Programming in circular arithmetics

Andrzej Blikle
February 27th, 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, but not by 0. Assume that a sequence of such integers describes a history of pointer rotations where 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 Three steps on the way to the target program

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. To incorporate this model into **Lingua-T**, we will build it as an extension of AlgDen-V, with new constructors and possibly new carriers.

New constructors may be defined either explicitly, i.e., algorithmically, or implicitly, i.e., by axioms. In the case of algorithmic definitions, we usually prove additional lemmas about constructors, which may be more handy in program derivation than the definitions themselves. In our example, we define new constructors algorithmically using **MetaSoft**.

Once we have a model, we express definitions, axioms, and lemmas in **Lingua-T** to store them in the repository. We assume that the **D-Theory** is already in the repository and therefore add only new axioms. After having completed **D-Theory**, we derive our program.

Summing up, we will split the derivation of a program into three stages:

1. building a model, i.e., extending the algebra AlgDen-V,
2. extending **D-Theory** by new functional symbols and new axioms; here we also extend **Lingua-T**,
3. deriving the target program based on the extended theory.

Since it may be hard to predict in advance all domains and functions (especially functions) we may need to derive our program, the three components of the triad are built layer by layer, adding new functions and possibly new domains when necessary. Also, in our example, we shall proceed in this way.

In Sections 3 and 4, we present two alternative solutions for building our target program, depending on how we initially formulate the task specification.

3 Recursive approach

3.1 Building a model

To begin with, we introduce three specific domains:

pos : Position = {0,…,99} positions of the pointer
rot : Rotation = Integer – {0} rotation of the pointer; rotations must be different from zero
tra : Trace = Rotation^{c*} trace of rotations (may be empty)

The first two functions correspond to single steps in the rotation of the pointer:

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 — in each full rotation, zero is passed exactly once — plus a residuum of less than 100 steps, where zero may be passed or not. To decompose rotations in this way, we introduce two further functions: one computes the integer division by 100, and the second, the residuum of this division:

div100 : Rotation ⟼ Integer the integer part of division by 100 (the number of full rotations)

```
div100.rot =
   if rot > 0
      then
         if rot < 100
            then 0
            else 1 + div100.(rot – 100)    # rot ≥ 100
         fi
      else                                  # rot < 0
         if rot > –100
            then 0
            else –1 + div100.(rot + 100)
         fi
   fi
```

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

Since we assume each rotation starts where the previous one ends, we need a function that computes the pointer's final position after a single rotation.

last : Position x Rotation ⟼ Position the last position of the pointer after rotation rot

```
last.(pos, rot) =
   if rot > 0
      then # rot > 0
         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
```

lastTra : Position x Trace ⟼ Position the last position of the pointer after the execution of trace tra

```
lastTra.(pos, tra) =
   empty.tra    ➔ pos
   true         ➔ lastTra.(last.(pos, head.tra), tail.tra)
```

Since we intend to build a correct metaprogram that counts the number of passes and 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, in each case for one rotation and for the whole program. We also have to decide whether starting from 0 may be counted as passing by zero. Let's assume that the answer is: NO. This, in particular, means that if we start from 0 and rotate 100, then the number of passes is 1.

```
noPass.(pos, rot) =                          the number of passes over zero during one rotation   (3.1-1)
  if rot > 0
    then
      if rot ≥ 100
        then
          if res100.rot = 0 then div100.rot else div100.rot + noPass.(pos, res100.rot) fi
        else     # 0 < rot < 100
          if pos + rot ≥ 100 then 1 else 0 fi
      fi
    else     # rot < 0
      if rot ≤ − 100
        then
          if res100.rot = 0 then − div100.rot else − div100.rot + noPass.(pos, res100.rot) fi
        else     # - 100 < rot < 0
          if pos + rot ≤ 100 then 1 else 0 fi
      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
```

In the second definition, we have used the functions head and tail, which are available in Lingua-V, so we do not need to define them. We also assume (arbitrarily) that if a rotation starts from position 0, then we may recognize that the pointer passed by 0.

```
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 rot = 1
        then
          if pos + 1 = 100 then 1 else 0 fi
        else # rot > 1
          noEnd.(suc.pos, rot-1)
      fi
    else # rot < 0
      if rot = -1
        then
          if pos – 1 = 0 then 100 else 0 fi
        else
          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
```

