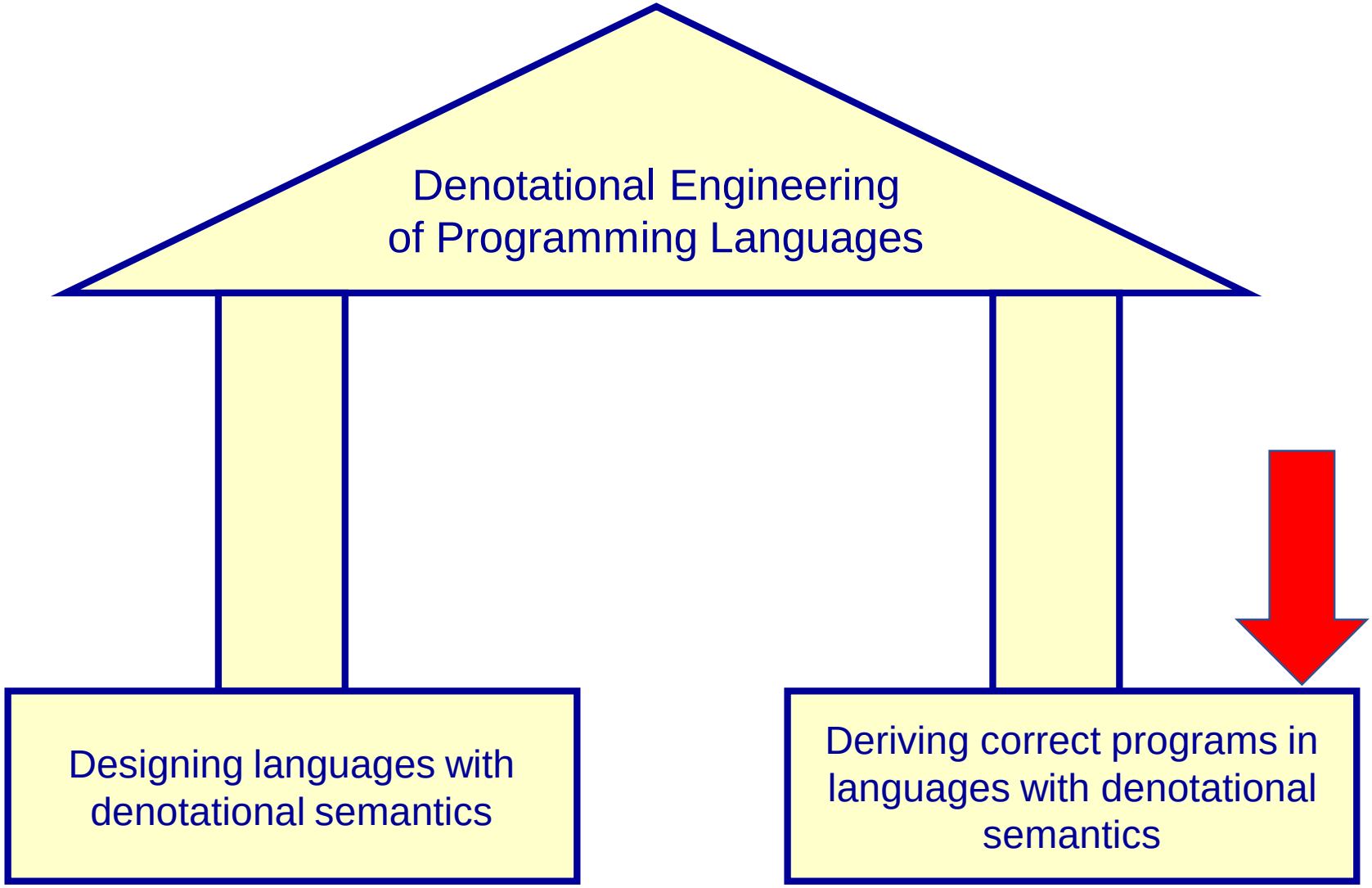
A Denotational Engineering of Programming Languages

…

Part 9: Lingua-2V Syntax and semantics
(Section 8.1 – 8.4 of the book)

Andrzej Jacek Blikle

April 6th, 2020


Denotational Engineering
of Programming Languages
Designing languages with
denotational semantics
Deriving correct programs in
languages with denotational
semantics

