A Denotational Engineering of Programming Languages

…

Part 11: Lingua-2V Transformational programming

(Section 8.6 of the book)

Andrzej Jacek Blikle

June 9th, 2021

Enriching the functionality of programs

Installing an appliance on an engine

Step 1: A trivial search engine (linear search)

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

installing an applience

Def: $isrt(n)^2 \le n < (isrt(n)+1)^2$

Step 2: A trivial program

```
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)` ≡ `$(x+1)^2 \le n$ whenever x,n is nnint`

Step 2: a slow program

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

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

The derivation of Dahl's integer square root (1)

(deriving a logarithmic search engine)

rząd wielkości

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

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

Q1: searches for 2*mag.k e.g. 2*mag.11 = 16

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

combine these programs sequentially

Q2:

```
pre x,k,z is nnint and 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 z = 1
```

k = 11

2*mag.11 = 16

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

The derivation of Dahl's integer square root (2)

(with a logarithmic search engine)

```
Q3: pre x,k,z is nnint :     a "pure" search engine
      z := 1;
      x := 0;
       asr x,k,z is nnint and po2.z :
         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 z = 1
```

Replace `k` by `isrt(n)` and use

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

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

A

The derivation of Dahl's integer square root (3)

(with a logarithmic search engine)

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

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

First introduce new variable q with $q=z^2$ to avoid the recalculation of z^2

The derivation of Dahl's integer square root (4)

(with a logarithmic search engine)

```
Q5: pre z,x,n,q is nnint:
      z := 1;
      x := 0;
      q := 1;
      asr z,x,n is nnint and po2.z and $q=z^2$
        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 $x^2+2*x*z+q ≤ n$ then x:=x+z fi
          od
      rsa
    post x=isrt(n) and z=1 and $q=z^2$
```

register identifier

identyfikator rejestrowy

register expression

wyrażenie rejestrowe

register condition

warunek rejestrowy

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

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

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

A

The derivation of Dahl's integer square root (5)

(with a logarithmic search engine)

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

```
Q6: pre z,x,n,q,y,p is nnint:
      z := 1; x := 0; q := 1;
      asr z,x,n is nnint and $q=z^2$ :
        while $q \le n$ do off z:=2*z; q:=4*q on od
        y:= n;
        p:= 0;
        asr $y=n-x^2$ and p=x*z :
          while q > 1
            do
              off z:=z/2; q:=q/4; p:=p/2; on
              if $2*p+q \le y$ then x:=x+z; p:=p+q; y:=y-2p-q fi
            od
        rsa
      rsa
    post x = isrt(n) and z = 1 and $q=z^2$ and p=x*z and $y=n-x^2$
```

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

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

The derivation of Dahl's integer square root (6)

(with a logarithmic search engine)

```
Q7: pre z,x,n,q,y,p is nnint:
      z := 1; x := 0; q := 1;
      asr z,x,n is nnint and 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 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 z=1 and q=1 and p=x and 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`

The derivation of Dahl's integer square root (9)

(with a logarithmic search engine)

```
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 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 q=1 and p=x and $y=n-x^2$
```

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

after this transformation x becomes unnecessary

we also remove assertions which we will not need

The derivation of Dahl's integer square root (10)

(with a logarithmic search engine)

```
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 q=1
```

replace by

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

The derivation of Dahl's integer square root (11)

(with a logarithmic search engine)

```
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 easilly implementable in binary arithmetic.

This is the Ole Dahl's program.

Did he developed it in a similar way?

Adding a register identifier

The idea of a method of register indentifiers

Inserting `ide-r` with `ide-r=dae-r` into P

```
P: pre prc
     ins-h;    (head)
     asr con rsa ;
     asr con:
       ins
     rsa
     ins-t     (tail)
   post poc
```

```
Q: pre prc
     ins-h ;
     ide-r := dae-r ;
     asr con and ide-r=dae-r :
       $(ins, ide-r=dae-r)
     rsa
     ins-t
   post poc
```

syntactic metaoperator making Q correct

Assumptions:

`ide-r` is not in P

`pre con: ide-r:=dae-r post TT`

`ide-r:=dae-r` will be executed without error or looping

`ide-r=dae-r` — register condition
`ide-r` — register identifier
`dae-r` — register expression

An inductive definition of $

Let ins be ide:=dae   Let Sde.[sin @ dae] = Ssi.[sin] ● Sde.[dae]

arbitrary
data expression

If ide not in dae-r then

$(ide:=dae, ide-r=dae-r) = ide:=dae

If ide is in dae-r then

$(ide:=ade, ide-r=dae-r) =

= the equality of syntactic objects

(1) = **off** ide:=ade; ide-r:=dae-r **on**

(2) (reverse order) = **off** ide-r:=(ide:=dae)@dae-r; ide:=ade **on**

E.g. transformation from (1) into (2) in the context of **asr** $q=z^2$ **rsa**

asr $q=z^2$ **rsa;** **off** z:=2*z ; $q:=z^2$ **on** ≡

the elimination of z from $q:=z^2$

asr $q=z^2$ **rsa;** **off** $q:=(z:=2*z)@z^2$; z:=2*z **on** ≡

asr $q=z^2$ **rsa;** **off** $q:=4z^2$; z:=2*z **on** ≡

asr $q=z^2$ **rsa;** **off** q:=4q; z:=2*z **on**

≡ the equality of denotations

An inductive definition of $ (con.)
Imperative-procedure call

If none of actual reference parameters is in `dae-r` then

