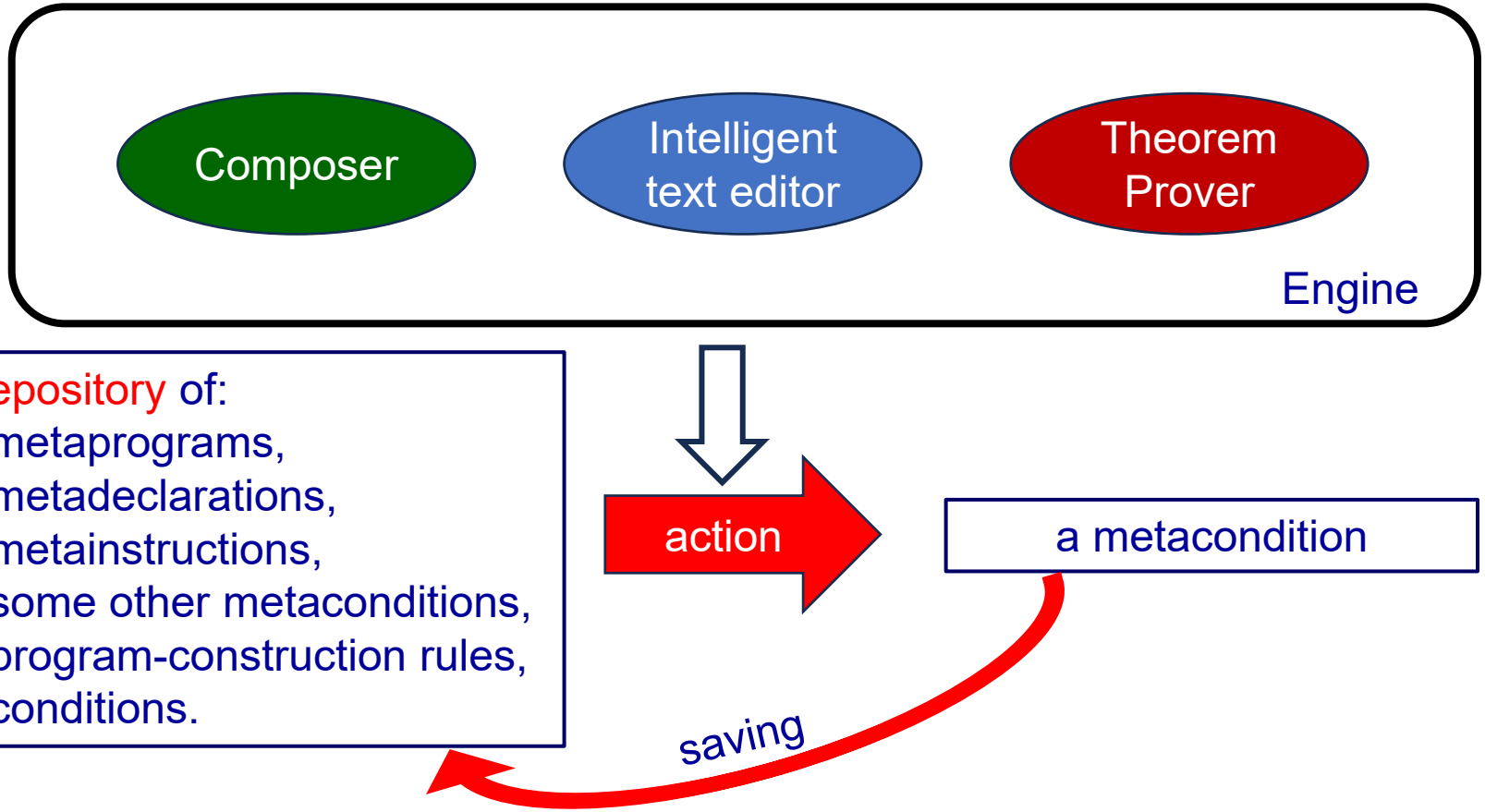
Investigations on the logical aspects of ecosystems for programmers in Lingua-V

Andrzej Jacek Blikle

October 4th, 2025

Preliminaries

An ecosystem of programmers in Lingua-V



Steps of metacondition's development regarded as proofs:

ex-ante constructive (implicit) proofs – involving Composer
ex-post analytic (explicit) proofs – involving Theorem Prover

We need a formalized theory where such proofs or constructions may be carried out.

Our way to a formalized theory of **Lingua-V**

1. Extending the denotational model of **Lingua** to **Lingua-V**:
   a. extending **AlgSyn** to **AlgSyn-V,**
   b. extending **AlgDen** to **AlgDen-V**.
2. Lifting a denotational model of **Lingua-V** to **Lingua-D**:
   a. lifting **AlgSyn-V** to **AlgSyn-D**,
   b. lifting **Alg-Den-V** to **AlgDen-D**.
3. Building a denotational model of **Lingua-E** – a language of an ecosystem:
   a. Building an algebra of denotations **AlgDen-E**
   b. Building an algebra of syntax **AlgSyn-E**

A hierarchy of languages:

Lingua — a source programming language,
Lingua-V = **Lingua** + conditions + metaconditions
Lingua-D = **Lingua-V +** variables
Lingua-E — **Lingua-DA**-based language of ecosystem

running over denotations of **Lingua-V**

The need of a formalized theory

We need a formalized theory rich enough to prove
lemmas about metacomponents
and in particular
the soundness of program-construction rules

A formalized theory:
- a formal language **Lingua-D,**
- a set of axioms,
- a set of inference rules.

An ecosystem as a set of tools

Examples of theorems to be proved

x **is** integer ⇨ x < x + 1

(x+1 ≤ isrt(n))
≡
((x+1)^2 ≤ n)
whenever (x, k is integer) **and-k** (x, y ≥ 0) **and-k** ((isrt(n)+1)^2 ≤ M) **and-k** (x ≤ isrt(n))

the largest integer in the implementation

We shall not need to prove the correctness of metaprograms!

Correct metaprograms will be developed.

An example of a program development (1)

Program to be developed

```
pre (x is free) and-k (y is free) :
    let x be integer tel;
    let y be integer tel;
    x := 3;
    y := x+1 ;
    x := 2*y
post (x is integer) and-k (y is integer) and-k (x < 10)
```

The names of the elements of repository are printed in **blue:**
P - program
A - axiom
L - lemma

The underlined
ide, tex,…
are variables in **Lingua-D**

Step 1: synthesize the declaration of x

A1: pre (ide is free) and-k (tex is type)
let ide be tex tel
post var ide is tex

a rule in RUL

P1 : pre (x is free) and-k (integer **is type**)
let x be integer **tel**
post var x is integer

substitute(A1, [ide/x, tex/integer], P1

an action of Lingua-E

An example of a program development (2)

Step 2: remove tautology

to be eliminated

P1 : pre (x **is free**) **and-k** (integer **is** type)
let x **be** integer **tel**
post var x **is** integer

P2 : pre (x **is free**)
let x **be** integer **tel**
post var x **is** integer

P3 : pre (y **is free**)
let y **be** integer **tel**
post var y **is** integer

Some axioms to be applied (all are listed in the report):

A2: **error-transparent**(con) **implies** (con ⇨ **NT**) – the definition of **NT**
A3: (**error-transparent**(con1) **and** (**NT** ⇨ con2)) **implies** (con1 **and-k** con2 ⇔ con1)
A4: (con1 ≡ con2) **implies** (con1 ⇨ con2)
A5: (con1 ≡ **NT**) **implies** ((con1 **and** con2) ≡ con1))
A6: integer **is type** ≡ **NT**

A8:

pre prc **:** sin **post** poc
prc ⇔ prc-1
---
pre prc-1 **:** sin **post** poc

error-transparency is crucial:
con.er-sta = tt and
(con **and-k NT**).er-sta = err

An example of a program development (3)

Step 3 and 4: the strengthening of conditions

P2 : pre (x **is free**)
let x **be** integer **tel**
post var x **is** integer

P4 : pre (x **is free**) **and-k** (y **is free**)
let x **be** integer **tel**
post var x **is** integer **and-k** (y **is free**)

P3 : pre (y **is free**)
let y **be** integer **tel**
post var y **is** integer

P5 : var x **is** integer **pre** (y **is free**)
let y **be** integer **tel**
post (**var** y **is** integer) **and-k**
(**var** y **is integer**)

Lemmas to be applied:

