Lingua Project

(13) Metaprograms' development in Lingua (2)

(Sec. 9.4.6 and 9.5.1)

The book "**Denotational Engineering**" may be downloaded from:
https://moznainaczej.com.pl/what-has-been-done/the-book

Andrzej Jacek Blikle

May 10th, 2025

Rules for structured instructions

Lemma 9.4.6-1 Rule for conditional branching **if-then-else-fi**

pre (prc **and-k** vex**)** **:** sin-1 **post** poc
pre (prc **and-k (not-k** vex**))** **:** sin-2 **post** poc
prc ⇨ (vex **or-k (not-k** vex))

---

pre prc **if** vex **then** sin-1 **else** sin-2 **post** poc

Lemma 9.4.6-2 Rule for loop **while-do-od**

pre (inv **and-k** vex**) :** sin **post** inv
limited replicability of (asr vex **rsa ;** sin) **if** inv
prc ⇨ inv
inv ⇨ (vex **or-k (not-k** vex))
inv **and-k (not-k** vex)) ⇨ poc

---

pre prc **: while** vex **do** sin **od post** poc

we have to find an invariant condition inv

Rule for assignment instructions

Lemma 9.4.6-3 @-tautology

pre sin @ con
  sin
post con

Initial program where + and < are integer operations

pre x := y+1 **@** 2*x < 10
  x := y+1
post 2*x < 10

Conclusions (of the correctness of our program):
x, y – declared as integer variables
y+1, 2*x – evaluate cleanly

A proved lemma:
x := y+1 @ 2*x < 10 ⇔ (x **is integer**) **and-k** 2*(y+1) < 10

Constructed program

pre (x **is** integer) **and-k** 2*(y+1) < 10
  x := y+1
post 2*x < 10

replacement of a condition
by a weakly equivalent one
(no strong equivalence!)

A general programmer's step

In a general case, building a program in **Lingua** may be seen as a sequence of the following steps:

given
prc – a precondition
poc – a postcondition,
create
rei – a reinforcement of precondition
spr – a specprogram
such that
pre prc **and-k** rei : spr **post** poc – is correct

In a general case:

rei = und-rei **and-k** der-rei

und-rei – underivable condition; must be conjunctively added to the preconditions and postconditions of preceding programs,
der-rei – derivable condition; the preconditions and postconditions of preceding programs must be appropriately strengthened.

A procedural programmer's step (1)

Programmer's tasks for a procedure call:

given

prc-call – a precondition of the (future) call
poc-call – a postcondition of the (future) call

create

rei – a precondition reinforcement
proc myProc **(val** fpa-v **ref** fpa-r **begin** my-body **end** – a procedure decl.

such that

pre prc-call **and-k** prc-rei **:**
call MyClass.myProc **(val** apa-f **ref** apa-r **)** – is correct (final task)
post poc-call

the **main challenge** is to build a correct metaprogram
pre prc-body **:** my-body **post** poc-body
appropriately related to the call

construction rule

A procedural programmer's step (2)

Lemma 9.4.6-4 Rule for a call of an imperative procedure

prc-call ⇨ myProc (**val** fpa-v **ref** fpa-r) my-body **imperative in** MyClass
prc-call ⇨ **(pass actual val** apa-v **ref** apa-r **to formal val** fpa-v **ref** fpa-r **with** MyClass**) @** prc-body
prc-call ⇨ **procedure** MyClass**.**myProc **is opened**
prc-call ⇨ coe **is current**
prc-body ⇨ my-body **@** poc-body i.e. **pre** prc-body **:** my-body **post** poc-body
poc-body ⇨ fpa-r **well-valued in** coe
poc-body[fpa-r/apa-r] ⇨ poc-call

---

pre prc-call **:**
call MyClass.myProc (**val** apa-v **ref** apa-r)
post poc-call

An example of a program development

non-negative integer

```
pre n, q, y, p is nnint:
   q :=1;
   while q ≤ n do q:=4*q od
   y:= n;
   p:= 0;
   while q > 1
      do
         q:=q/4;
         if p+q ≤ y
            then p:=p+q; y:=y-p-q
            else p:=p/2
         fi
      od
post p=isrt(n)
```

