rot > 0))

From (5), (6), (7) and Lemma 7 we derive:

```
(8) pre basic and-k (rot ≠ 0) :
      if rot > 0
        then
          sin of (5)
        else
          sin of (6)
      fi
    post basic and-k finalRot
```

To put (4) and (8) together using Lemma 7, we substitute:

```
basic                 → prc
rot = 0               → vex
basic and-k finalRot  → poc
sin of (4)            → sin1
sin of (8)            → sin2
```

We prove an auxiliary lemma:

(9) basic ⇨ ((rot = 0) **or-k** (rot ≠ 0))

From (4), (8), (9), and Lemma 4, we derive:

```
(10) pre basic :
       if rot = 0
         then
           sin of (4)
         else
           if rot > 0
             then
               sin of (5)
             else
               sin of (6)
           fi
       fi
     post basic and-k finalRot
```

In this way, we have derived a program that realizes our goal for one rotation.

To build a program that computes the sums of the results for a trace of rotations, we have to use Lemma 8 and, therefore, define appropriate substitutions for metavariables. Let's start by introducing a new parameter

```
sin of (10) :: if rot = 0
                  then
                     sin of (4)
                  else
                     if rot > 0
                        then
                           sin of (5)
                        else
                           sin of (6)
                     fi
               fi
```

With this parameter metaprogram (10) may be written in a concise form:

```
(10a) pre basic :
         sin of (10)
      post basic and-k finalRot
```

The next step is to define appropriate substitutions to Lemma 8. We start with a new parameter that points to the invariant of the while loop. To define it, we have to introduce a new function that, given a positive integer int and a trace tra, returns a trace consisting of the first int elements of tra or an error message:

```
Lead : Integer x Trace ⟼ Trace | Error
Lead.(int, tra) =
   if int = 0
      then
         ()
      else
         if empty.tra
            then
               'trace is empty'
            else
               push head(tra) to Lead(int-1, tail(tra)) sup
         fi
   fi
```

With this function, we associate the following lemma:

Lemma 9 *For any trace* tra *and* int ≥ 0:

```
NoPassTra.(pos, Lead.(0, tra)) = 0
NoEndTra.(pos, Lead.(0, tra) = 0
```

To define substitutions to Lemma 8, we introduce two new parameters:

```
invariant ::  basic                                        and-k
              step ≤ length(traPro)                        and-k
              zp = NoPassTra(posPro, Lead(step, traPro))   and-k
              ze = NoEndTra(posPro, Lead(step, traPro))    and-k
              step = length(tra)

body      ::  rot := head(tra) ;
              tra := tail(tra) ;
              pos := lp ;
              sin of (10) ;                                computes zp, ze and lp for current rotation
              ac_zp := ac_zp + zp ;
```

```
        ac_ze := ac_ze + ze ;
        step := step + 1
```

We need the variable step in this program exclusively to be used in the invariant. At the end of the program development process, step may be eliminated. The substitutions are now the following:

```
initial          → con1
finalPro         → con2
invariant        → con3
not empty(tra)   → vex
body             → sin
```

With these substitutions completed, we have to prove/derive metaconditions (1) – (5) of Lemma 8. Let's investigate them one-by-one:

ad. (1) **pre** (invariant **and-k** step > 0) : body **post** invariant

The corresponding proof refers to (10) and is carried by induction on the step.

ad. (2) invariant **insures LR of asr not** empty(tra) **rsa** ; body

Proof refers to Lemma 1, i.e., to the basic property of lists.

ad. (3) initial ⇨ invariant

Proof refers to Lemma 9.

ad. (4) invariant ⇨ (**not** empty(tra)) **or-k** empty(tra)

invariant guarantees that tra is a trace and, therefore, empty(tra) evaluates cleanly.

ad. (5) invariant **and-k** empty(tra) ⇨ finalPro

This metaimplication follows from the following facts:

```
empty(tra) ⇨ step = 0
Lead(0, traPro) = traPro
NoPassTra(posPro, Lead(0, traPro)) = NoPassTra(posPro, traPro)
NoEndTra(posPro, Lead(0, traPro)) = NoPassTra(posPro, traPro)
```

