Lingua Project

(10) Program correctness in Lingua

(Sec. 9.1 and 9.2)

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

Andrzej Jacek Blikle

April 5th, 2025

Propositional calculus of Mc'Carthy
(non-transparent operations)

if x ≠ 0 **and** 1/x < 10 **then** x := x+1 **else** x := x–1 **fi**

If **and** is transparent, then our program aborts for x = 0.

The solution of John McCarthy:

ff **and-m** ee = ff — lazy evaluation from left to right

error or undefinedness

or-m	**tt**	**ff**	**ee**
tt	tt	tt	tt
ff	tt	ff	ee
ee	ee	ee	ee

and-m	**tt**	**ff**	**ee**
tt	tt	ff	ee
ff	ff	ff	ff
ee	ee	ee	ee

not-m	
tt	ff
ff	tt
ee	ee

Propositional calculus of Mc'Carthy

(some properties)

and-m, or-m — associative (if only one ee)

p **and-m** q ≠ q **and-m** p — not commutative

p **or-m** (**not** p) ≠ ff — never false

and-m is distributive over **or-m** only on the right-hand side, i.e.

p **and-m** (q **or-m** s) = (p **and-m** q) **or-m** (p **and-m** s)

Propositional calculus of Kleene

Even „more lazy” than McCarthy’s calculus

or-k	tt	ff	ee
tt	tt	tt	tt
ff	tt	ff	ee
ee	tt	ee	ee

and-k	tt	ff	ee
tt	tt	ff	ee
ff	ff	ff	ff
ee	ee	ff	ee

not-k	
tt	ff
ff	tt
ee	ee

Now commutativity

p **or-k** q = q **or-k** p

p **and-k** q = q **and-k** p

hence in particular

tt **or-k** ee = ee **or-k** tt = tt

ff **and-k** ee = ee **and-k** ff = ff

If ee may be an infinite computation, then Kleene's calculus requires a simultaneous evaluation of arguments.

Syntactic categories of Lingua-V
(V stands for „validating”)

Lingua-V includes all categories of Lingua (in colloquial version) plus five new categories:

1. conditions – representing partial 3-valued (Kleene's) partial predicates.
2. assertions – instructions aborting programs when a condition is not satisfied
3. specified programs – programs with nested assertions
4. metaconditions – describing relationships between conditions
5. metaprograms – specified programs with pre- and post-conditions

An example of a metaprogram:

```
pre x,k is integer and-k k > 0:          - a precondition
    x := 0;
    asr x = 0 rsa;                       - an assertion
    while x+1 ≤ k do x := x+1 od
post x = k                               - a postcondition
```

Conditions

Some specific notation

cod : ConDen = WfState → BooValE – the denotations of conditions
val : BooVal = {tv, fv} | Error

tv = (tt, 'boolean')
fv = (ff, 'boolean')

con : Condition – the syntactic domain of conditions
[ ] : Condition ⟼ ConDen – the semantics of conditions

[con] : WfState → BooValE – the denotation of con (transparent for errors)
{con} = {sta | [con].sta = tv} – the truth domain of con

NT : Condition – a special condition called nearly true
[**NT**].sta =
  is-error.sta ➔ error.sta – transparency for errors
  true ➔ tv

con is error-sensitive if it has one of two following properties:

  if is-error.sta then [con].sta = error.sta con is error-transparent
  if is-error.sta then [con].sta = fv con is error-negative

Value-oriented conditions

Value-oriented conditions include:

1. all value expressions with boolean values but with Kleene's operators,
2. some specific conditions, e.g.:

   vex-1 = vex-2 – (in Lingua, we do not allow for the comparison of arbitrary values)

   increasingly ordered (ide) – where ide points to an array

   vex **is value** – vex evaluates cleanly

   …

Basic value-, type-, reference-oriented conditions
(constructors of conditions)

(1) att at-ide **is** tex **with** yex **in** cl-ide **as** pst	
	– at-ide is declared with tex and yex in class cl-ide…
(2) ty-ide **is type in** cl-ide,	– ty-ide is declared as type constant in class cl-ide
(3) **var** ide **is** tex **with** yex,	– ide is a declared variable of type tex and yoke yok
(4) rex **is reference**,	– reference expression rex evaluates cleanly
(5) vex **is value**,	– value expression vex evaluates cleanly
(6) vex **is** tex	– vex evaluates cleanly and its value is of type indicated by tex (which evaluates cleanly)
(7) tex **is type,**	– type expression tex evaluates cleanly
(8) cli **is class,**	– cli is either empty-class or an identifier of a declared class
(9) ide **child of** cli	– ide is an identifier of a declared class which is a child of class indicated by cli
(10) tex1 **covers** tex2,	– tex1 and tex2 evaluate cleanly and…
(11) ide **is free**	– ide has not been declared