Ole-Johan Dahl
1931 – 2002
Turing Award winner 2001

integer square root

Def: $\mathrm{isrt}(n)^2 \leq n < (\mathrm{isrt}(n)+1)^2$

Building a searching engine

First goal

pre x = 0 **and-k** k **is** nnint :
  while x+1 ≤ k
    do x := x+1 **od**
post x = k

Rule to be applied

1) **pre (**inv **and-k** vex**) :** sin **post** inv
2) **limited replicability of (asr** vex **rsa ;** sin**) if** inv
3) prc ⇨ inv
4) inv ⇨ (vex **or-k (not-k** vex**))**
5) inv **and-k (not-k** vex**))** ⇨ poc

---

pre prc **: while** vex **do** sin **od post** poc

The application of the rule:

inv: 0 ≤ x ≤ k

1) **pre** 0 ≤ x ≤ k **and-k** x+1 ≤ k **:** x := x+1 **post** 0 ≤ x ≤ k
2) **limited replicability of (asr** x+1 ≤ k **rsa ;** x := x+1 **) if** 0 ≤ x ≤ k
3) x = 0 **and-k** k **is** nnint ⇨ 0 ≤ x ≤ k
4) 0 ≤ x ≤ k ⇨ x+1 ≤ k **or-k (not** x+1 ≤ k)
5) 0 ≤ x ≤ k **and-k (not** x+1 ≤ k) ⇨ x = k

By sequential composition

pre k **is** nnint
  x := 0 **;**
  while x+1 ≤ k
    do x := x+1 **od**
post x = k

Installing an appliance on the engine

Linear search engine

pre k **is** nnint :
  x := 0;
  while x+1 ≤ k
    do x := x+1 **od**
post x = k

installing
an appliance

Applied rule: substitution

pre x,n **is** nnint :
  x := 0;
  while x+1 ≤ isrt(n)
    do x := x+1 **od**
post x = isrt(n)

x+1 ≤ isrt(n) ≡ $(x+1)^2$ ≤ n **whenever** x,n **is** nnint

A liner-time program

pre x,n **is** nnint :
  x := 0;
  asr x,n **is** nnint
  while $(x+1)^2$ ≤ n
    do x := x+1 **od**
  rsa
post x = isrt(n)

If we wish to speed up our program, we have to change the engine

Step 1- building a logarithmic search engine

The magnitude of k: If $2^m \le k < 2^{m+1}$ then mag.k = 2^m e.g. mag.11 = 8

Def: po2.k iff $(\exists m \ge 0)\ k=2^m$:   k is a power of 2

Q1: **pre** x, k, z **is** nnint :   searches for 2*mag.k   e.g. 2*mag.11 = 16

```
Q1: pre x, k, z is nnint :
      z := 1;
      asr x, k, z is nnint and-k po2.z  :
         while z ≤ k do z:=2*z od
      rsa
   post x, k, z is nnint and-k z = 2*mag.k
```

k = 11

2*mag.11 = 16

11 = 1*8 + 0*4 + 1*2 + 1*1

```
Q2: pre  x, k, z is nnint and-k z = 2*mag.k  :
      x := 0;
      while z > 1
         do
            z := z/2;
            if x + z ≤ k then x:=x + z fi
         od
   post x = k and-k z = 1
```

Next step:
combine these programs sequentially

Step 2 - combining programs sequentially

```
Q3: pre x,k,z is nnint :       a logarithmic search engine
      z := 1;
      x := 0;                                  - assignment moved up
       asr x, k, z is nnint and-k po2.z :   - the range of assertion extended down
          while z ≤ k do z:=2*z od
          while z > 1
            do
              z := z/2;
              if x + z ≤ k then x:=x + z fi
            od
        rsa
      post x = k and-k z = 1
```

Next step:

replace k by isrt(n) and use

$z \leq isrt(n) \equiv z^2 \leq n$ **whenever** z, n **is** nnint

$x + z \leq isrt(n) \equiv (x + z)^2 \leq n$ **whenever** z, n, x **is** nnint

Step 3 – substitution and replacement

Q4:
```
pre z,x,n is nnint:
   z := 1;
   x := 0
   asr z,x,n is nnint and-k po2.z :
      while z² ≤ n do z:=2*z od
      while z > 1
         do
            z := z/2;
            if (x+z)² ≤ n then x:=x+z fi
         od
   rsa
post x = isrt(n) and-k z = 1
```

We time-optimize this program by restricting the number of executions of arithmetic operations (time).