We assume that traces in our model are represented as lists. We, therefore, will use the following four functions on the lists available in **Lingua-V**:

```
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
```

Additionally, we introduce a predicate that ensures that there are no zeros in the trace and an operation of list concatenation:

```
noneZero : Trace ⟼ {tt, ff}
©        : Trace x Trace ⟼ Trace
```

At this time, we have defined 12 functions that we expect to use in developing our program. However, these functions can't be regarded (yet) as new constructors in AlgDen-V, since they are not functions on denotation but on integers and their lists. They are functions at the same level of abstraction as arithmetical operations plus and minus versus the corresponding constructors of the denotations of value expressions (Sec. 6.4.2 of [2]). To incorporate our new functions AlgDen-V and **Lingua-T**, we need to define corresponding constructors for denotations. Their signatures will be the following (prefix ved- stands for "value-expression denotation):

```
 1. ved-suc         : ValExpDen                ⟼ ValExpDen
 2. ved-pre         : ValExpDen                ⟼ ValExpDen
 3. ved-div100      : ValExpDen                ⟼ ValExpDen
 4. ved-res100      : ValExpDen                ⟼ ValExpDen
 5. ved-last        : ValExpDen x ValExpDen    ⟼ ValExpDen
 6. ved-lastTra     : ValExpDen x ValExpDen    ⟼ ValExpDen
 7. ved-noPass      : ValExpDen x ValExpDen    ⟼ ValExpDen
 8. ved-noPassTra   : ValExpDen x ValExpDen    ⟼ ValExpDen
 9. ved-noEnd       : ValExpDen x ValExpDen    ⟼ ValExpDen
10. ved-noEndTra    : ValExpDen x ValExpDen    ⟼ ValExpDen
11. ved-concatenate : ValExpDen x ValExpDen    ⟼ ValExpDen
12. ved-noneZero    : ValExpDen                ⟼ ValExpDen
```

We recall that:

```
ved : ValExpDen = WfState ⟶ Value | Error
```

These functions should be defined in a standard way as described in Sec. 6.4.2 [2]. Let's see one example:

```
ved-noPass : ValExpDen x ValExpDen ⟼ ValExpDen          i.e.                    (3.1-2)
ved-noPass : ValExpDen x ValExpDen ⟼ WfState ⟶ Value | Error
ved-noPass.(ved-p, ved-r).sta =                  p- stands for "position", r- stands for "rotation"
  is-error.sta          ➔error.sta
  ved-p.sta # ??        ➔??
  ved-p.sta : Error     ➔ ved-p.sta
  ved-r.sta # ??        ➔??
```

```
ved-r.sta : Error              ➔ ved-p.sta
let
   (dat-p, typ-p) = ved-p.sta
   (dat-r, typ-r)  = ved-r.sta
typ-p ≠ 'integer'              ➔ 'position must be an integer'
typ-t ≠ 'integer'              ➔ 'rotation must be an integer'
dat-p < 0 or-m dat-p > 99      ➔ 'position out of scope'
dat-r = 0                      ➔ 'rotation must not be zero'
let
   noOfPass = noPass.(dat-p, dat-r)
true                           ➔ (noOfPass, 'integer')
```

3.2 Extending Lingua-T and D-Theory

When we are done with the extension of AlgDen-V, we may proceed to the extension of **Lingua-T** by new symbols and **D-Theory** by new axioms. We assume, for simplicity, that the new syntax will be "spelled" in the style of primary syntax (Sec. 7.2 of [2]), with one exception for concatenation @.

In our example, we extend the syntax (grammar) with new value-expression constructors. Since in an eco-system, grammars are represented by lists of typed patterns, we have to add one such pattern for each of our new functions:

1. (vex, suc(vex))
2. (vex, pre(vex))
3. (vex, div100(vex))
4. (vex, res100(vex))
5. (vex, last(vex1, vex2))
6. (vex, lastTra(vex1, vex2))
7. (vex, noPass(vex1, vex2))
8. (vex, noPassTra(vex1, vex2))
9. (vex, noEnd(vex1, vex2))
10. (vex, noEndTra(vex1, vex2))
11. (vex, vex1 © vex2)
12. (vex, noneZero(vex))

The extension of the function of the semantics of value expressions to new patterns is again standard, e.g.,

```
SEM.vex : ValExp ⟼ ValExpDen
SEM.vex[ noPass(vex1, vex2) ] = ved-noPass.(SEM.vex[ vex1 ], SEM.vex[ vex2 ])
```

