A Denotational Engineering of Programming Languages

...

Part 10: Lingua-2V Program-construction rules

(Section 8.5 of the book)

Andrzej Jacek Blikle

April 7th, 2020

The role of declarations

The derivation of a correct metaprogram

```
pre prc: dec; sin post poc
```

can be split into the derivation of two metaprograms:

```
pre prc:
  dec; skip-i
post prc and poc-dec
```

```
pre prc and poc-dec:
  skip-d; sin
post poc
```

prc can't include identifiers declared in dec

poc-dec conjunction of declaration-oriented conditions:

```
ide is tex
ide is-type tex
ide proc-with ipd
ide fun-with fpd
```

In the majority of program-construction rules we don't need to include declarations

A

Declaration-oriented conditions implicite in data-oriented conditions

```
pre prc:
  dec; skip-i
post poc-dec and prc
```

`x is number and x > 0` $\Leftrightarrow$ `x > 0`

because

`x > 0` $\Longrightarrow$ `x is number`

but ≡ does not hold, e.g. if `x is word`

`x is number and x > 0` may be replaced by `x > 0`

in:

- pre- and post conditions,
- assertions.

A

The case of structured instructions

Rules concerning pre- and post conditions

Rule 8.5.2-1 Strengthening precondition

```
pre prc : sin post poc
prc-1 ⇨ prc
──────────────────────────
pre prc-1 : sin post poc
```

Rule 8.5.2-2 Weakening postcondition

```
pre prc : sin post poc
poc ⇨ poc-1
──────────────────────────
pre prc : sin post poc-1
```

Assignment

Rule 8.5.2-1 Assignment

```
pre ide:=dae @ con:
  ide := dae
post con
```

Proof follows directly from the semantics of algorithmic conditions.

Example

```
pre ide:=y+1 @ 2*x > 10:
   x := y+1
post 2*x > 10
```

```
pre 2*(y+1) > 10:
   x:= y+1
post 2*x > 10
```

```
x:=y+1 @ 2*x>10 ⇔ 2*(y+1)>10
```

Here we have ≡
but we only need ⇔

A

Sequential composition

Rule 8.5.2-6 Sequential composition of a metaprogram with an instruction

(1) **pre** prc-1: dec; sin-1 **post** poc-1
(2) **pre** prc-2: sin-2 **post** poc-2
(3) poc-1 ⇨ prc-2

---

(4) **pre** prc-1: dec; sin-1;sin-2 **post** poc-2
(5) **pre** prc-1: dec; sin-1; **asr** poc-1 **rsa**; sin-2 **post** poc-2
(6) **pre** prc-1: dec; sin-1; **asr** prc-2 **rsa**; sin-2 **post** poc-2

Implication only top-down!

(1), (2) – constructions of programs
(3) – "usual" mathematical proof (by proof checker)

- mathematically not very sophisticated
- but may include many variables

A

Conditional branching

Rule 8.5.2-2 Conditional branching if-then-else-fi

```
pre (prc and dae)     : sin-1 post poc
pre (prc and not dae) : sin-2 post poc
prc ⇨ dae or(not dae)
─────────────────────────────────────────
pre prc:
  if dae then sin-1 else sin-2 fi
post poc
```

satisfaction of `prc` guarantees the definedness of `dae`

Absent in Hoare's logic

This program aborts if:

- x = 0,
- x is not a number,
- x has not been declared.

In Hoare's logic we can prove:

```
pre true:
  if 1/x > 0 then x:=x else x:=-x fi
post x > 0
```

A

While loop

Rule 8.5.2-8 Loop while-do-od

absent in Hoare's logic

clean total correctness of `sin`

```
prc ⇨ inv
inv ⇨ (dae or (not dae))
inv and (not dae) ⇨ poc
pre inv and dae: sin post inv
asr dae rsa ; sin fi limited-replicability in inv
```

```
pre prc:
  while dae do sin od
post poc
```

The application of this rule requires:

1. proving three metaimplications,
2. constructing a correct metaprogram; inventing an invariant
3. proving halting property; inventing a well-founded set and a corresponding function

A

An example of a while-program derivation

```
pre m,n≥0 and x=n and k=1:      — precondition prc
  while x≠0                     — data expression dae
  do
    k := k*m ;                  — the beginning of sin
    x := x-1 ;                  — the end of sin
  od ;
  post k=m^n                    — postcondition poc
```

The values of `n` and `m` remain constant.

Let `inv` be: `k=m^(n-x)`

(1) **pre** `k=m^(n-x)` **and** `x≠0: k:=k*m; x:=x-1` **post** `k=m^(n-x)`
(2) **asr** `x≠0` **rsa**`; k:=k*m; x:=x-1` **limited-replicability in** `k=m^(n-x)`
(3) `n,m≥0` **and** `x=n` **and** `k=1` ⇨ `k=m^(n-x)`
(4) `k=m^(n-x)` ⇨ `x=0` **or** `x≠0`
(5) `k=m^(n-x) and x=0` ⇨ `k=m^n`

well-founded set:
(non-negative integers, >)
K.sta = Sde.[x].sta

To derive our program we have to derive or prove, (1), and prove (2) - (5).