L2: different(ide1, ide2) **implies** ((ide-1 **is free**) **is irrelevant for** (**let** ide-2 **be** tex)),
L3: different(ide1, ide2) **implies** ((ide-1 **is** tex-1) **is irrelevant for** (**let** ide-2 **be** tex-2)),

L4:
pre prc **:** sin **post** poc
con **irrelevant for** sin
---
pre prc **and-k** con **:** sin **post** poc **and-k** con

An example of a program development (4)

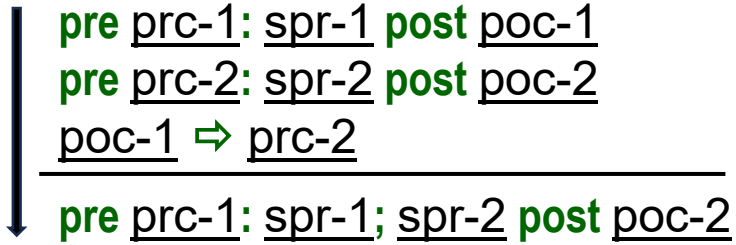
Step 5: sequential composition of P4 and P5:

P4 : pre (x **is free**) **and-k** (y **is free**)
let x **be** integer **tel**
post var x **is** integer **and-k** (y **is free**)

P5 : var x **is** integer **pre** (y **is free**)
let y **be** integer **tel**
post (**var** y **is** integer) **and-k**
(**var** y **is integer**)

P6 : pre (x **is free**) **and-k** (y **is free**)
let x **be** integer **tel** ;
let y **be** integer **tel**
post var x **is** integer **and-k** (y **is integer**)

Lemma to be applied:



An example of a program development (5)

Step 6: the development of assignment

A9: **pre** sin **@** con
sin
post con

substitution

P6.1: **pre** x := 3 **@** **(var** x **is** integer**) and-k (var** y **is** integer**) and** (x = 3)
x := 3
post (var x **is** integer**) and-k (var** y **is** integer**) and-k** (x = 3)

x := 3 **@ (var** x **is** integer**) and-k (var** y **is** integer**) and** x = 3 ⟺
(var x **is** integer**) and-k (var** y **is** integer**)**

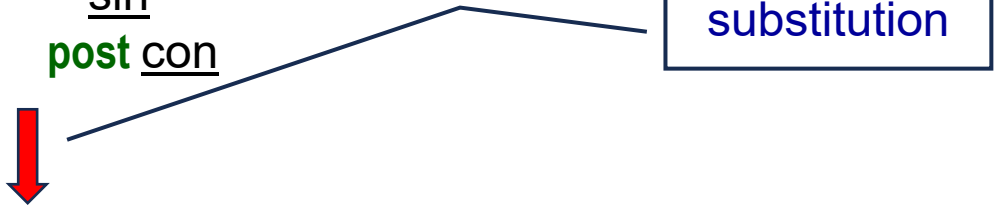
substitution and the elimination of @

P7: **post (var** x **is** integer**) and-k (var** y **is** integer**)**
x := 3
post (var x **is** integer**) and-k (var** y **is** integer**) and-k** (x = 3)

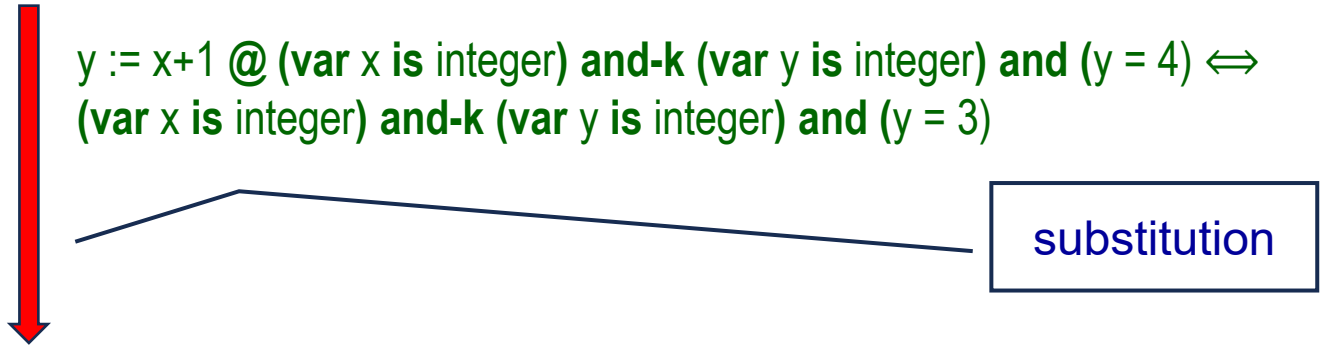
An example of a program development (6)

Step 7: the development of assignment

A9: **pre** sin **@** con
sin
post con



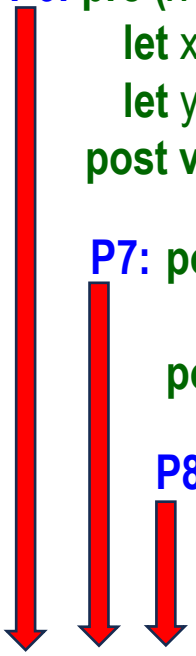
pre y := x+1 **@** **(var** x **is** integer**) and-k (var** y **is** integer**) and** (y = 4**)**
y := x+1
post (var x **is** integer**) and-k (var** y **is** integer**) and-k** (y = 4**)**



P8: **post (var** x **is** integer**) and-k (var** y **is** integer**) and** (y = 3**)**
y := x+1
post (var x **is** integer**) and-k (var** y **is** integer**) and-k** (y = 4**)**

An example of a program development (7)

Step 8: sequential composition



P6: **pre** (x **is free**) **and-k** (y **is free**)
let x **be** integer **tel** ;
let y **be** integer **tel**
post var x **is** integer **and-k** (y **is integer**)

P7: **post** (**var** x **is** integer) **and-k** (**var** y **is** integer)
x := 3
post (**var** x **is** integer) **and-k** (**var** y **is** integer) **and-k** (x = 3)

P8: **post** (**var** x **is** integer) **and-k** (**var** y **is** integer) **and** (y = 3)
y := x+1
post (**var** x **is** integer) **and-k** (**var** y **is** integer) **and-k** (y = 4)

P9: **pre** (x **is free**) **and-k** (y **is free**)
let x **be** integer **tel**
let y **be** integer **tel**
x := 3;
y := x + 1
post (**var** x **is** integer) **and-k** (**var** y **is** integer) **and-k** (y = 4)

An example of a program development (8)

Step 9: the development of an assignment

P10 : **pre (var** x **is** integer**) and-k (var** y **is** integer**) and-k** (y = 4)
x := 2*y
post (var x **is** integer**) and-k (var** y **is** integer**) and-k** (y = 4**) and-k** (x = 8)

(var x **is** integer**) and-k** (y=4**) and-k** (x = 8) ⇨ **(var** x **is** integer**) and-k** (x < 10)

P11 : **pre (var** x **is** integer**) and-k (var** y **is** integer**) and-k** (y = 4)
x := 2*y
post (var x **is** integer**) and-k (var** y **is** integer**)** (x < 10)

theorem prover:
(8 < 10) **≡ NT**

Axiom to be applied:

A10:

pre prc: spr **post** poc
poc ⇨ prc-1
---
pre prc **:** spr **post** poc-1

An example of a program development (8)

Step 9: sequential composition — RPM(1), RUL(3)

```
P9: pre (x is free) and-k (y is free)
      let x be integer tel
      let y be integer tel
      x := 3;
      y := x + 1
    post (var x is integer) and-k (var y is integer) and-k (y = 4)

P11:  pre (var x is integer) and-k (var y is integer) and-k (y = 4)
        x := 2*y
      post (var x is integer) and-k (var y is integer) (x < 10)
```

```
P12: pre (x is free) and-k (y is free) :
       let x be integer tel;
       let y be integer tel;
       x := 3;
       y := x+1 ;
       x := 2*y
     post (x is integer) and-k (y is integer) and-k (x < 10)
```

target program

Numbers of uses:

composer – 40
prover – 1

A recollection of formalized theories

A recollection of formalized theories (1)
First-order theories – preliminaries

In first-order theories we talk about:

ele : Uni — elements of a set called a universe
fun : Uni^{cn} ⟼ Uni — functions with n ≥ 0
pre : Uni^{cn} ⟼ Bool — predicates with n ≥ 0