Next step:
To avoid the recalculation of z^2 introduce a register variable
(1) Introduce new variable q with $q=z^2$
(2) Introduce updates of q to keep $q = z^2$.

Step 4 – introducing of a register variable q

```
Q5: pre z, x, n, q is nnint:
      z  :=1;
      x  :=0;
      q  :=1;
      asr z,x,n is nnint and-k po2.z and-k q=z²
        while q ≤ n do off z:=2*z; q:=4*q on od
        while z > 1
          do
            off z:=z/2; q:=q/4 on
            if x²+2*x*z+q ≤ n then x:=x+z fi
          od
      rsa
    post x=isrt(n) and-k z=1 and-k q=z²
```

register variable

register expression

register condition

assertion
switched off locally

Next step:

$z>1 \equiv q>1$ **whenever** $(z>0$ **and-k** $q=z^2)$

new variables y and p with $y=n-x^2$ and $p=x*z$

$x^2 + 2*x*z + q \le n \equiv 2*p+q \le y$ **whenever** $(y=n-x^2$ **and-k** $p=x*z)$

The introduction of y and p is an invention to be justified later.

Step 5 – introducing register variables y, p

Q6: **pre** z, x, n, q, y, p **is** nnint:

```
    z := 1; x := 0; q := 1;
    asr z, x, n is nnint and-k q=z^2 :
      while q ≤ n do off z:=2*z; q:=4*q on od
      y:= n;
      p:= 0;
      asr y=n-x^2 and-k p=x*z :
        while q > 1
          do
            off z:=z/2; q:=q/4; p:=p/2; on
            if 2*p+q ≤ y  then x:=x+z; p:=p+q; y:=y-2p-q fi
          od
      rsa
    rsa
```

post x = isrt(n) **and-k** z = 1 **and-k** $q=z^2$ **and-k** p=x*z **and-k** $y=n-x^2$

$q=z^2$ ⇔ isrt(q)=z **whenever** z **is** nnint

We replace z by isrt(q) in order to eliminate z in the next step.

Step 6 – introducing isrt(q)

Q7: **pre** z, x, n, q, y, p **is** nnint:
  z := 1; x := 0; q := 1;
  asr z, x, n **is** nnint **and-k** isrt(q)=z :
    while q ≤ n **do off** z:=2*isrt(q); q:=4*q **on od**
    y:= n;
    p:= 0;
    asr y=n-x^2 **and-k** p=x*isrt(q) **:**
      while q > 1
        do
          off z:=isrt(q)/2; q:=q/4; p:=p/2 **on**
          if 2*p+q ≤ y
            then x:=x+isrt(q); p:=p+q; y:=y-2p-q
          fi
        od
    rsa
  rsa
post x=isrt(n) **and-k** z=1 **and-k** q=1 **and-k** p=x **and-k** y=n-x^2

z can be removed because it doesn't contribute to other variables and we do not need its terminal value.

since z=1

Step 7 – eliminating z

Q8: **pre** x, n, q, y, p **is** nnint:
x := 0; q := 1;
asr x,n **is** nnint:
while q ≤ n **do** q:=4*q **od**
y:= n;
p:= 0;
asr y=n-x^2 **and-k** p=x*isrt(q) **:**
while q > 1
do
off q:=q/4; p:=p/2 **on**
if 2*p + q ≤ y
then p:=p+q; y:=y-2p-q
fi
od
rsa
rsa
post x=isrt(n) **and-k** q=1 **and-k** p=x **and-k** y=n-x^2

after this transformation x becomes unnecessary

x=isrt(n) ≡ p=isrt(n) **whenever** p=x

we also remove assertions which we will not need

Step 8 – eliminating x

Q9:

```
pre n, q, y, p is nnint:
   q := 1;
   while q ≤ n do q:=4*q od
   y:= n;
   p:= 0;
   while q > 1
      do
         q:=q/4;
         p:=p/2; if 2*p+q ≤ y   then p:=p+q; y:=y-2p-q fi
      od
post p=isrt(n) and-k q=1
```