As a conclusion of Lemma 8, we can now derive the next program

(11)
```
pre initial :
    asr not empty(tra) rsa ;
    while not empty(tra)
        do
            body
        od
post finalPro
```

Applying Rule 2 about the omission of assertions in Sec. 10.7.4.2 of [2], we derive:

(12)
```
pre initial :
    while not empty(tra)
        do
            body
        od
post finalPro
```

Our last step consists of adding such declarations and initialization instructions in front of the while instruction, that will make the condition initial satisfied at the end:

(13)
```
pre (pos, posPro is-v integer) and-k (0 ≤ pos, posPro ≤ 99) :
    let rot, zp, ze, ac_zp, ac_ze, lp, step be integer ;
    let tra, traPro be list(integer) ;
```

```
        step := 0 ;
        zp := 0 ;
        ze := 0
    post initial
```

Combining sequentially (13) with (12), we get:

(14)
```
pre (pos, posPro is-v integer) and-k (0 ≤ pos, posPro ≤ 99) :
    let rot, zp, ze, ac_zp, ac_ze, lp, step be integer ;
    let tra, traPro be list(integer) ;
    step := 0 ;
    zp := 0 ;
    ze := 0 ;
    while not empty(tra)
        do
            body
        od
post finalPro
```

Finally, we unfold all parameters, thus getting an implementable version of our program:

(15)
```
pre (pos, posPro is-v integer) and-k (0 ≤ pos, posPro ≤ 99) :
    let rot, zp, ze, ac_zp, ac_ze, lp, step be integer ;
    let tra, traPro be list(integer) ;
    step := 0 ;
    zp := 0 ;
    ze := 0 ;
    while not empty(tra)
        do
            if rot = 0
                then
                    if pos = 0
                        then
                            zp := 1; ze := 1; lp = pos
                        else
                            zp := 0; ze := 0; lp := pos
                    fi
                else
                    if rot > 0
                        then
                            zp := div100(rot) + if pos = 0 then 1 else if pos + res100(rot) > 99 then 1 else 0 fi fi;
                            ze := if pos + res100(rot) = 100 then 1 else 0 fi
                            lp  := if pos + res100(rot) > 99 then pos + res100(rot) – 100 else pos+res100(rot) fi
                        else
                            zp := div100(- rot) + if pos = 0 then 1 else if pos+res100(rot) < 99 then 1 else 0 fi fi;
                            ze := if pos + res.rot = 100 then 1 else 0 fi
                            lp  := if pos + res100(rot) < 0 then pos + res100(rot) + 100 else pos + res100(rot) fi
                    fi
            fi
        od
 post zp = NoPassTra(posPro, traPro)  and-k ze = NoEndTra.(posPro, traPro)
```

5 Some remarks at the end

One crucial problem associated with program correctness is a lack of an adequate language of task specifications, i.e., a language:

- sufficiently clear for future users for the program to express their expectations,
- sufficiently formal (unambiguous) for future programmers.

Of course, for each domain of applications, one must consider a language dedicated to it. Let's have a look at our exercise from this perspective.

In this case, a potential future user of the program may be a mechanical engineer designing an encrypted lock for a strongbox. We may assume that such a user has a fairly good understanding of mathematics, but will she or he be able to write the recursive definitions of our target functions, NoPassTra, and NoEndTra? And if not, then in which unambiguous way can we describe the fact that the pointer of our device has passed by or stopped at zero? Is there a non-algorithmic way to describe these facts?

The reader has also probably noticed that the definitions of our two main functions can be regarded as recursive declarations of functional procedures and, therefore, can be used directly as program code. Here we are again faced with the question of whether our specification is what the user really meant. What if our functions do not compute what they are supposed to? We leave these questions open, hoping that our readers will share their remarks on this issue.

6 References

[1] A. Blikle, *Lingua-T — a language of a growing D-Theory*, (a manuscript)

[2] A. Blikle, P. Chrząstowski-Wachter, J. Jabłonowski, A. Tarlecki, *A Denotational Engineering of Programming Languages*, a book in statu nascendi available at https://moznainaczej.com.pl/what-has-been-done/the-book.