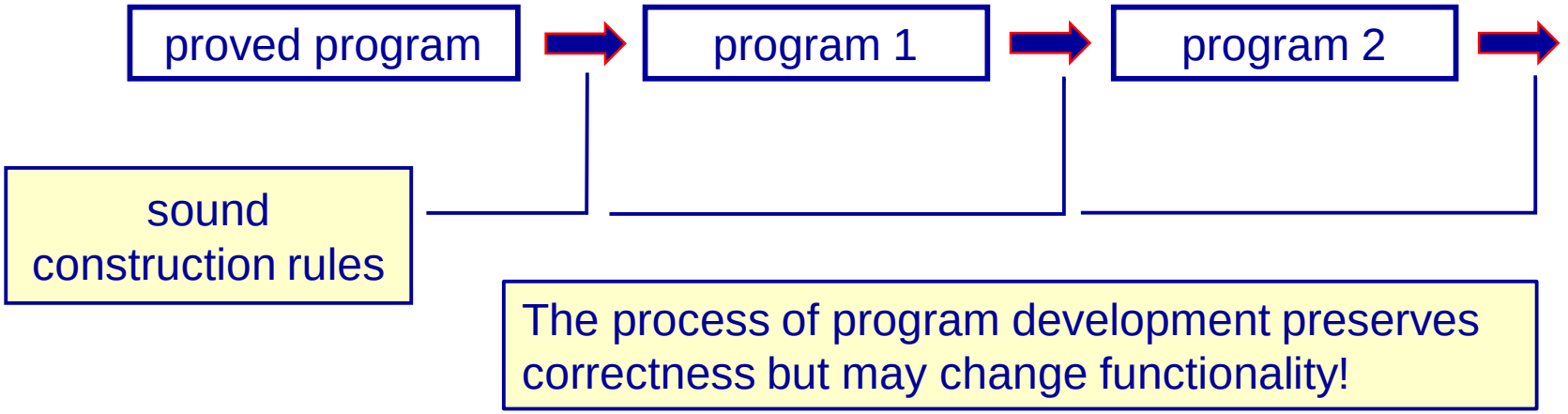
Validating programming

A metaprogram consists of two mutually nested (interleaved) layers:

- programming layer — a program in the usual sense,
- descriptive layer — pre- and post-conditions
  assertions “nested” in-between instructions

A metaprogram is said to be correct if its programming layer is cleanly totally correct wrt its pre- and post-condition.

Validating programming (program development)



The syntax and the semantics of Lingua-2V

A validating language

Lingua-nV = Lingua-n + descriptive layer

1. Conditions — denotations are three-valued partial predicates on states.
2. Specified instructions/programs — denotations are partial functions on states and the descriptive layer describes the properties of the programming layer.
3. Propositions — denotations are classical Boolean values tt and ff; propositions are split into three subcategories: (tezy)
   a. properties that express syntactic properties of programs, e.g. that a given procedure declaration appears in program's declaration,
   b. metaconditions that express the semantic properties of conditions, e.g. that one condition is satisfied iff another is satisfied as well,
   c. metaprograms that express total-correctness properties of programs which they include.

In building Lingua-Vn from Lingua-n we proceed from syntax to denotations.

Conditions

Auxiliary notations (v – value)

vt = (tt, (('Boolean'), TT))

vf = (ff, (('Boolean'), TT))

con : Condition =

basic conditions

DatCon | data-oriented conditions

DecCon | declaration-oriented conditions

SpecInstruction @ Condition | algorithmic conditions

composed conditions

(Condition **and** Condition) | (Condition **or** Condition) | (**not** Condition) |

(∀ Identifier **:** Condition) | (∃ Identifier **:** Condition)

Sco : Condition ⟼ State → ValueE

Notation:

[con] = Sco.[con]

{con} = {sta | [con].sta = tt}

Data-oriented conditions

Data-oriented conditions:

1. Boolean data-expressions of **Lingua**,
2. extended Boolean data-expressions referring to value-constructors which are not available in the source language e.g. `sorter-list` or `dae-1 = dae-2` for arbitrary data expressions.