Having enriched **Lingua-T** — i.e., its syntax and semantics — we should now appropriately extend our repository of lemmas, i.e., **D-Theory**. A natural way to start this process is to "translate" the definitions of function in **MetaSoft** (in the model) to definitions written in **Lingua-T.** Below, we see two examples of such definitions:

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

and

```
(vex1 is integer) and-k (vex2 is integer) and-k 0 ≤ vex1 ≤ 99 and-k vex2 ≠ 0 ⇨                (3.2-1)
   noPass(vex1, vex2) =
      if vex2 > 0
         then
            if vex2 ≥ 100
               then
                  if res100.vex2 = 0 then div100.vex2 else div100.vex2 + noPass.(vex1, res100.vex2) fi
               else       # 0 < vex2 < 100
```

```
                if vex1 + vex2 ≥ 100 then 1 else 0 fi
        fi
        else      # vex2 < 0
            if vex2 ≤ − 100
                then
                    if res100.vex2 = 0 then − div100.vex2 else − div100.vex2 + noPass.(vex1, res100.vex2) fi
                else      # - 100 < vex2 < 0
                    if vex1 + vex2 ≤ 100 then 1 else 0 fi
            fi
    fi
```

In both cases, our axioms are metaimplications with premises that describe the signatures and the ranges of definability of the corresponding functions. It may be worth mentioning in this place that equations like, e.g.,

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

can't be regarded as axioms, because in **Lingua-T** they are not formulas, but terms.

A few comments are necessary here about extending a formal theory by defining a new functional symbol.

If the definition of such a symbol is grammatically correct, such an extension is always "safe" in the sense that it never makes the modified theory inconsistent (cf. [3] page 207). The theory still has a model, although this model differs from the former — say from AlgDen-V — by having one more constructor.

From the theoretical perspective, we are on the safe side, but practically, we can't be satisfied with the fact that our new theory has "a" model. We want this model to be the AlgDen-V with the new constructor defined by (3.1-2). However, we have a technical problem, since (3.2-1) says anything about the "behavior" of the denotation of noPassTra(vex1, vex2) for states that do not satisfy the premise, whereas (3.1-2) does. There seem to be two alternative solutions in this situation:

1. extending (3.2-1) by additional clauses corresponding to the behavior of the function for states not satisfying the premise of (3.2-1),
2. accepting the fact that (3.2-1) identifies a class of models, rather than just one (up to isomorphism), differing from each other on functions ved-noPass.

The first solution is more complicated, since it requires an extension of **Lingua-T** that allows the description of the mechanisms of abstract errors and of ??-undefinedness of functions. The second is simpler, but in that case, we pay a price: an incomplete definition of our function.

Since in our approach to program correctness we primarily focus on cases where programs "behave correctly", we choose the second solution, leaving the first for future research. That is, however, not the end of the story. We still have to prove that every ved-noPass in all extended models satisfies (3.1-2) for states that satisfy the condition

```
(vex1 is integer) and-k (vex2 is integer) and-k 0 ≤ vex1 ≤ 99 and-k vex2 ≠ 0 .
```

To prove that, we have to refer not only to (3.1-2) but also to (3.1-1). A formal proof of that fact may be quite technical.

After having introduced all definitional axioms corresponding to new functions, we may consider introducing further axioms that are more "handy" for program development. Of course, all of them must be proved either on the basis of the model (outside the ecosystem) or on the basis of **D-Theory** (within the ecosystem). To keep our paper within reasonable bounds, we skip all these proofs. We start from a "foundational" axiom for lists:

Lemma 1

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

post empty(ide)

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

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

```
noPass.(pos, rot)  = if res100.rot = 0
                     then div100.rot
                     else div100.rot + noPass.(pos, res100.rot) fi
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) = if res100.vex1 = 0
                        then div100(vex2)
                        else div100(vex2) + noPass(vex1, res100(vex2)) fi          and-k
   noEnd(vex1, vex2) = noEnd(vex1, res100(vex2))                                   and-k
   last(vex1, vex2)   = if vex1 + res100(vex2) > 99 then vex1 + res100(vex2) – 100 else vex1 + res.vex2 fi
```