Procedure-oriented conditions

(constructors of conditions)

Examples:

- pr-ide **(val** fpv **ref** fpr**) begin** body **end imperative in** cl-ide,
- fu-ide **(val** fpv **ref** tex**) begin** body **return** vex **end functional in** cl-ide,
- ob-ide **(val** fpv **ref** ob-ide**) begin** body **end objectional in** cl-ide,
- **procedure** cl-ide.pr-ide **opened** ,
- **(pass actual val** apa-v **ref** apa-r **to formal val** fpa-v **ref** fpa-r **with** cl-ide **to** con

Procedure-oriented conditions (cont.)

[pr-ide **(val** fpc-v **ref** fpc-r**) begin** body **end imperative in** cl-ide].sta =
  is-error.sta ➔ error.sta
  let
    ((cle, pre, cov), sto) = sta
  cle.cl-ide = ? ➔ 'class unknown'
  let
    (cl-ide, tye, mee, obn) = cle.cl-ide
  mee.pr-ide = ? ➔ 'pre-procedure unknown'
  let
    declared-pre-proc = mee.pr-ide
    expected-pre-proc = create-imp-pre-pro.([fpd-v], [fpd-r], [body])
  declared-pre-proc ≠ expected-pre-proc ➔ fv
  true ➔ tv

Our condition claims three facts:

1. cl-ide is a name of a declared class,
2. pr-ide is a name of a procedure in this class,
3. pre-procedure pointed by pr-ide is equal to a pre-procedure that would be created by a declaration
   proc pr-ide **(val** fpc-v, **ref** fpc-r**) begin** body **end**.
   (a claim about a denotational effect of a declaration)

Procedure-oriented conditions (cont.)

```
[ pass actual val apa-v ref apa-r to formal val fpa-v ref fpa-r with cl-ide to con]
                                                          : WfState ⟶ BooValE
 [ pass actual val apa-v ref apa-r to formal val fpa-v ref fpa-r with cl-ide to con].sta =
  is-error.sta          ➔ error.sta
  let
    (env, sto)  = sta
    new-sto     = pass-actual.(fpa-v, fpa-r, apa-v, apa-r, cl-ide).env.sto
  is-error.new-sto   ➔ error.new-sto
  let
    new-sta = (env, new-sto)
  true                ➔ [con].new-sta
```

Condition con is satisfied after passing of actual parameters to formal parameters by a procedure call.

Assertions

asr : Assertion = **asr** Condition **rsa**

[asr] : WfState → WfState
[**asr** con **rsa**].sta =
  is-error.sta ➔ sta
  [con].sta = ? ➔ ?
  [con].sta : Error ➔ sta ◄ [con].sta
  [con].sta = fv ➔ sta ◄ ‘assertion not satisfied’
  true ➔ sta

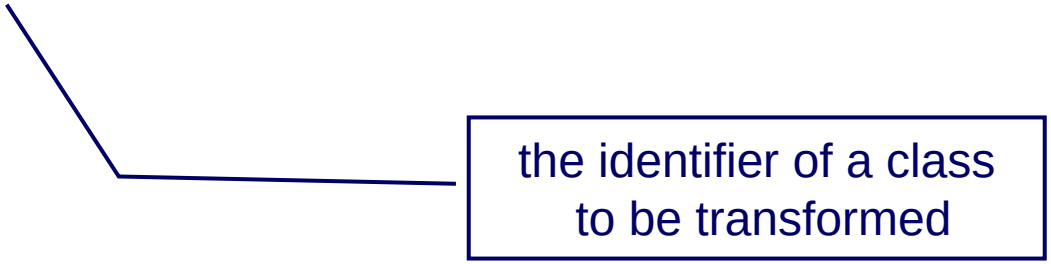
An error message will be generated by assertions in two situations:

1. when the value of the condition is an error,
2. when the condition is not satisfied.

Anchored class transformers

Class transformers:

ctd : ClaTraDen = Identifier ⟼ WfState → WfState.