A language of first-order theories includes two syntactic categories

terms — represent functions
formulas — represent predicates

names and separators will be printed in green

variables will be printed in black

Primitives of syntax

var : Variable — variables (running over Uni)
fn : Fn — function names
pn : Pn — predicate names
sep : Separator — separators, e.g.: „ (” , „) ” , „ , …

Alphabet = Variable | Fn | Pn | Separator
arity : Fn | Pn ⟼ {0, 1, 2,…} — arity of names

A recollection of formalized theories (2)

First-order theories – the language

```
var : Variable =
   x | y | z | x-1 | y-1 | z-1 |…        variables may have indices

ter : Term =
   mk-term(Variable)                 |  make a term from a variable
   fn()                              |  for all fn : Fn with arity.fn = 0
   fn(Term, … ,Term)                 |  for all fn : Fn with arity.fn = n
                 n
for : Formula =
   true                              |
   false                             |
   pn(Term, … ,Term)                 |  for all pn : Pn with arity.pn = n
                 n
   not(Formula)
   and(Formula, Formula)             |
   or(Formula, Formula)              |
   implies(Formula, Formula)         |
   (∀ Variable) Formula              |
   (∃ Variable) Formula              |
```

ground formulas – no variables; e.g. 1 < 2

free formulas – have variables; e.g., x < 2

A recollection of formalized theories (3)
An example of a first-order theory of Peano arithmetic (1)

Language

Variable = {x, y, z,…, x-1, x-2,…}, variables may have indices,
Fn = {zer, suc)
Pn = {num, equ}

with

arity.zer = 0 zer() or just zer represents number zero
arity.suc = 1 suc(x) is the successor of x
arity.num = 1 num(x) means that x is a number
arity.equ = 2 equ(x,y) means that x and y are equal

Examples of formulas

true, num(zer),
equal(suc(zer), suc(x)),
and(equal(suc(zer), suc(x)), equal(suc(suc(y)), suc(suc(x))),
(∀x) not((equal(x, suc(x))).

A recollection of formalized theories (4)
An example of a first-order theory of Peano arithmetic (2)

A reader-friendly notation:
(ter-1 = ter-2) for equ(ter-1, ter-2),
(for-1 and for-2) for and(for-1, for-2)
(pre-1 → pre-2) for implies(pre-1, pre-2).

Axioms

x = x
x = y → y = x
(x = y and y = z) → x = z
(x-1 = y-1 and … and x-n = y-n) → (fn(x-1,…,x-n) = fn(y-1,…,y-n)) for all fn : Fn
(x-1 = y-1 and … and x-n = y-n) → (pn(x-1,…,x-n)= pn(y-1,…,y-n)) for all pn : Pn

num(zer)	zero is a number,
num(x) → num(suc(x))	the successor of a number is a number,
num(x) → not (suc(x) = zer)	the successor of a number never equals zero,
x = suc(y) and x = suc(z) → y = z	suc is a reversible function

A recollection of formalized theories (5)
Interpretation and semantics (1)

An interpretation of a language of a formalized theory:

Int = (Uni, F, P)

with

Uni – set called universe, its elements are called primitive elements,

F – function; F[fn] : Uni^{cn} ⟼ Uni for arity.fn = n
F[fn] : ⟼ Uni for arity.fn = 0

P – function; P[pn]: Uni^{cn} ⟼ Bool;
P[true] = tt, P[false] = ff

A valuation is a total function that assigns primitive elements to variables:

vlu : Valuation = Variable ⟼ Uni

The semantics of variables, terms and formulas:

SV : Variable ⟼ Variable
ST : Term ⟼ Valuation ⟼ Uni
SF : Formula ⟼ Valuation ⟼ Bool

A recollection of formalized theories (6)
Interpretation and semantics (2)

The semantics of variables:

ST : Variable ⟼ Variable
ST.[var] = var

The semantics of terms:

ST : Term ⟼ Valuation ⟼ Uni

ST.[mk-term(var)].vlu	= vlu.var,	var : Variable
ST[fn(ter-1,…,ter-n)].vlu	= F[fn].(ST[ter-1].vlu,…,ST[ter-n].vlu)	arity.fn = n

The semantics of formulas:

SF : Formula ⟼ Valuation ⟼ Bool

SF[true].vlu	= tt	
SF[false].vlu	= ff	**and**, **not** are classical metaoperations
SF[pn(ter-1,…,ter-n)].vlu	= P[pn].(ST[ter-1].vlu,…,ST[ter-n].vlu),	arity.fn = n
SF[(for-1 and for-2)].vlu	= SF[for-1].vlu **and** SF[for-2].vlu	
SF[not(for)].vlu	= **not** SF[for]	
SF[(∀ var)for].vlu	= tt iff for every ele : Uni,	for.vlu[var/ele] = tt
SF[(∃ var)for].vlu	= tt iff there exists ele : Uni, such that	for.vlu[var/ele] = tt

A recollection of formalized theories (6)
Satisfaction, models and validity

For a given interpretation:

Int = (Uni, F, P)

A formula for is said to be satisfied in Int if:

SF[for].vlu = tt for every vlu : Valuation

An interpretation Int is said to be a model of a theory with set of axioms A if all axioms are satisfied in Int.

A formula for is said to be valid in a theory with a set of axioms A, in symbols

A |= for

if it is satisfied in every model of this theory.

E.g.: Peano |= not(zer = suc(zer))

A recollection of formalized theories (7)
Deduction – a way of proving the validity of formulas

A |- for for is a theorem in the theory with axioms A if it can be derived from A by means of deduction rules

The main deduction rules

Rule of substitution

$$\frac{A \vdash for(x)}{A \vdash for(ter)}$$

x free in for(x)
ter – an arbitrary term

Rule of detachment

$$\frac{A \vdash for1 \quad A \vdash for1 \rightarrow for2}{A \vdash for2}$$

Rule of generalization

$$\frac{A \vdash for(x)}{A \vdash (\forall x)\ for(x)}$$

x free in for(x)

Gödel's completeness theorem

In every first-order theory with axioms A

A |= for iff A |- for

A recollection of formalized theories (7)
The weaknesses of first-order theories

Every first-order theory which has an infinite model,
has infinitely many non-isomorphic models.

Colloquially: In first-order theories we never know what we are talking about.

Three models of Peano arithmetic:

1. Uni = NatNum, zer = 0, suc(x) = x+1 all elements of Uni are reachable
2. Uni = ReaNum, zer = 0, suc(x) = x+1 not all elements of Uni are reachable
3. Uni = NatNum | {0,5}, zer = 0, suc(x) = x+1 for x : NatNum, suc(0,5) = 0,5

standard model

In first-order Peano arithmetic
$x \neq suc(x)$
is not a theorem!

A recollection of formalized theories (8)
Second-order theories

Second-order Peano's arithmetics:

- All first-order axioms
- A second-order axiom: (P(zer) and (P(x) → P(suc(x)) → (num(x) → P(x))

P – a predicative variable (only one such variable in this case)

Two metatheorems:

1. All models of 2nd-order Peano's arithmetic are isomorphic to the standard model.
2. 2PA |- x ≠ suc(x)

Proof of 2. by induction:

1. 0 ≠ suc(0) – is an axiom
2. if x ≠ suc(x) then suc(x) ≠ suc(suc(x)) – suc is reversible by an axiom
3. x ≠ suc(x) for all x – by the 2-order axiom

In second-order theories with arithmetic
we can carry out proofs by induction.

A recollection of formalized theories (8)
The weaknesses and strengths of second-order theories

Gödel's incompleteness theorem

In second-order theories with arithmetics there exist valid formulas which can't be proved, i.e. |= for but not |- for.

Gödel's adequacy theorem

In second-order theories with arithmetic every proved formula is valid i.e. if |- for then |= for.

we can trust the theorems that have been proved

A recollection of formalized theories (9)
Consistency and completeness of formalized theories

Definition

A theory is consistent if it has a model.

Theorems:

1. A theory is consistent iff there exists formula for such that for and **not**(for) are valid.
2. A theory is consistent iff there exists a formula for which is not valid.
   a → (**not** a → b)

Inconsistent theories do not distinguish between truth and falsity!

Definition

pełna

A theory is complete if it is consistent and for any formula for
either A |- for or A |- **not**(for).

The majority of interesting theories are incomplete!