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

```
noPass.(pos, res100.rot)  = if pos + res100.rot > 99 then 1 else 0 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, res100.vex2)  = if vex1 = 0 then 1 else if vex1 + res100.vex2 > 99 then 1 else 0 fi fi   and-k
   noEnd(vex1, res100.vex2)   = if vex1 + res100.vex2 = 100 then 1 else 0 fi
```

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

```
noPass.(pos, rot)  = if res100.rot = 0
                     then – div100.rot
                     else – div100.rot  + noPass.(pos, res.rot) fi
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))                                  and-k
   noEnd(vex1, vex2)    = noEnd(vex1, res(vex2))                                                        and-k
   last(vex1, vex2)     = if vex1 + res100(vex2) < 0 then vex1+res100(vex2)+100 else vex1+res100(vex2)  fi
```

Lemma 5 *If* -99 ≤ rot ≤ 0, *i.e.,* res.rot = rot, *then for every position* pos*,*

```
noPass.(pos, res100.rot)  = if pos + res100.rot < 99 then 1 else 0 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   and-k
   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 6

```
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 7

(1) **pre** (con3 **and-k** vex) : sin **post** con3
(2) con3 **ensures 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 **: while** vex **do** sin **od post** con2

At the end, one methodological remark is in order. So far, we have extended Lingua-T "on paper"; shall we implement all functions? There seem to be two alternative solutions again:

1. extending the implementation of **Lingua-V,** i.e., its parser and interpreter/compiler,
2. defining new functions as functional procedures expressed in **Lingua-V**.

Approach 1 may be worth considering for large or domain-specific projects, where we may wish to "tune" **Lingua-V** to some particular needs. Still, in our example, the second approach seems more appropriate. In both cases, we postpone implementing our functions until our program is ready, since only then will we know which functions are used in the algorithmic part of the program. E.g., in our example, functions noPassTra and noEndTra will be used only in the postcondition, hence we do not need to implement them.

3.3 The derivation of the program

We shall build our program in two phases:

- the derivation of the code that executes a single rotation,
- the derivation of a loop that uses this code to execute a trace of rotations.

In the derivation of the program, we generate, derive, or prove a series of concrete, i.e., grounded, formulas in **Lingua-**T, most frequently metaprograms. All these formulas will be formally lemmas, but in this section, we are not calling them in this way, to make a distinction from lemmas of Sec. 3.2. We will also skip their proofs to keep this section in a reasonable size. We adopt a pragmatic approach similar to a "working mathematician" who practically never develops formalized proofs.

In our metaprograms, we shall use the following variables:

- pos — the starting position of the current rotation,
- posPro — the starting position of the whole program (phantom constant),
- tra — the remaining trace to be executed,
- traPro — the initial trace of the program (phantom constant).
- lead — the currently elaborated trace,
- rot — the current rotation to be executed,
- 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,
- lp — the last position at the end of the current rotation,

In practical situations, this list will usually grow dynamically as the program is developed, but for the reader's convenience, we list them above. We start by introducing the following acronyms and storing them in the repository of final acronyms (they won't be modified):

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

```
                empty(lead)                                  and-k
                (posPro is-p integer)                        and-k      phantom constant
                (traPro is-p list(integer))                             phantom constant

initial     ::  basic                 and-k
                not empty(tra)        and-k
                not empty(traPro)     and-k
                traPro = tra          and-k
                zp = 0                and-k
                ze = 0                and-k
                ac_zp = 0             and-k
                ac_ze = 0

finalRot    ::  basic                       and-k
                zp = noPass(pos, rot)       and-k
                ze = noEnd(pos, rot)        and-k
                lp = last(pos, rot)

finalPro    ::  ac_zp = noPassTra(posPro, traPro)  and-k
                ac_ze = noEndTra(posPro, traPro)
```

finalPro is the intended postcondition of our target program. basic summarises the information about the types and the ranges of variables. It will be perpetual in all our programs (see Sec. 9.3.6 of [1]) derived later. From lemmas 2 and 3, we derive our first metaprogram:

```
(1) pre basic and-k rot > 0 :
      zp := div100(rot) + if pos + res100(rot) > 99 then 1 else 0 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 4 and Lemma 5, we derive further:

```
(2) pre basic and-k rot < 0 :
      zp := div100(- rot) + if pos + res100(rot) < 99 then 1 else 0 fi ;
      ze :=  if pos + res.rot = 0 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 two new acronyms pointing to their specinstructions:

```
sin of (1)  → zp := div100(rot) + if pos + res100(rot) > 99 then 1 else 0 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 (2)  → zp := div100(- rot) + if pos + res100(rot) < 99 then 1 else 0 fi
              ze := if pos + res.rot = 0 then 1 else 0 fi
              lp  := if pos + res100(rot) < 0 then pos + res100(rot) + 100 else pos + res100(rot) fi
```

Now, we use Lemma 6 to put (1) and (2) together. To do that, we define the following substitution:

```
basic and-k rot ≠ 0   → con1
rot > 0               → vex
basic and-k finalRot  → con2
sin of (1)            → sin1
sin of (2)            → sin2
```

To apply Lemma 6, we have to prove the metaimplication

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

This lemma is satisfied, since whenever basic is satisfied, the expression rot > 0 evaluates cleanly. From (1), (2), (3), and Lemma 6, we derive:

(4) **pre** basic **and-k** (rot ≠ 0) :
if rot > 0 **then** sin of (1) **else** sin of (2) **fi**
post basic **and-k** finalRot

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

sin of (4) :: **if** rot > 0 **then** sin of (1) **else** sin of (2) **fi**

Then, the metaprogram (4) may be written in a concise form:

(4a) **pre** basic **and-k** (rot ≠ 0) :
sin of (4)
post basic **and-k** finalRot

The next step is to use Lemma 7 to derive a while-loop. This loop should "consume" one-by-one all rotations of the current trace, getting the head of the trace as the current rotation and making its tail the next current trace. The loop should also accumulate the sum of successive values of zp and ze on variables ac_zp and ac_ze (ac stands for "accumulated"). After completing each step, the loop should calculate the next starting position. Under these assumptions, we define the following acronym that points to the body of the future loop:

```
body  :: rot := head(tra) ;
         tra := tail(tra) ;
         sin of (4) ;                          computes zp, ze, and lp for the current rotation
         ac_zp := ac_zp + zp ;
         ac_ze := ac_ze + ze ;
         lead := lead © (rot) ;
         pos := lp
```

In Lemma 7, this instruction will be substituted for sin.

It is now pretty clear that the condition of the loop should be (**not** empty(tra)). The next challenge is to find an invariant of the loop to be substituted for con3. Our guess is:

```
invariant :: basic                          and-k
             ac_zp = noPassTra(posPro, lead) and-k
             ac_ze = noEndTra(posPro, lead)  and-k
             lp    = lastTra(posPro, lead)   and-k
             traPro = lead © tra
```

Our substitutions associated with Lemma 7 are, then, the following:

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

With these substitutions assumed, we have to prove/derive metaconditions (1) – (5) of Lemma 7. Let's investigate them one by one:

ad. (1) **pre** invariant **and-k** (**not** empty(tra)) : body **post** invariant

This metaprogram should be derived from (4) and the definition of body.

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

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

ad. (3) initial ⇨ invariant

Proof by a simple check.

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

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

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

This metaimplication follows from the fact that if tra is empty, then lead = traPro, hence

```
ac_zp = noPassTra(posPro, traPro)     and-k
ac_ze = noPassTra(posPro, traPro)
```

As a conclusion of Lemma 7, we can derive our next program:

(5)
```
pre initial and-k not empty(tra) :
    while not empty(tra)
        do
            body
        od
post finalPro
```

Our next step is to add such declarations and initializations before the while-instruction, which will ensure the condition initial satisfied before entering the loop. We, therefore, build a metaprogram where:

- the precondition is restricted to "input variables" — pos and tra,
- the remaining variables and constants are declared; they are local variables,
- the condition initial is a postcondition.