McCarthy's logical connectives and Kleenee's quantifiers

∀ : Identifier x Condition ⟼ Condition

[`(∀ide: con)`].sta =
  is-error.sta ➔ error.sta
  let
    (env, (vat, 'OK')) = sta
  for every val : Value, [`con`].(env, (vat[ide/val], 'OK')) = vt ➔ vt
  there is val : Value, [`con`].(env, (vat[ide/val], 'OK')) = vf ➔ vf
  true ➔ 'never-false'

vt, err, ? (? needs not be computable)

regarded as an error message

Data-oriented conditions (cont.)

∃ : Identifier x Condition ⟼ Condition
[(∃ide:con)].sta =
  is-error.sta ➔ error.sta
  let
    (env, (vat, 'OK')) = sta
  there is val : Value, [con].(env, (vat[ide/val], 'OK')) = vt ➔ vt
  for every val : Value, [con].(env, (vat[ide/val], 'OK')) = vf ➔ vf
  true ➔ 'never-false'

[(∀ide: con)].sta = vf even if sometimes error

[(∃ide: con)].sta = vt even if sometimes error

Declaration-oriented conditions

is-free(ide) – ide is not declared

ide **is** tex – ide is declared as a variable of type defined by tex

e.g.:

```
length is real
employee is
  record-type
    c-name as word,
    f-name as word
  ee
```

ide **is-type** tex – ide is declared as a type constant of type defined by tex

conformant(fpa-v, fpa-r, apa-v, apa-r)
– list of parameters are dynamically compatible

Declaration-oriented conditions (cont.)

`ide` is bound in sta to a procedure whose declaration is `ipd`

[`ide` **`proc-with`** `ipd`].sta = vt

iff

(1) sta does not carry an error

(2) `ipd` is a declaration of `ide`, i.e. is of the form

```
proc ide (val ForPar ref ForPar) Program endproc,
```

(3) there exists sta-ini, such that sta = Sipd.[`ipd`].sta-ini

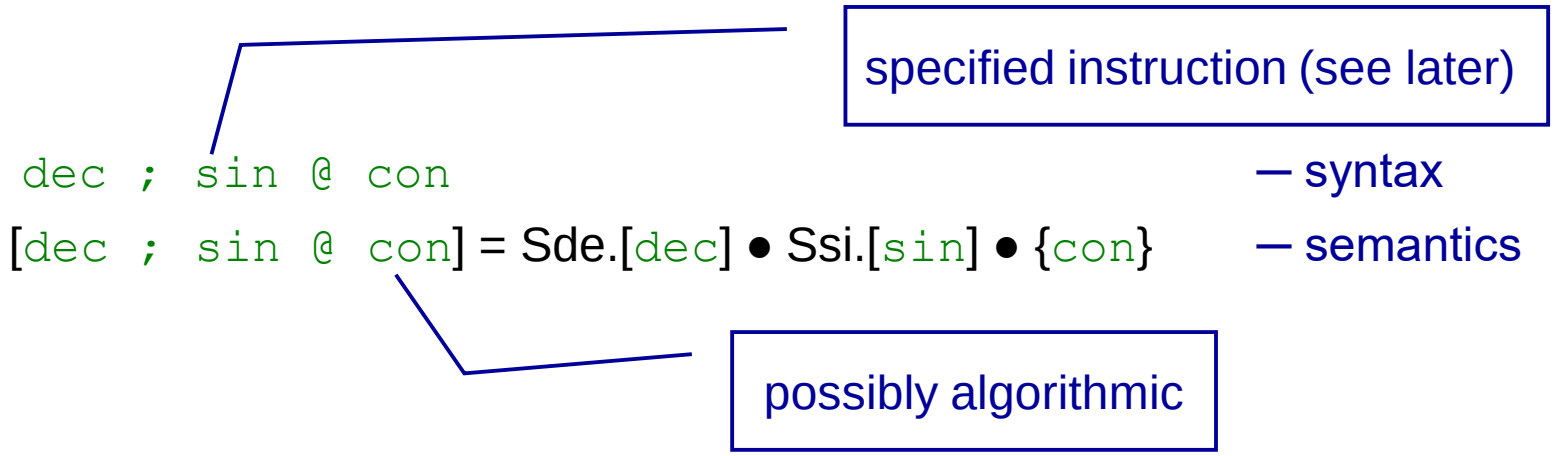
otherwise

[`ide` **`proc-with`** `ipd`].sta = vf

Analogous for functional procedures:

`ide` **`fun-with`** `fpd`

Algorithmic conditions



`sin @ con` is the weakest precondtion which guarantees that `sin` terminates and the terminal state satisfies `con`.