Observation (which may lead to confusion)

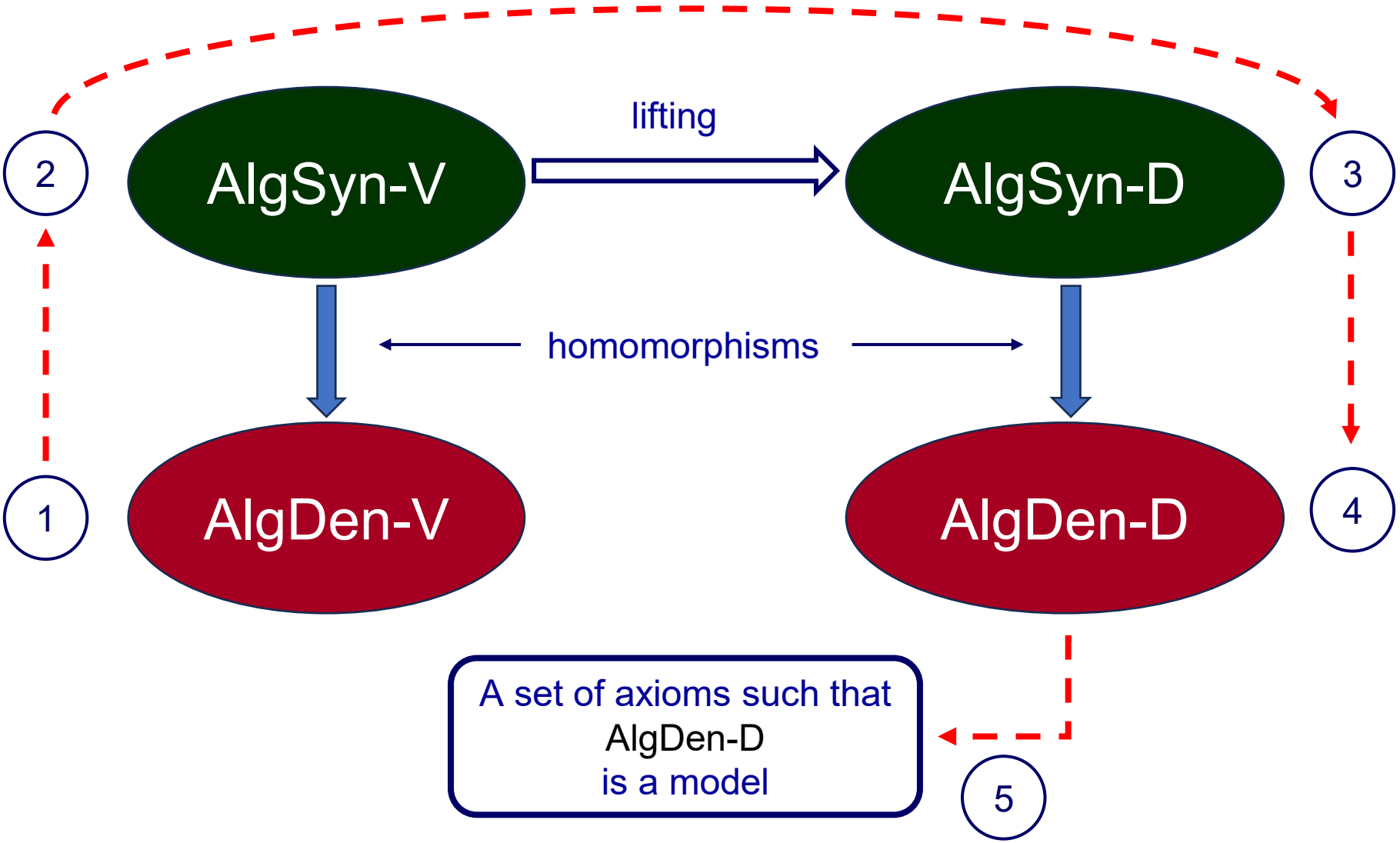
In every theory for any formula for
A |- (for **or not**(for)).

Our theory of denotations must be consistent but will not be complete

Building a language of a formalized theory of the denotations of of a programming language

Building Language-D for Language-V (1)

D – denotations



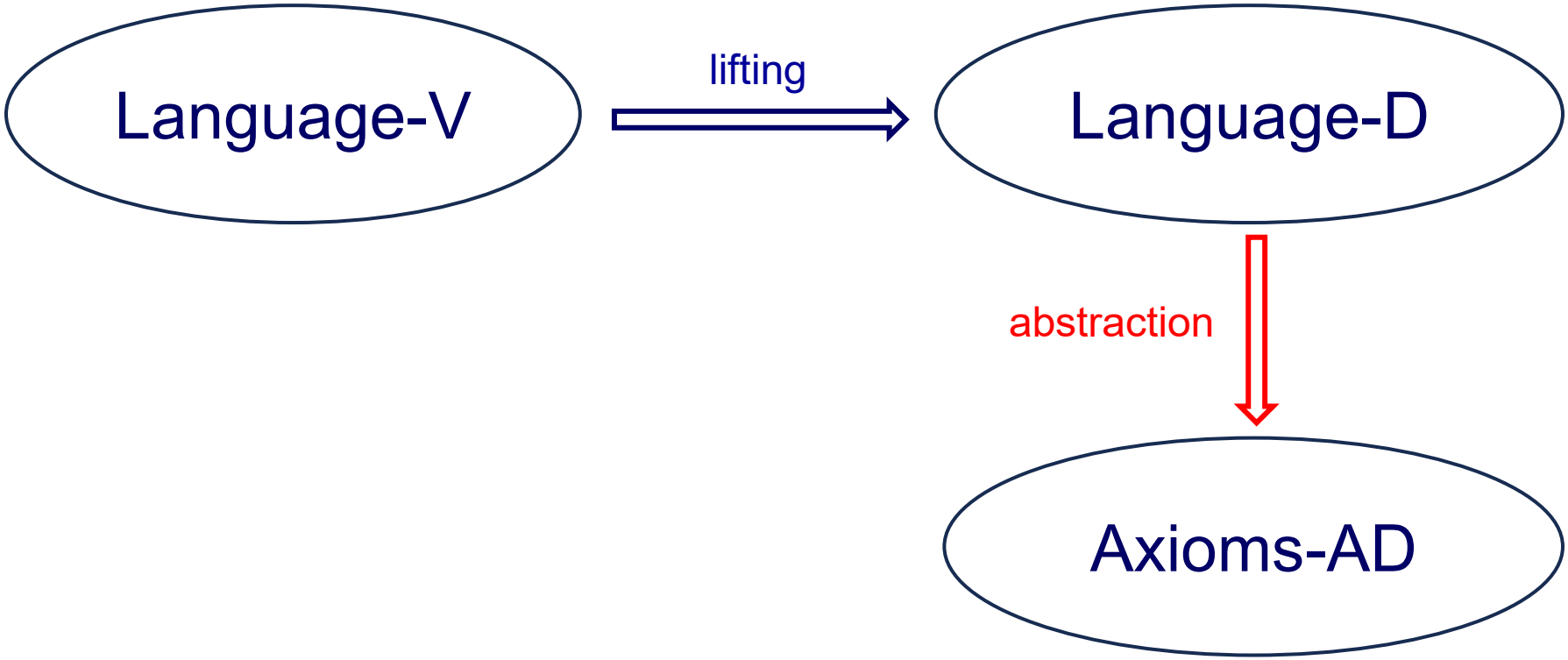
Lifting Language-V to Language-D (2)

How theories and models are built

mathematical logic: axioms ➔ its models

working mathematician: a model ➔ its axioms

our case



Building Language-D for Language-V (2)
From AlgDen-V to AlgSyn-V

Source algebra (Language-V)

AlgDen-V = (Sig-V, CarDen-V, FunDen-V, carDen-V, funDen-V)
Sig-V = (Cn-V, Fn-V, arity-V, sort-V)

Cn-V = {cn-1,…,cn-n}
Fn-V = {fn-1,…, fn-k}

Assumptions on **AlgDen-V**:

boo : Cn-V
and, or, implies, not : Fn-V

carDen-V.boo = Bool where Bool = {tt, ff}
and, or, implies, not represent classical connectives

Note: boo is the sort of metaconditions rather than conditions!