```
$(call ide (ref apa-r val apa-v), ide-r=dae-r) =
      = call ide (ref apa-r val apa-v)
```

If thre is a reference parameter in `dae-r` then

```
$(call ide (ref apa-r val apa-v), ide-r=dae-r) =
      = off call ide (ref apa-r val apa-v); ide-r:=dae-r on
```

Structured instructions

```
 $((ide-1 ; ide-2), ide-r=dae-r) =
  $(ide-1, ide-r=dae-r) ; $(ide-2, ide-r=dae-r)


$(if dae-b then ins-1 else ins-2 fi, ide-r=dae-r) =
if dae-b
   then $(ins-1, ide-r=dae-r)
   else $(ins-2, ide-r=dae-r)
fi
```

For **`while`** analogously.

$ versus `@`

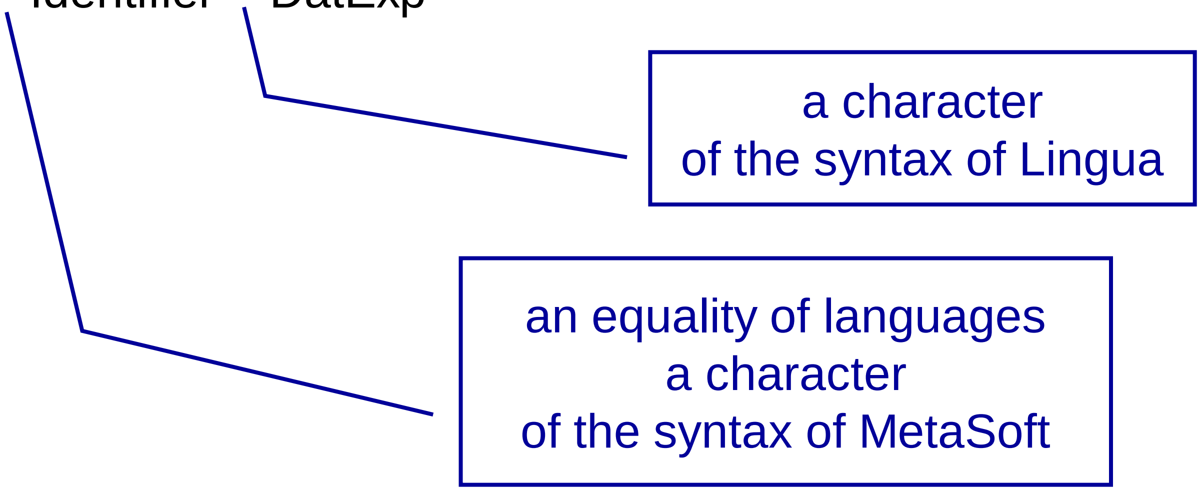
`@` is a symbol from the syntax of Lingua

Sde.[`ins @ dae`] = Sin.[`ins`] ● Sde.[`dae`]

$ a syntactic constructor

$: Instruction x RegisterCondition $\mapsto$ Instruction where

RegisterCondition = Identifier `=` DatExp



a character
of the syntax of Lingua

an equality of languages
a character
of the syntax of MetaSoft

Invariants versus assertions

A strong invariant (in proofs of total correctness)

`con ⇨ ins @ con` i.e.

$\{\texttt{con}\} \subseteq \mathrm{Sin}.[\texttt{ins}] \bullet \{\texttt{con}\}$

A weak invaiant (in proofs of partial correctness)

$\{\texttt{con}\} \bullet \mathrm{Sin}.[\texttt{ins}] \subseteq \{\texttt{con}\}$

A loop invariant (in proofs of total correctness of **`while`**)

```
there exists a condition inv such that:
pre inv and dae: sin post inv
pre inv while dae do sin od post TT
prc ⇨ inv
inv and (not dae) ⇨ poc
────────────────────────────────────
pre prc:
  while dae do sin od
post poc
```

To be an invariant is a property of condition relativized to an instruction.

Assertions

`asr` `con` **`rsa`**

are specinstructions.

Thank you for your attention