(6)
```
pre (pos is-v integer) and-k (tra is-v list(integer)) and-k (0 ≤ pos ≤ 99) and-k (not empty(tra)) and-k noneZero(tra):
    let rot, zp, ze, ac_zp, ac_ze, lp is-v integer tel ;
    let lead is-v list(integer) tel ;
    let phantom posPro be list(integer) = pos tel ;
    let phantom traPro be list(integer) = tra tel ;
    zp := 0 ;
    ze := 0 ;
    ac_zp := 0 ;
    ac_ze := 0 ;
    lead := ()                                          the initialization by an empty list
post initial
```

In this program, we declare two phantom constants. Their initial values will never change. Combining sequentially (6) with (5), in this order, we get our target metaprogram with acronyms:

(7)
```
pre (pos is-v integer) and-k (tra is-v list(integer)) and-k (0 ≤ pos ≤ 99) and-k (not empty(tra)) and-k noneZero(tra):
    let rot, zp, ze, ac_zp, ac_ze, lp be integer tel ;
    let lead is-v list(integer) tel
    let phantom posPro be list(integer) = pos tel;
    let phantom traPro be list(integer) = tra tel;
    zp := 0 ;
    ze := 0 ;
    ac_zp := 0 ;
    ac_ze := 0 ;
    lead := () ;
    asr initial rsa ;                  # this assertion does not "intervene" in the execution; its role is only informative
    while not empty(tra)
        do
            body
        od
post finalPro
```

Finally, we unfold the acronyms, thus getting an implementable version of our metaprogram:

(8)
```
pre (pos is-v integer) and-k (tra is-v list(integer)) and-k (0 ≤ pos ≤ 99) and-k (not empty(tra)) and-k noneZero(tra):
    let rot, zp, ze, ac_zp, ac_ze, lp be integer tel ;
    let lead is-v list(integer) tel
```

```
    let phantom posPro be list(integer) = pos tel;
    let phantom traPro be list(integer) = tra tel;
    zp := 0 ;
    ze := 0 ;
    ac_zp := 0 ;
    ac_ze := 0 ;
    lead := () ;
    asr initial rsa ;
    while not empty(tra)
       do
          rot := head(tra) ;
          tra := tail(tra) ;
          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 + res100.rot = 100 then 1 else 0 fi
                lp  := if pos + res100(rot) < 0 then pos + res100(rot) + 100 else pos + res100(rot) fi
          fi ;
          ac_zp := ac_zp + zp ;
          ac_ze := ac_ze + ze ;
          lead := lead © (rot) ;
          pos := lp
    od
  post zp = noPassTra(posPro, traPro)  and-k ze = noEndTra(posPro, traPro)
```

Note now that since pos is not referred to in the first three assignments after the **if**-instruction, we can move the assignment pos := lp backwards to each of the two branches. In each of them, we will have, therefore, two following adjacent assignments:

```
lp := vex ;
pos := lp
```

We may replace these two assignments with equivalent ones:

```
pos := vex ;
lp := pos
```

Since lp does not appear anywhere else, neither in the program nor in the postcondition, we can remove the instruction lp := pos from the program, and lp from the declaration of variables.

Further on, we observe that lead does not "contribute" to the calculations of the values of other variables, except itself, and, therefore, also this variable may be removed from the program.

Finally, under Rule 2 in Sec. 10.7.4.2 of [2], we may remove the assertion **asr** initial **rsa** from the program. We get in this way:

(9)
```
pre (pos is-v integer) and-k (tra is-v list(integer)) and-k (0 ≤ pos ≤ 99) and-k (not empty(tra)) and-k noneZero(tra):
    let rot, zp, ze, ac_zp, ac_ze be integer tel ;
    let phantom posPro be list(integer) = pos tel;
    let phantom traPro be list(integer) = tra tel;
    zp := 0 ;
    ze := 0 ;
    ac_zp := 0 ;
    ac_ze := 0 ;
```

```
    while not empty(tra)
        do
            rot := head(tra) ;
            tra := tail(tra) ;
            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
                    pos   := 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 + res100.rot = 100 then 1 else 0 fi
                    pos   := if pos + res100(rot) < 0 then pos + res100(rot) + 100 else pos + res100(rot) fi
            fi ;
            ac_zp := ac_zp + zp ;
            ac_ze := ac_ze + ze ;
        od
post zp = noPassTra(posPro, traPro)  and-k ze = noEndTra(posPro, traPro)
```

Of course, to perform all these transformations, we should first prove the appropriate general rules and then apply them. This again requires some additional research.

After having developed our program, we should "implement" two functions, div100 and res100, that appear in it. We can either declare them as functional procedures or replace their "calls" with their definitions. We leave this obvious step to the readers.

Two observations may be worth considering at the end.

Of the 12 functions introduced in Sec. 3.2 and used in the development of our program, only five remained in the final metaprogram — div100, div100, noneZero, noPassTra, noEndTra — and only div100 and div100 must be implemented. The others need not, since they appear only in conditions. This observation justifies our claim expressed at the end of Sec 3.2, that the implementation of new functions should be postponed until the target program is ready.