Three algebras to be derived:

AlgSyn-V = (Sig-V, CarSyn-V, FunSyn-V, carSyn-V, funSyn-V) abs. syntax
AlgSyn-D = (Sig-D, CarSyn-D, FunSyn-D, carSyn-D, funSyn-D) abs. syntax
AlgDen-D = (Sig-D, CarDen-D, FunDen-D, carDen-D, funDen-D)

Building Language-D for Language-V (3)
From AlgDen-V to AlgSyn-V

The abstract-syntax grammar of **AlgDen-V**

```
ter : Term.cn =                                          for all cn : Cn-V – {boo}
      fn(Term.cn-1,…,Term.cn-n)    |  for all fn : Fn-V with
                                          arity.fn = (cn-1,…,cn-n)
                                          sort.fn = cn

for : Formula  =                                         instead of Term.boo
      pn(Term.cn-1,…,Term.cn-n)    |  for all pn : PreNam with
                                          arity.pn = (cn-1,…,cn-n)
      and(Formula, Formula)        |
      or(Formula, Formula)         |
      implies(Formula, Formula)    |
      not(Formula)
```

Building Language-D for Language-V (4)
From AlgDen-V to AlgSyn-V introducing variables

Sets of variables:

inv : IndVar — individual variables
fuv : FunVar — functional variables
prv : PreVar — predicational variables

Sorts and arities of variables:

sort : IndVar ⟼ Cn-V

arity : FunVar ⟼ Cn-V^{c*}
sort : FunVar ⟼ Cn-V

arity : PreVar ⟼ Cn-V^{c*}
sort : PreVar ⟼ {boo}

Sorted domains of variables:

IndVar.cn = {inv : IndVar | sort.inv = cn}
FunVar.cn = {fuv : FunVar | sort.fuv = cn}
PreVar = {prv : PreVar | sort.prv = boo}

Term- and formula-creating functions

mk-term-cn.inv = mk-term-cn(inv) for all inv : IndVar.cn and cn : Cn-V – {boo]
mk-formula.inv = mk-formula(inv) for all inv : IndVar.boo

Building Language-D for Language-V (5)
From AlgSyn-V to AlgSyn-D by their grammars

The abstract-syntax grammar of (the future) **AlgDen-D**

ter : Term.cn = for all cn : Cn-V – {boo}

➡ mk-term-cn(IndVar.cn) |

fn(Term.cn-1,…,Term.cn-n) | for all fn : Fn-V with
arity.fn = (cn-1,…,cn-n)
sort.fn = cn

➡ fuv(Term.cn-1,…,Term.cn-n)| for all fuv : FunVar with
arity.fuv = (cn-1,…,cn-n)
sort.fuv = cn

Building Language-D for Language-V (6)
From AlgSyn-V to AlgSyn-D

The abstract-syntax of (the future) **AlgDen-D**

for : Formula =

- pn(Term.cn-1,…,Term.cn-n) | for all pn : PreNam with arity.pn = (cn-1,…,cn-n)
- ➡ prv(Term.cn-1,…,Term.cn-n) | for all pv : PreVar with arity.pv = (cn-1,…,cn-n)
- and(Formula, Formula) |
- or(Formula, Formula) |
- implies(Formula, Formula) |
- not(Formula) |
- ➡ (∀i IndVar) Formula |
- ➡ (∃i IndVar) Formula |
- ➡ (∀f FunVar) Formula |
- ➡ (∃f FunVar) Formula |
- ➡ (∀p PreVar) Formula |
- ➡ (∃p PreVar) Formula

instead of Term.boo

AlgSyn-D is implicit in this grammar

Building Language-D for Language-V (7)
The common signature of AlgSyn-D and AlgDen-D

```
cn : Cn-D =
    {IndVar.cn | cn : Cn-V}    |  all carriers of individual variables
    Cn-V – {boo}               |  all former names except boo now replaced by formula
    {formula}

fn : Fn-D =
    {civ-cn-inv | cn : Cn-V, inv : IndVar.cn}  |  all names of individual-variable constructors
    {mk-term-cn | cn : Cn-V}                   |  the names of mk-term of all sorts cn
    Fn-V                                       |  all former names (including boo sort)
    FunVar                                     |  all functional variables
    PreVar                                     |  all predicational variables
    {mk-formula}                               |  the name of function mk-formula
    {and, or, implies, not, ∀i, ∃i, ∀f, ∃f, ∀p, ∃p}
```

Building Language-D for Language-V (8)
From AlgSyn-D to AlgDen-D

Valuations

uni : Universe = U{car.cn | cn : Cn-V}
vlu : IndValuation ⊆ IndVar ⟼ Universe
vlu : FunValuation ⊆ FunVar ⟼ {fun | fun : Universe^{c*} ⟼ Universe}
vlu : PreValuation ⊆ PreVar ⟼ {pre | pre : Universe^{c*} ⟼ Bool}
vlu : Valuation ⊆ IndValuation | FunValuation | PreValuation

Well-formed valuations

if inv : IndVar.cn
then vlu.inv : carDen-V.cn

if fuv : FunVar with arity.fuv = (cn-1,…,cn-n) and sort.fuv = cn
then vlu.fuv : carDen-V.cn-1 x … x carDen-V.cn-n ⟼ carDen-V.cn

if prv : PreVar with arity.pv = (cn-1,…,cn-n)
then vlu.prv : carDen-V.cn-1 x … x carDen-V.cn-n ⟼ carDen-V.boo

Domains of denotations

carDen-D.IndVar.cn = IndVar.cn for all cn : Cn-V,
carDen-D.cn = Valuation ⟼ carDen-V.cn for all cn : Cn-V,
carDen-D.formula = Valuation ⟼ carDen-V.boo

Building Language-D for Language-V (9)
Defining the interpretation funDen-D of functional symbols (1)

(1) Variable-creating functions – for every civ-cn-inv

funDen-D.civ-cn-inv : ⟼ IndVar.cn i.e.
funDen-D.civ-cn-inv.() = civ-cn-inv.()

(2) Term-making functions mk-term-cn:

funDen-D.mk-term-cn : IndVar.cn ⟼ carDen-D.cn i.e.
funDen-D.mk-term-cn : IndVar.cn ⟼ Valuation ⟼ carDen-V.cn for all cn : Cn-V
funDen-D.mk-term-cn.inv.vlu = vlu.inv

(3) Functional name fn : Fn-V with arity.fn = (cn-1,…,cn-n)
sort.fn = cn:

funDen-D.fn : carDen-D.cn-1 x …x carDen-D.cn-n ⟼ carDen-D.cn
funDen-D.fn.(den-1,…,den-n).vlu = funDen-V.fn.(den-1.vlu,…,den-n.vlu)

(4) Functional variable fuv : FunVar.cn with arity.fuv = (cn-1,…,cn-n)
sort.fuv = cn:

funDen-D.fuv : carDen-D.cn-1 x …x carDen-D.cn-n ⟼ carDen-D.cn
funDen-D.fuv.(den-1,…,den-n).vlu = vlu.fuv.(den-1.vlu,…,den-n.vlu)

Building Language-D for Language-V (10)
Defining the interpretation funDen-D of functional symbols (2)

(5) Formula-making function mk-formula

funDen-D.mk-formula : IndVar.boo ⟼ carDen-D.boo i.e.
funDen-D.mk-formula : IndVar.boo ⟼ Valuation ⟼ carDen-V.boo
funDen-D.mk-formula.inv.vlu = vlu.inv

(6), (7) The cases of the **names of predicates** and of **predicational variables** are analogous to (3) and (4).

(8) Conjunction

funDen-D.and : CarDen-D.formula x CarDen-D.formula ⟼ CarDen-D.formula
funDen-D.and.(den-1, den-2).vlu = funDen-V.and.(den-1.vlu, den-2.vlu)

interpretation from Language-V

(9) General quantifier

funDen-D.$\forall i$: IndVar x CarDen-D.formula ⟼ CarDen-D.formula
funDen-D.$\forall i$.(inv, den).vlu =
  for any uni : Universe, den.(vlu[inv/uni]) = tt ➔ tt
  true ➔ ff

Building Language-D for Peano Arithmetics (1)

A **standard model** of a 0th-order Peano Arithmetic (a **Language-V** level)

constants are printed in green

Carriers

nat : Natural = {0, 1,…}
boo : Bool = {tt, ff}

Cn = {nat, boo}
Fn = {zer, suc, num, equ}

Functions

zer :		⟼ Decimal	arity.zer = (),	sort.zer = nat	
suc : Decimal		⟼ Decimal	arity.suc = (nat),	sort.zer = nat	
num : Decimal		⟼ Bool	arity.num = (nat),	sort.num = boo	
equ : Decimal x Decimal		⟼ Bool	arity.equ = (nat, nat),	sort.equ = boo	