Banachowski Lech, Kreczmar Antoni, Mirkowska Grażyna, Rasiowa Helena, Salwicki Andrzej, *An introduction to Algorithmic Logic — Metamathematical Investigations of Theory of Programs*, T. 2: Banach Center Publications. Warszawa PWN, 1977, s. 7-99, series: Banach Center Publications, vol.2

Specified instructions

sin : SpecInstruction =

Instruction	|
asr Condition **rsa**	|
if DatExp **then** SpecInstruction **else** SpecInstruction **fi**	|
if-error DatExp **then** SpecInstruction **fi**	|
while DatExp **do** SpecInstruction **od**	|
SpecInstruction **;** SpecInstruction	

Ssi : SpecInstruction ⟼ State → State

Ssi.[**asr** con **rsa**].sta **=**

- is-error.sta ➔ sta
- [con].sta = ? ➔ ?
- [con].sta =vt ➔ sta
- **true** ➔ sta ◀ 'assertion-not-satisfied'

in all other cases semantic clauses are as in Lingua-2

ff or error

Specified instructions (cont.)

Two special colloquialisms

`asr` `con: sin` **`rsa`**

Insert **`asr`** `con` **`rsa`** between any two atomic instructions.

`off` `sin` **`asr`**

Remove all assertions from `sin`.

See the corresponding restoring transformation in Sec. 8.3 of the book.

Propositions

Tezy

syntactic propositions — describe properties of the syntax of programs

metaconditions — describe semantic properties of conditions

metainstructions — describe semantic properties of instructions

metaprograms — describe semantic properties of programs

Propositions evaluate to tt or ff

When we talk about properties of programs
we remain in classical logic.

A

Syntactic propositions

IS-CORRECT(dec)	— no identifier declared twice in dec,
ide **DEC-AS-PRO** ipd **IN** dec	— ide is declared by ipd in dec
ide **DEC-AS-FUN** fpd **IN** dec	— ide is declared by ipf in dec
ide **NOT-IN** dec	— ide has not been declared in dec
dec-1 **SEPARATED-FROM** dec-2	— the sets of identifiers declared in dec-1 and dec-2 are disjoint.

Metaconditions

Metaconditions describe such properties of conditions that refer to their denotations.

Metaconditions do not belong to the syntax of Lingua-2V. They belong to the syntax of MetaSoft.

⇨ , ⊑ , ⇔ , ≡ : Condition x Condition ⟼ Proposition — metapredicates

{con} = {sta : [con].sta = vt}

DEFINITIONS

con-1 ⇨ con-2	**iff**	{con-1} ⊆ {con-2}	stronger/weaker than(metaimplication)
con-1 ⊑ con-2	**iff**	[con-1] ⊆ [con-2]	less/more defined than
con-1 ⇔ con-2	**iff**	{con-1} = {con-2}	weakly equivalent
con-1 ≡ con-2	**iff**	[con-1] = [con-2]	strongly equivalent

SOME PROPERTIES

con-1 ≡ con-2	is equivalent to	(con-1 ⊑ con-2 and con-2 ⊑ con-1)
con-1 ⇔ con-2	is equivalent to	(con-1 ⇨ con-2 and con-2 ⇨ con-1)
con-1 ⇔ con-2	implies	con-1 ⇨ con-2

Metaconditions (cont.)

`pre` ⇨ `ins @ post` — clean total correctness (for deterministic `ins`)

`con` ⇨ `ins @ con` — strong invariant of `ins`

`x>0` **`and`** $\sqrt[2]{x}$ `> 2` ≡ `x > 4`

$\sqrt[2]{x}$ `> 2` ⇔ `x > 4` but ≡ does not hold

$\sqrt[2]{x}$ `< 2` ⊑ `x < 4` but neither ≡ nor ⇔ holds

$\sqrt[2]{x}$ `> 4` ⇨ `x > 3` but neither ⇔ nor ⊑ holds

Metaimplication versus implication

Three logical levels

implies	: Condition x Condition ⟼ Condition	- syntactic constructor
⇨	: Condition x Condition ⟼ {tt, ff}	- metaimplication
implies	: {tt, ff} x {tt, ff} ⟼ {tt, ff}	- **MetaSoft** implication

(con-1 **implies** con-2) ≡ **TT** implies con-1 ⇨ con-2

Lingua — MetaSoft — Lingua-2V

The converse implication is not true.