The second observation concerns phantom constants posPro and traPro. Since they do not appear anywhere in the algorithmic part of our metaprogram — if they were, our program would not be correct — their declaration may be removed when the program is going to be executed. If the initial tra is a large file, this may save a substantial amount of memory, since declaring traPro practically doubles the program's memory usage. Of course, when we remove phantom constants, the postcondition is no longer satisfied, but we do not need to care about this fact anymore.

3.4 Some general observations

One crucial problem associated with program correctness is a lack of an adequate language for the specification of programs, i.e., a language that would be at the same time:

- **sufficiently clear** for the users (investors) of the program to express their expectations,
- **sufficiently formal** (unambiguous) for 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.

Potential future users of our program may be mechanical engineers designing, e.g., encrypted locks for strongboxes. We may assume, therefore, that they have a fairly good understanding of mathematics, but can we assume that they can write the recursive definitions of our target functions, noPassTra and noEndTra? And if not, then in which unambiguous way can we/they describe the fact that the pointer of our device has passed by or stopped at zero a certain number of times? Is there a more direct or more obvious way of describing such facts than our functions?

We may also note that the definitions of NoPassTra and NoEndTra can be regarded as recursive declarations of functional procedures and can therefore be used directly in program code. From this perspective, our task performed in Sec. 3 is an implementation of recursive procedures by an iterative program.

Independently, we may also ask if our recursive specifications are adequate? But, adequate in what sense? Adequate to what? If they are used to specify our task, then they are adequate by definition! To prove their correctness, or better, to convince us (even more?) of their adequacy, we should specify our task in another formal way, and then prove that the two specifications are equivalent. We are tackling this problem in Sec 4.

4 Iterative approach

4.1 Simplifying assumptions

To concentrate on the issue of how to describe the behavior of a physical device adequately, we will consider a simplified version of our task, assuming for simplicity that:

1. all rotations are positive, i.e., clockwise,
2. we count only passes by zero,
3. we restrict our problem to one rotation.

4.2 The specification of the contract

It seems fairly evident (we can't escape from such remarks) that the most direct way of describing the behavior of our pointer is to describe our device "literally" by the following metaprogram:

```
pre (zp, pos, rot is-v integer) and-k (0 ≤ pos ≤ 99) and-k (rot > 0) :          (Błąd!
Nie można odnaleźć źródła odwołania.-1)
    let phantom rotStart be integer = rot tel;
    zp := 0 ;
    while rot > 0
        do
        asr zp = noPass(rotStart – rot) rsa
            if pos = 0 then zp := zp+1 else skip fi ;
            pos := suc.pos ;
            rot := rot – 1 ;
        od
post zp = noPass(pos, rotStart)
```

This program may now be regarded as an axiom defining the function noPass. Our mechanical engineers will probably agree that this is what they mean by "counting zero-passes". But what is then our programming task, if the specification of this task is a program "ready to use"?

A typical task may be to speed up our program since, so far, it computes zp in linear time. Formally, our contract with the investor of the program may be the following:

Assuming that (**Błąd! Nie można odnaleźć źródła odwołania.**-1) is an axiomatic definition of noPass, derive a faster but still correct metaprogram of the form:

```
pre (zp, pos, rot is-v integer) and-k (0 ≤ pos ≤ 99) and-k (rot > 0) :          (Błąd!
Nie można odnaleźć źródła odwołania.-2)
    (some instruction)
post zp = noPass(pos, rot)
```

From the axiom, we may deduce by induction the following properties of our function :

```
noPass.(pos, 1)      = if pos = 0 then 1 else 0 fi
noPass.(pos, n+1)    = noPass.(pos, n) + if pos + n + 1 = 0 then 1 else 0 fi
```

Further lemmas that may be proved are the following:

noPass.(pos, 100 x n) = n for any 0 ≤ pos ≤ 99 **and-k** n ≥ 1
noPass.(pos, rot) = **if** pos + rot < 100 **then** 0 **else** 1 **fi** for any rot < 100

…to be continued.

5 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.

[3] H. Rasiowa, R. Sikorski, *The Mathematics of Metamathematics*, Państwowe Wydawnictwo Naukowe, Warszawa 1963