Grammar

ter-V : Term-V = ground terms, e.g.: zer(), suc(zer())
zer() | suc(Term-V)

for-V : Form-V = ground formulas, e.g.: num(zer()), equ(zer(), suc(zer()))
num(Term-V) | equ(Term-V, Term-V)

carDen.nat = Natural term denotations
carDen.boo = Bool formula denotations

Building Language-D for Peano Arithmetics (2)

A 2nd-order Peano Arithmetic - syntax (a **Lingua-D** level)

IndVar.nat = {x, y, z} individual decimal variables; sort.x = nat
IndVar.boo = {a, b, c} individual boolean variables; sort.a = boo
PreVar = {P} one predicational variable; sort.P = boo arity.P = (nat)

Grammar

```
inv-D : IndVar.nat =
        x | y | z
inv-D : IndVar.boo =
        a | b | c
ter-D : Term-D =
        mk-term(IndVar.nat) | zer() | suc(Term-D)
for-D : Form-D =
        mk-formula(IndVar.boo) | num(Term-D) | P(Term-D) |
        equ(Term-D, Term-D) | and(Formula-D , Formula-D) | …
```

no syntactic cathegory of 2nd-order variables!

variables are printed in black

a formula with 2nd-order variable

Building Language-D for Peano Arithmetics (3)

A 2nd-order Peano Arithmetic - denotations (a **Lingua-D** level)

```
vlu : Valuation   =
      ({x, y, z}  ⟼ Natural)          |
      ({a, b, c}  ⟼ Bool)             |
      {P}         ⟼ (Natural ⟼ Bool)
```

The domains of denotations:

```
carDen-D.IndVar.nat   = IndVar.nat    the denotations of variables are variables
carDen-D.IndVar.boo   = IndVar.boo

carDen-D.nat          = Valuation ⟼ Natural   the denotations of terms
carDen-D.boo          = Valuation ⟼ Bool      the denotations of formulas
```

An example of function's interpretation

```
funDen-D.suc : TerDen ⟼ TerDen                          i.e.
funDen-D.suc : TerDen ⟼ Valuation ⟼ Natural
funDen-D.suc.ted.vlu = funDen-V.suc.(ted.vlu)
```

Building a language of a formalized theory of the denotations of Lingua-V

Formalizing Lingua-V (1)
syntax – new categories

To the grammar of **Lingua,** we add the following categories:

con	: Con-V	— conditions
asr	: Asr-V	— assertions
sin	: SpeIns-V	— specified instructions
sde	: SpeDec-V	— specified declarations
sct	: SpeClaTra-V	— specified class transformations
spp	: SpeProPre-V	— specified program preambles
spr	: SpePro-V	— specified programs
mco	: MetCon-V	— metaconditions

Formalizing Lingua-V (2)
syntax – conditions

The syntax of conditions is similar to that of value expressions with boolean values, with two exceptions:

- new predicates not available for value expressions,
- Kleene's logical connectives instead of McCarthy's.

```
con : Con-V =
    duplicates of atomic boolean expressions of Lingua
      con-equal-int(ValExp , ValExp)            |   equal integers
      …
    conditions with predicates that are not available for value expressions
      con-is-typ(Identifier, TypExp)            |   identifier declared as a type constant
      con-proc-opened(Identifier, Identifier)   |   opened procedure
      …
    algorithmic conditions
      con-left-algorithmic(SpePro-V , Con-V)    |   in concrete syn. SpePro-V@Con-V
      con-right-algorithmic(Con-V , SpePro)     |   in concrete syn. Con-V@SpePro-V
      …
    compound conditions with Kleene's operators
     con-or-k(Con-V, Con-V)
```

Formalizing Lingua-V (3)
syntax – metaconditions (1)

mco : MetCon-V =

relational metaconditions

mco-stronger(Con-V , Con-V)	in concrete syntax ⇨
mco-weak-equivalent(Con-V , Con-V)	in concrete syntax ⇔
mco-less-defined(Con-V , Con-V)	in concrete syntax ⊑
mco-strong-equivalent(Con-V , Con-V)	in concrete syntax ≡

behavioral metaconditions

mco-insures-LR(Con-V , Ins)	
mco-resilient(Con-V , SpePro)	
…	

temporal metaconditions

mco-primary(Con-V , MetPro-V)	MetPro-V — metaprograms
mco-induced(Con-V , MetPro-V)	
…	

language related metaconditions

mco-immunizing(Con-V)	
mco-immanent(Con-V)	
…	

Formalizing Lingua-V (4)
syntax – metaconditions (2)

mco : MetCon-V = (cont.)

metaprograms

mco-metaprogram(Con-V, SpePro-V, Con-V) |

compound metaconditions with classical operators

mco-and(MetCon-V , MetCon-V) |

mco-or(MetCon-V , MetCon-V) |

mco-implies(MetCon-V , MetCon-V) |

mco-not(MetCon-V)

Formalizing Lingua-V (5)
denotations

New carriers

cod : ConDen-V = State → BoolE
asd : AsrDen-V = State → BoolE
sid : SpeInsDen-V = State → State
sct : SpeClaTraDen-V = State → State
spd : SpeProPreDen-V = State → State
spd : SpeProDen-V = State → State
mcd : MetConDen-V = {tt, ff}

Signatures of constructors (examples)

cod-equal-int : ValExpDen-V x ValExpDen-V ⟼ ConDen-V
cod-less-int : ValExpDen-V x ValExpDen-V ⟼ ConDen-V
…
cod-is-typ : Identifier x TypExpDen-V ⟼ ConDen-V
cod-var-is-typ : Identifier x TypExpDen-V ⟼ ConDen-V
cod-proc-opened : Identifier x Identifier ⟼ ConDen-V
…
con-left-algorithmic : SpeProDen-V x ConDen-V ⟼ ConDen-V
con-right-algorithmic : ConDen-V x SpeProDen-V ⟼ ConDen-V

Lifting Lingua-V to Lingua-D (1)
syntax - variables

Individual variables – for all cn : Cn-V

inv : IdeVar-D	= ide	| ide-1	|…	variables corresponding to identifies,
inv : ValExpVar-D	= vex	| vex-1	|…	variables corresponding to val. expressions,
inv : RefExpVar-D	= rex	| rex-1	|…	variables corresponding to ref.-expressions,
inv : SpeInsVar-D	= sin	| sin-1	|…	variables corresponding to specinstructions,
inv : ConVar-D	= con	| con-1	|…	variables corresponding to conditions,
…				
inv : MetConVal-D	= mec	| mec-1	| …	variables corresponding to metaconditions.

Note the difference:
vex – a metavariable running over domain ValExp-D
vex – an individual variable of sort „value expression”; an element of ValExpVar-D

Second-order variables (vacate – so far)

We shall need second-order axioms (at least) for proving termination properties of programs.

Lifting Lingua-V to Lingua-D (2)
syntax – terms (1)

value expressions (to the grammar of Lingua-V we add the first clause below)

```
vex : ValExp-D =
  vex-make-vex(ValExpVar-D)                    |  single-variable term
  vex-bo(BooleanSyn-D)                         |
  vex-in(IntegerSyn-D)                         |
  vex-re(RealSyn-D)                            |
  vex-te(TextSyn-D)                            |
  vex-variable(Ide-D)                          |  single-identifier value-expression
  vex-attribute(ValExp-D , Ide-D)              |
  vex-call-fun-pro(Ide-D, Ide-D, ActPar-D)     |
  vex-add-int(ValExp-D , ValExp-D)             |
  vex-less-int(ValExp-D , ValExp-D)            |
  vex-or-m(ValExp-D , ValExp-D)                |  McCarthy's alternative
  vex-create-li(ValExp-D)                      |
  vex-get-from-rc(ValExp-D , Ide-D)            |
  ...
```