A

The case of imperative procedures

Procedures

Non-procedural case:

build a program
with expected properties

Given conditions (expectations):

`prc, poc`

Build a correct program (instruction):

pre prc: $\boxed{\texttt{ins}}$ **post** poc

programming task

Procedural case:

build a <u>declaration</u> of a procedure
such that
the <u>call</u> of that procedure has expected properties

Given a procedure call (expectations):

```
pre prc-call :
  call DoIt(val fop-v ref fop-r)
post poc-call
```

Build a procedure declaration (body):

```
proc DoIt(val fop-v ref fop-r)
  body
end proc
```

```
pre prc-body:
  body
post poc-body
```

programming task

A

A step-by-step construction

Given a program:

```
pre prc-call :
  call DoIt(val acp-v ref acp-r)
post poc-call
```

expectation

Build a declaration:

```
proc DoIt(val fop-v ref fop-r)
  body
end proc
```

where:

```
pre prc-body:
  body
post poc-body
```

programming task

What should we assume about the future programming context of the call to make the call executable?

```
prc-call ⇨ DoIt proc-with ipd
prc-call ⇨ conformant(fop-v, fop-r, acp-v, acp-r)
```

call-time state `prc-call`

A

A step-by-step construction (cont.)

What should we assume about the properties of body

```
pre prc-body:
 body
post poc-body
```

to make the call correct?

```
prc-call ⇨ prc-body[fop-v/acp-v, fop-r/acp-r]
poc-body ⇨ poc-call[acp-r/fop-r]
```

A

Imperative procedures

As a conclusion of our analysis we have the following rule:

Rule 8.5.3-1 Building a declaration of an imperative procedure

```
(1) pre prc-bod: body post poc-bod
(2) prc-call ⇨ DoIt proc-with ipd
(3) prc-call ⇨ conformant(fop-v, fop-r, acp-v, acp-r)
(4) prc-call ⇨ prc-bod[fop-v/acp-v, fop-r/acp-r]
(5) poc-bod  ⇨ poc-call[acp-r/fop-r]
─────────────────────────────────────────────────────
(6)pre prc-call
    call DoIt (val acp-v ref acp-r)
   post poc-call
```

If `DoIt` is a recursive procedure then we cannot prove (1) independently of (6)

A

Example of body derivation

GOAL: Derive a declaration of procedure `Power` such that:

```
pre Power proc-with ipd and a,b,c is nnint :
  call Power(val a,b ref c)
post c=a^b.
```

since `a,b` are value parameters, their values are not changed by the call

STARTING POINT:

```
pre m,n≥0 and x=n and k=1:
  while x≠0 do k:=k*m; x:=x-1 od
post k=m^n
```

a predefined yokeless type of non-negative integers

EXPECTED HEADER OF PROCEDURE:

```
Power(val m,n nnint ref k nnint)
```

EXPECTED BODY PRECONDITION:

```
m,n,k is nnint
```

A

Step-by-step construction

```
pre m,n,k is nnint :
  let x be nnint tel;
  x:=n; k:=1;
post m,n≥0 and x=n and k=1
```

```
pre m,n≥0 and x=n and k=1:
  while x≠0 do k:=k*m; x:=x-1 od
post k=m^n
```

assumption (1)
is satisfied

```
pre m,n,k is nnint :
  let x be nnint tel;
  x:=n; k:=1;
  while x≠0 do k:=k*m; x:=x-1 od
post k=m^n
```

check satisfaction
of (2) – (5)

```
proc Power(val m,n nnint ref k nnint)
  let x be nnint tel;
  x:=n; k:=1;
  while x≠0 do k:=k*m; x:=x-1 od
endproc
```

```
pre Power proc-with ipd and a,b,c is nnint :
  call Power(val a,b ref c)
post c=a^b.
```

A

The case of recursion

Recursion – a new pattern of validation

NO RECURSION:

given (hypothesis) implies prove (conclusion)

```
pre prc-bod:                 pre prc-call
 body              implies    call DoIt (val acp-v ref acp-r)
post poc-bod                 post poc-call
```

RECURSION:

Prove that both are correct!

```
pre prc-bod:                 pre prc-call
 body                and      call DoIt (val acp-v ref acp-r)
post poc-bod                 post poc-call
```

In the case of recursion we can't avoid a correctness proof!