Anchored class transformers:

act : AncClaTra = ClaTra **in** Identifier

with the following semantics:

[ctr **in** ide] : WfState → WfState
[ctr **in** ide] = [ctr].ide.

Specified programs

sin : SpeIns = specified instructions (specinstructions)

Instruction	
Assertion	
SpeIns ; SpeIns	
asr con **in** SpeIns **rsa**	on-zone instructions
off con **in** SpeIns **on**	off-zone instructions
if ValExp **then** SpeIns **else** SpeIns **fi**	
if-error ValExp **then** SpeIns **fi**	
while ValExp **do** SpeIns **od**	
skip-ins	

sde : SpeDec = specified declarations (specdeclarations)

Declaration	
Assertion	
SpeDec **;** SpeDec	
skip-dec	

Specified programs (cont.)

```
sct : SpeClaTra =          specified class transformers (spectransformers)
        AncClaTra                                    |
        Assertion                                    |
        SpeClaTra ; SpeClaTra                        |
        skip-sct
spp : SpeProPre =          specified program preambles (specpreambles)
      SpeDec                                         |
      SpeIns                                         |
      SpeProPre ; SpeProPre                          |
      skip-spp

spr : SpePro =             specified programs (specprograms)
      SpeProPre ; open procedures ; SpeIns           |
      SpeProPre                                      |
      SpeClaTra
```

Algorithmic conditions

con : AlgCondition =
  SpePro @ Condition | left algorithmic conditions
  Condition @ SpePro | right algorithmic conditions

[ ] : AlgCondition ⟼ WfState ⟼ {tv, fv}

[spr @ con].sta =
  (∃ sta1 : {con}) [spr].sta = sta1 ➔ tv
  true ➔ fv

[con @ spr].sta =
  (∃ sta1 : {con}) [spr].sta1 = sta ➔ tv
  true ➔ fv

i.e. [con].([spr].sta) = tv

We assume that conditions are closed under @.

Since algorithmic conditions are 2-valued, they are unambiguously identified by their truth domains:

{spr @ con} = [spr] ● {con}
{con @ spr} = {con} ● [spr]

None of them is error-transparent and:

- if spr is error transparent and con is error sensitive, then spr @ con is error negative,
- con @ spr need not be error sensitive.

Metaconditions
(formulas of our 2-valued logic of programs)

Atomic metaconditions:

con1 ⇨ con2	iff (def)	{con1} ⊆ {con2}	metaimplication; stronger than
con1 ⇔ con2	iff (def)	{con1} = {con2}	weak equivalence
con1 ⊑ con2	iff (def)	[con1] ⊆ [con2]	better definedness; more defined than
con1 ≡ con2	iff (def)	[con1] = [con2]	strong equivalence

MetaConditions = the least language that includes atomic metaconditions and is closed under 2-valued propositional connectives and quantifiers.

x > 0 **and-kl** $\sqrt[2]{x} > 2$	≡	x > 4	
$\sqrt[2]{x} > 2$	⇔	x > 4	but ≡ does not hold,
$\sqrt[2]{x} > 4$	⇨	x > 3	but neither ⇔ nor ⊑ holds.
$\sqrt[2]{x} < 2$	⊑	x < 4	if $\sqrt[2]{x}$ undefined for x < 0

con1	≡	con2	iff	con1 ⊑ con2 and con2 ⊑ con1
con1	⇔	con2	iff	con1 ⇨ con2 and con2 ⇨ con1
con1	≡	con2	implies	con1 ⇔ con2
con1	≡	con2	implies	con1 ⊑ con2
con1	⇔	con2	implies	con1 ⇨ con2

Three linguistic levels

Lingua	– a (classical) programming language
Lingua-V	– a language of validating programming metaprograms used to talk about programs
MetaLingua	– a language of a logic to talk about metaprograms

implies-kl : Condition x Condition ⟼ Condition		constructor in Lingua-V
⇨ : Condition x Condition ⟼ {tt, ff}		constructor in MetaLingua
implies : {tt, ff} x {tt, ff} ⟼ {tt, ff}		classical implication in MetaLingua

(con1 **implies-k** con2) ≡ **NT** implies con1 ⇨ con2

con1 ⇨ con2 does not imply (con1 **implies-kl** con2) ≡ **NT**.

Despite that the metaimplication $\sqrt[2]{x} > 4$ ⇨ x > 3 holds, the condition

$\sqrt[2]{x} > 4$ **implies-k** x > 3

is undefined for x < 0.


Thank you for
your attention