Lifting Lingua-V to Lingua-D (3)
syntax - terms (2)

specinstructions (we add the first clause)

```
sin : SpeIns-D =
        sin-make-sin(SpeInsVar-D)                                  | single-variable term
        sin-make-asr(Con-D)                                        | assertions
        sin-skip-ins()                                             |
        sin-assign(RefExp-D , ValExp-D)                            |
        sin-call-imp-pro(Ide-D , Ide-D , ActPar-D , ActPar-D)      |
        sin-call-obj-con(Ide-D , Ide-D , ActPar-D)                 |
        sin-if(ValExp-D , SpeIns-D , SpeIns-D)                     |
        sin-if-error(ValExp-D , SpeIns-D)                          |
        sin-while(ValExp-D , SpeIns-D)                             |
        sin-compose-ins(Ins-D , Ins-D)
```

identifiers

```
ide : Ide-D =
        IdeVar-D   |
        Identifier
```

Lifting Lingua-V to Lingua-D (4)
syntax – terms (3)

conditions

```
con : Con-D =
        con-make-con(ConVar-D)                  |  single-variable term
        con-or-k(Con-D , Con-D)                 |  Kleene's alternative
        con-less-int(ValExp-D , ValExp-D)       |
        con-is-value(ValExp-D)                  |
        con-is-free(Ide-D)                      |
        con-left-algorithmic(SpeIns-D , Con-D)  |
        con-right-algorithmic(Con-D , SpePro-D) |
        ...
```

examples of ground terms (in abstract syntax)

```
sin-assign(x, vex-divide-re(1, z))
vex-less(y, 0)
sin-while(vex-less-int(x, 0), sin-assign(x, vex-add-int(a, 1)))
sin-skip-ins
```

examples of free terms

```
sin-assign(rex, vex-divide-re(vex-1, vex-2))
vex-less(vex, 0)
sin-while(vex-less-int(vex-1, vex-2), sin-assign(rex, vex-add-int(vex-3, 1)))
```

Lifting Lingua-V to Lingua-D (5)
syntax – formulas (metaconditions)

```
mec : MetCon-D =
  mec-make-mec(MetConVar-D)                          |  single-variable formula
  mec-stronger(Con-D ,  Con-D)                       |  in concrete syntax ⇨
  mec-weakly-equivalent(Con-D , Con-D)               |  in concrete syntax ⇔
  mec-less-defined(Con-D , Con-D)                    |  in concrete syntax ⊑
  mec-strongly-equivalent(Con-D , Con-D)             |  in concrete syntax ≡
  mec-insures LR(Con-D , SpeIns-D)                   |
  mec-hereditary(Con-D , MetPro-D)                   |
  mec-immunizing(Con-D)                              |
  mec-metaprogram(Con-D , SpePro , Con-D)            |
  …
  mec-and(MetCon-D , MetCon-D)                       |  classical connectives
  mec-or(MetCon-D , MetCon-D)                        |
  mec-implies(MetCon-D , MetCon-D)                   |
  mec-not(MetCon-D
```

Lifting Lingua-V to Lingua-D (6)
syntax – examples of formulas (in colloquial syntax)

ground formulas in **Lingua-D** i.e., metaformulas in **Lingua-V**:

$\sqrt[2]{x} > 2 \Leftrightarrow x > 4$

```
pre nni(x, k) and-k n+1 ≤ M:
   x := 0;
   while x+1 ≤ n do x := x+1 od
post x = n
```

free formulas in **Lingua-D**:

```
pre sin @ con
   sin
post con
```

pre prc-1**:** spr-1 **post** poc-1
pre prc-2**:** spr-2 **post** poc-2
poc-1 ⇨ prc-2

---

pre prc-1**:** spr-1**;** spr-2 **post** poc-2
pre prc-1**:** spr-1**; asr** poc-1 **rsa;** spr-2 **post** poc-2
pre prc-1**:** spr-1**; asr** prc-2 **rsa;** spr-2 **post** poc-2

(**pre** prc-1**:** spr-1 **post** poc-1) **and pre** … **post and pre** … **post**) **implies**
(**pre** … **post and pre** … **post and pre** … **post**)

Lifting Lingua-V to Lingua-D (7)
denotations – valuations

uni : Universe = Identifier | TypExpDen-V | RefExpDen-V | ValExpDen-V | …
vlu : IndValuation ⊆ IndVar⟼ Universe
vlu : FunValuation ⊆ FunVar ⟼ {fun | fun : Universe^{c*} ⟼ Universe}
vlu : PreValuation ⊆ PreVar ⟼ {pre | pre : Universe^{c*} ⟼ Bool}
vlu : Valuation ⊆ IndValuation | FunValuation | PreValuation

Every valuation vlu is sort-wise well-formed, e.g.:

if vex : ValExpVar-D i.e., vex stands for an individual variable of sort
then vlu.vex : ValExpDen-V

e.g.,
vlu.vex-1 : ValExpDen-V here vex-1 is a concrete individual variable

Lifting Lingua-V to Lingua-D (8)
denotations – examples of carriers

ved : ValExpDen-D = Valuation ⟼ ValExpDen-V
red : RefExpDen-D = Valuation ⟼ RefExpDen-V
ind : InsDen-D = Valuation ⟼ InsDen-V
spd : SpeProDen-D = Valuation ⟼ SpeProDen-V
ide : IdeDen-D = Ide-D
cod : ConDen-D = Valuation ⟼ ConDen-V
…
mcd : MetConDen-D = Valuation ⟼ MetConDen-V

Lifting Lingua-V to Lingua-D (9)
denotations – examples of constructors

ind-assign-ft : RefExpDen-D x ValExpDen-D ⟼ InsDen-D i.e.
ind-assign-ft : RefExpDen-D x ValExpDen-D ⟼ Valuation ⟼ InsDen-V
ind-assign-ft.(red, ved).vlu = ind-assign-v.(red.vlu, ved.vlu)

a constructor of Lingua-V

mcd-metaprogram-ft : ConDen-D x SpeProDen-D x ConDen-D
⟼ MetConDen-D i.e.
mcd-metaprogram-ft : ConDen-D x SpeProDen-D x ConDen-D
⟼ Valuation ⟼ {tt, ff}
mcd-metaprogram-ft.(cod-1, spd, cod-2).val =
mcd-stronger.(cod-1.val, con-left-algorithmic.(spd.val, con-2.val))

The same definition written in the colloquial metanotation:

mcd-metaprogram-ft.(cod-1, spd, cod-2).val = cod-1.val ⇨ (spd.val)@(con-2.val)

Note: here we are using not-underlined metavariables.

Building a formalized theory of the denotations of Lingua-V

Our ultimate goal

Given a denotational model of a language **Lingua-V** of validating programming:

sem : AlgSyn-V ⟼ AlgDen-V

build a formalized **D-theory**:

- whose **Lingua-D** includes **Lingua-V**
- AlgDen-V is „included” in a model of D-theory

Two theories to be used:

M-theory – formal but not formalized, based on **MetaSoft**

D-theory – formalized, based on **Lingua-D**

The structure of a formalized theory **D-theory** of the denotations of **Lingua-V**

1. **Language**: **Lingua-D**
2. **Axioms**:
   a. axioms on abstract mathematical entities: numbers, sets, functions, etc.
   b. axioms on **Lingua-V** values: integer values, objects, etc.
   c. axioms on **Lingua-V** denotations: of expressions, instructions, metaconditions, etc.
3. **Inference rules**:
   a. universal rules: substitution, detachment, generalization, etc.; significant
   b. standard rules – expressible as axioms; useful
   c. non-standard rules – not expressible as axioms; useful + significant (?)

we shall concentrate on group 2.c

1st-order axioms independent of **Lingua-V**
(examples)

Dependencies between metapredicates

(con1 **≡** con2**)** **iff** **((**con1 **⊑** con2**)** **and** **(**con2 **⊑** con1**))**
(con1 **⇔** con2**)** **iff** **((**con1 **⇨** con2**)** **and** **(**con2 **⇨** con1**))**
(con1 **≡** con2**)** **implies** **(**con1 **⇔** con2**)**

Relations **≡** and **⇔** are equivalences in the set of conditions

con **≡** con — reflexivity
(con1 **≡** con2**)** **implies** **(**con2 **≡** con1**)** — symmetry

De Morgan's laws

not (con1 **and-k** con2**)** **≡** **(not(**con2**) or-k not(**con1**))**
not (con1 **and-k** con2**)** **⇔** **(not(**con2**) or-k not(**con1**))**
…

The nearly-true condition **NT**
error-transparent(con**) implies (**con **⇨ NT)**

A comparison of three implications
(error-sensitive(con1**) and (**con1 **implies-k** con2**) ≡ NT)) implies (**con1 **⇨** con2**))**

1st-order axioms Lingua-V dependent
examples of program-construction rules (1)

Rule for variable declaration

pre (ide **is free) and-k (**tex **is type)**
  let ide **be** tex **tel**
post var ide **is** tex

Rule for an abstract attribute declaration

pre (ide-1 **is free) and-k (**ide-2 **is class) and-k (**tex **is type) :**
  let ide-1 **be** tex **with** yex **as** pst **tel in** ide-2
post att ide-1 **is** tex **with** yex **in** ide-2 **as** pst

Rule for a type constant declaration

pre (ide-1 **is free) and-k (**ide-2 **is class) and-k (**tex **is type) :**
  set ide-1 **be** tex **tes in** ide-2
post ide-1 **is** tex

These rules are used in metaprogram constructions by the application of the inference rule of substitution.