replace by

if p+q ≤ y **then** p:=p/2+q; y:=y-p-q **else** p:=p/2 **fi**

A

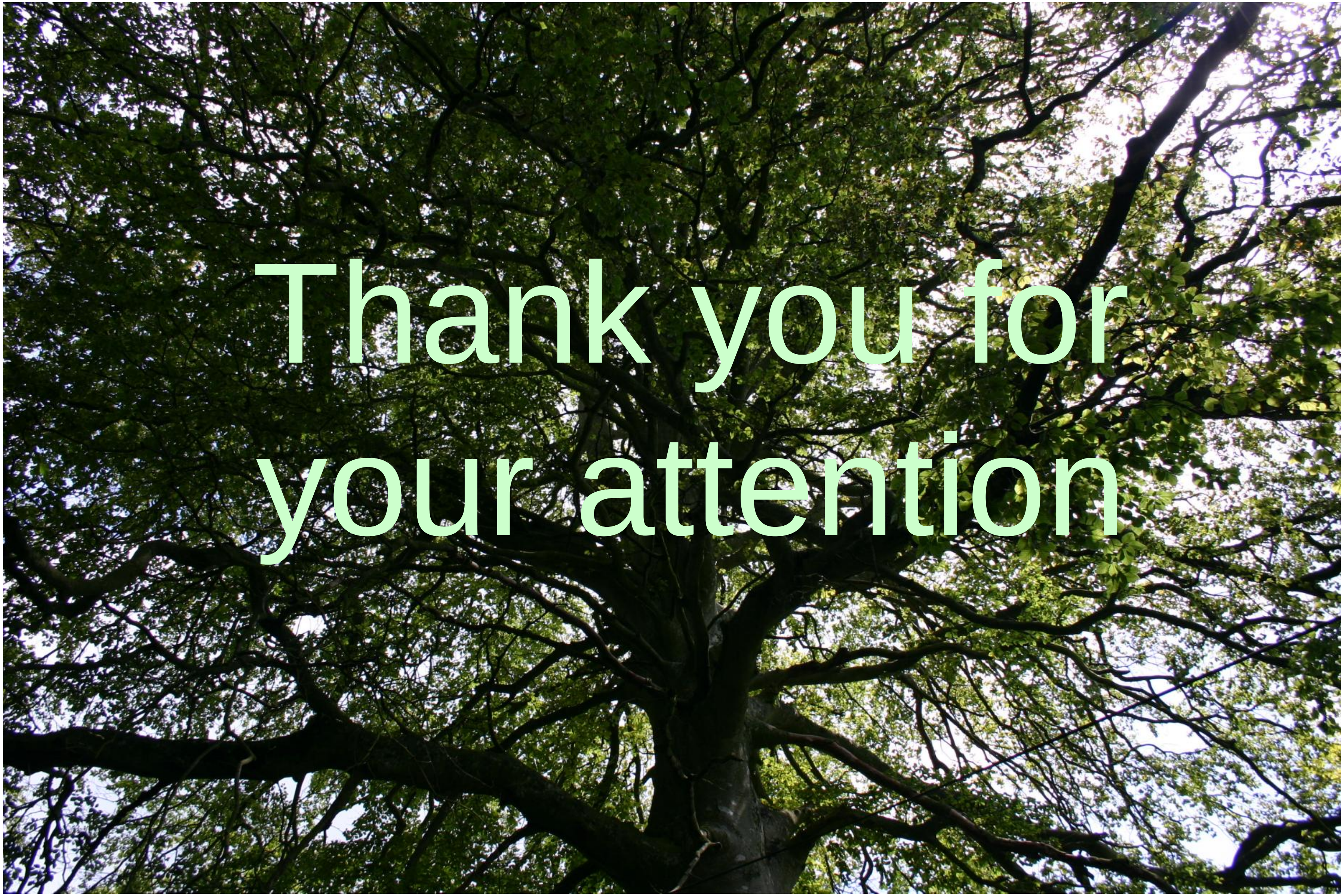
Step 9 – the Dahl's program

Q10:

```
pre n, q, y, p is nnint:
   q :=1;
   while q ≤ n do q:=4*q od
   y:= n;
   p:= 0;
   while q  > 1
      do
         q:=q/4;
         if p+q ≤ y
            then p:=p+q; y:=y-p-q
            else p:=p/2
         fi
      od
post p=isrt(n)
```

All arithmetic operations are easily implementable in binary arithmetic.

A


Thank you for your attention