A

Simple nondetermnistic recursion

(a repetition)

If T is the least solution of $X = HXT \mid E$ then for any $A, B \subseteq S$

Rule 7.6.2-3

there exists a family of preconditions $\{A_i \mid i \geq 0\}$
and a family of postconditions $\{B_i \mid i \geq 0\}$ such that

$$
\frac{
\begin{array}{lll}
(\forall i \geq 0)\ A_i & \subseteq & (H^i E T^i) B_i \\
A & \subseteq & \mathrm{U}\{A_i \mid i \geq 0\} \\
(\forall i \geq 0)\ B_i & \subseteq & B
\end{array}
}{
\begin{array}{lll}
A & \subseteq & RB
\end{array}
}
$$

— i recursive calls

An example of a correctness proof for simple recursion

Goal: construct a procedure declaration of `RecPow` to make this call correct

```
pre RecPow proc-with ipd and a,b,c is nnint :
  call RecPow(val a,b ref c)
post c=a^b
```

A mathematical task: prove the correctness of the call.

A candidate for declaration (`ipd`):

```
proc RecPow(val m,n nnint ref k nnint)
  let x be number tel;
  x:=n; k:=1;
  if x≠0
    then x:=x-1 ; call RecPow(val m,x ref k); k:=k*m
    else skip-i
  fi
end-proc
```

A

An example (cont.)

An inductive version of the hypothesis; induction on N (a concrete number).

```
pre RecPow proc-with ipd and a,b,c is nnint and b=N:
  call RecPow(val a,b ref c)
post c=a^N
```

First step: N = 0 and formal parameters replaced by actual parameters

```
pre RecPow proc-with ipd and a,b,c is nnint and b=0:
   let x be nnint tel;
   x:=0; c:=1;
   if x≠0
      then x:=x-1 ; call RecPow(val a,x ref c); c:=c*a
      else skip-i
   fi
post RecPow proc-with ipd and a,b,c is nnint and b=0 and c=a^b
```

Equivalent to:

```
pre RecPow proc-with ipd and a,b,c is nnint and b=0:
   let x be nnint tel;
   x:=0; c:=1
post RecPow proc-with ipd and a,b,c is nnint and b=0 and c=1
```

A

An example (cont.)

Inductive step: let b = N+1 for N ≥ 0

```
pre RecPow proc-with ipd and a,b,c is nnint and b=N+1:
  let x be nnint tel;
  x:=N+1; c:=1;
  if x≠0
    then x:=x-1 ;
    asr RecPow proc-with ipd and a,b,c is nnint and x=N rsa;
     call RecPow(val a,N ref c);
    asr RecPow proc-with ipd and a,b,c is nnint and b=N and c=a^N rsa;
    c:=c*a;
    else skip-i
  fi
post c=a^(N+1)
```

inductive hypothesis

A research problem:

Formalize and prove correctness rules for recursive procedures.

A

The case of functional procedures

Functional procedures

An example

```
fun RecPowerFun(m,n)
  let k is nnint tel
  call RecPower(val m,n ref k)
  return 3*k+1
endfun
```

Two forms of correctness statements:

```
pre RecPowerFun fun-with fpd and a,b is nnint:
   RecPowerFun(a,b)
post-exp 3*(a^b)+1
```

exported value as a function of actual parameters

```
pre RecPowerFun fun-with fpd and a,b is nnint:
   RecPowerFun(a,b)
post-yoke value > 1
```

property of exported value described by a yoke

A

Functional procedures

Formalization for arbitrary expressions

Generalization because fp call is an expression.

Properties of expressions described by expressions:

```
pre con
  dae                  means con ⇨ exp=p-exp
post-exp p-dae
```

The evaluations of both expressions terminate cleanly

Clean termination of a data expression `dae` under condition `con`:

`con ⇨ exp=exp`

Properties of expressions described by yokes:

```
pre con
  dae                  means con ⇨ exp □ yok
post-yoke yok
```

A

Functional procedures

New operator of conditions

Composite of the value of `exp` satisfies `yok`.

```
[exp □ yok].sta =
  is-error.sta        ➔ error.sta
  Sde.[exp].sta = ?   ➔ ?
  let
    val = Sde.[exp].sta
  val : Error         ➔ val
  let
    (com, yok-v)= val
    y-val       = Syoe.[yok].com
  true                ➔ y-val
```

Invariants versus assertions

Invariant of an instruction (condition):

{con} ● Sin.[ins] ⊆ {con}   partial invariant

{con} ⊆ Sin.[ins] ● {con}   total invariant

Invariant of a while-loop (condition):

```
prc ⇨ inv
inv ⇨ (dae or (not dae))
inv and (not dae) ⇨ poc
pre inv and dae: sin post inv
if dae then sin fi limited-replicability in inv
```

```
pre prc:
 while dae do sin od
post poc
```

Assertion (instruction):

```
asr con rsa
```


Thank you for
your attention