$\sqrt[2]{x} > 4 \Rightarrow x > 3$ but $\sqrt[2]{x} > 4$ **implies** x > 3 is undefined for x < 0

Equivalence and congruence

$\approx \subseteq A \times A$ — equivalence relation

$a \approx a$ — reflexive
$a \approx b$ then $b \approx a$ — symmetric
$a \approx b$ and $b \approx c$ then $a \approx c$ — transitive

$\approx \subseteq A \times A$ — congruence relations wrt $F : A^n \rightarrow A$

$a_i \approx b_i$ for $i = 1;n$ implies $F.(a_1,\ldots,a_n) \approx F.(b_1,\ldots,b_n)$

Metaconditions (cont.)

Some properties of ≡ and ⇔.

Lemma 8.4.2-1 Relations ≡ and ⇔ are both equivalences.

Lemma 8.4.2-2 Strong equivalence is a congruence wrt **`and`**, **`or`** and **`not`**,

Lemma 8.4.2-3 Weak equivalence is a congruence wrt **`and`** and **`or`**.

Weak equivalence is not a congruence wrt **`not`**.

$\sqrt[2]{x}$ `> 2 ⇔ x > 4` is satisfied but

$\sqrt[2]{x}$ `≤ 2 ⇔ x ≤ 4` is not (x = -1)

Lemma 8.4.2-4 The operators **`and`** and **`or`** are strongly and (of course also weakly) associative.

Lemma 8.4.2-7 The de Morgan's laws for **`and`**, **`or`** and for the negation of quantifiers are satisfied with strong equivalence

Lemma 8.4.2-8 Conjunction is weakly commutative.

Metaconditions (cont.)

Contextual metaconditions

DEFINITIONS

`con-1 ≡ con-2 whenever con` means `con and con-1 ≡ con and con-2`

`con-1 ⇔ con-2 whenever con` means `con and con-1 ⇔ con and con-2`

EXAMPLES

`n > x²` ≡ $\sqrt[2]{n}$ `> x whenever (n ≥ 0 and x ≥ 0)`

`n > x²` ⇔ $\sqrt[2]{n}$ `> x whenever x ≥ 0`

Metainstructions

`if` `dat` **`then`** `sin` **`fi limited-replicability in`** `con`

satisfied iff

[{`dat`}] Ssi.[`sin`] has limited replicability in {`con`}.

Metaprograms

mpr : MetaProgram =

pre Condition :

Declaration ;

SpecInstruction

post Condition

Smp : MetaProgram ⟼ {tt, ff}

Sde.[**pre** prc : dec ; sin **post** poc] = tt

iff (def)

{prc} ⊆ Sde.[dec] ● Ssi.[sin] ● {poc} i.e.

prc ⇨ dec ; sin @ poc

A metaprogram mpr is said to be correct if Smp[mpr] = tt.

Total correctness with clean termination.

Correctness of a metaprogram implies that for every execution that starts in {prc}:

1. dec, sin, poc do not generate an error,
2. all states of the execution are adequate for dec,
3. all assertions in sin are satisfied,
4. program terminates and terminal state does not carry an error.

Metaprograms

Correctness-preserving replacements in metaprograms

Weakly equivalent conditions in:
- preconditions,
- postconditions,
- assertions.

Weaker defined by stronger defined `dae-1` ⊑ `dae-2`, **in**:
- Boolean expressions,
- assertions.

In the sequel whenever we write

```
pre con-pr : dec;sin post con-po
```

we mean that

Smp[`pre con-pr : dec;sin post con-po`] = tt

Clean evaluations of expressions

DEF. A data expression `dae` evaluates cleanly under condition `con`, if

`con ⇨ dae=dae`

An equality from descriptive level of **Lingua-V**.


Thank you for
your attention