1st-order axioms Lingua-V dependent
examples of program-construction rules (2)

Rules about temporal metaconditions

not(ide **is free) hereditary in** mpr
(ide **is free) co hereditary in** mpr
(ide **is** tex**) hereditary in** mpr

Rule for conditional branching

pre (prc **and-k** vex) **:** sin1 **post** poc
pre (prc **and-k** (**not-k** vex)) **:** sin2 **post** poc
prc ⇨ (vex **or-k** (**not-k** vex))

---

pre prc : **if** vex **then** sin1 **else** sin2 **fi post** poc

Inference rules (1)

Program-building rules versus inference rules

pre prc **:** spr **post** poc
prc1 ⇨ prc
---
pre prc1 : spr **post** poc

A program-building rule in **M-theory**

((pre prc : spr **post** poc**) and (**prc1 ⇨ prc**)) implies (pre** prc1 : spr **post** poc**)**

(1) **((pre** prc : spr **post** poc**) and (**prc1 ⇨ prc**)) implies (pre** prc1 : spr **post** poc**)**
(2) mec1 **implies (**mec2 **implies (**mec1 **and** mec2**))** — a tautology
assume for concrete prc, spr,…
(3) **pre** prc **:** spr **post** poc
(4) prc1 ⇨ prc

Axioms in **D-theory**

By substitution of (3) and (4) to (2) and by detachment we can deduce

|- (**pre** prc : spr **post** poc) and |- (prc-1 ⇨ prc) implies |- (**pre** prc-1 : spr **post** poc)

|- pre prc **:** spr **post** poc
|- prc1 ⇨ prc

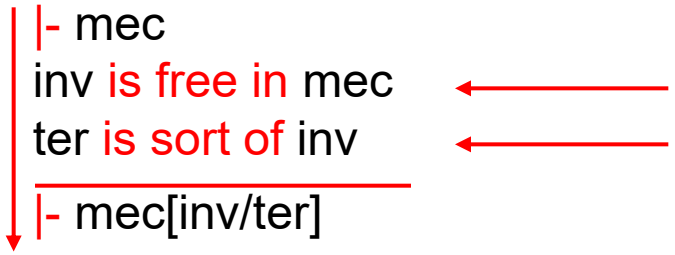
|- pre prc1 : spr **post** poc

An inference rule in **D-theory**

A formula in **M-theory**

Inference rules (2)

Universal inference rules: the substitution rule



They are not provable lemmas

Examples of substitutions:

pre sin **@** con **:** sin **post** con

pre ide := vex **@** con **:** ide := vex **post** con

pre x := x+1 **@** x > 0 **:** x := x+1 **post** x > 0

ide := vex → sin

x → ide
1 → vex

Here we can't replace
x by z
(unless by generating a new program)

How axioms induce standard inference rules (1)

Metatheorem 1

If mec1 and mec2 are metaconditions in **Lingua-D** such that

|- mec1 **implies** mec2

in other words, if

$$\frac{\text{mec1}}{\text{mec2}}$$

is in the repository of lemmas, then then the following inference rule is sound:

$$\frac{\text{|- mec1}}{\text{|- mec2}}$$

i.e., may be included in the repository of rules.

Note: we use here not-underlined metavariables.

How axioms induce standard inference rules (2)

Metatheorem 2

If

mec-1 **and**

…

mec-n

---

mec

is in repository, then then the following inference rule is sound:

|- mec-1

…

|- mec-2

|- mec

Example

Since

pre prc : spr **post** poc **and**

prc-1 ⇨ prc

---

pre prc : spr **post** poc

is in repository, the following inference rule is sound:

|- **pre** prc : spr **post** poc

|- prc-1 ⇨ prc

|- **pre** prc1 : spr **post** poc

Non-standard inference rules (1)

general observations

The creation of standard inference rules:

program-construction rule → metacondition → inference rule

The creation of non-standard inference rules:

program-construction rule → inference rule

Non-standard inference rules (2)

assignment-instruction rules

A general standard rule

```
pre sin @ con :
      sin
post con
```

An assignment standard rule

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

An assignment non-standard construction rule (1):

type-compatible(ide, vex**) and-k** con[ide/vex] ⇨ **(**ide := vex **@** con**)**

An assignment non-standard construction rule (2):

```
pre type-compatible(ide, vex) and-k con[ide/vex] :
      ide := vex
post con
```

An assignment non-standard inference rule:

```
|- true
-------------------------------------------------
|- pre type-compatible(ide, vex) and-k con[ide/vex] :
     ide := vex
   post con
```

Non-standard inference rules (3)

examples

The removal of an assertion:

```
|- pre prc : head ; asr con rsa ; tail post poc
|- error-sensitive(poc)
-------------------------------------------------
|- pre prc : head ; tail post poc
```

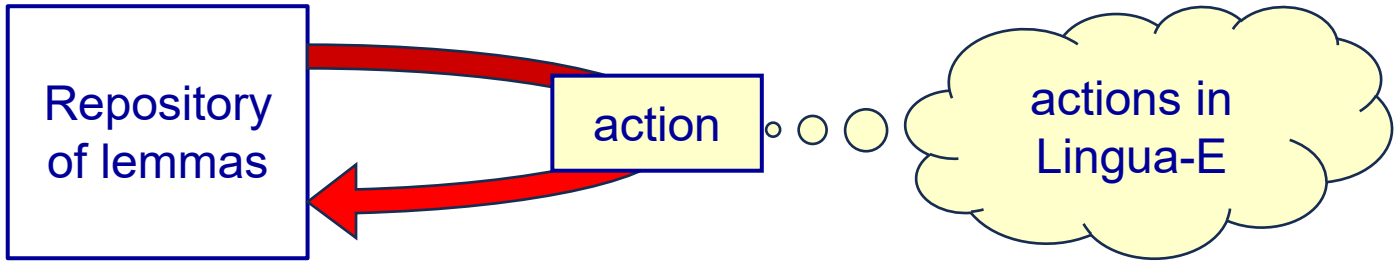
The replacement of a condition in an assertion by a weakly equivalent one

```
|- pre prc : head ; asr con1 rsa ; tail post poc
|- con1 ⇔ con2
|- error-sensitive(poc)
-------------------------------------------------
|- pre prc : head ; asr con2 rsa ; tail post poc
```

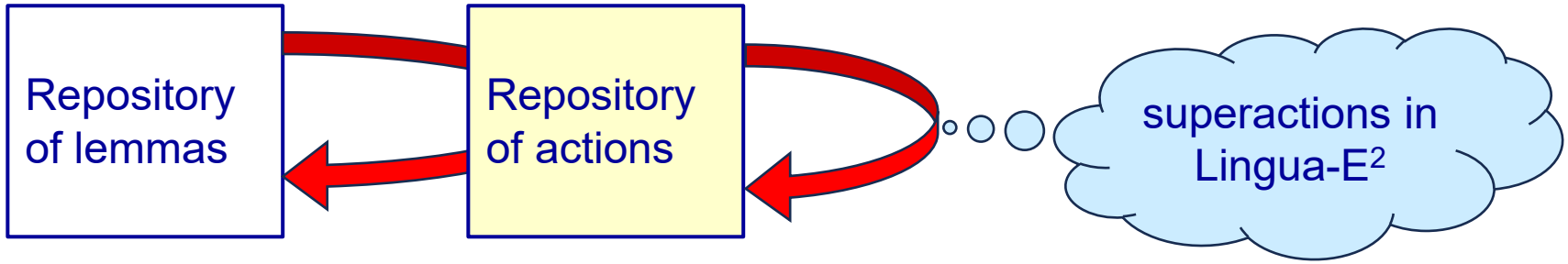
Building a denotational model of an ecosystem for programmers

Primary and secondary ecosystems

Primary ecosystems



Secondary ecosystems



Denotational models of ecosystems: **Lingua-E**

An ecosystem as a language with denotational semantics

Introductory domains

car : Character = {a, b, c, …, A, B, C,…, 0, 1,…,9, (,), …}
nam : Name = Character^{c+} the names of metaconditions
rep : Repository = Name ⟹ (ValMetCon-D | Con-V)

valid metaconditions and conditions

The domains of the algebra of denotations of **Lingua-E**:

tex : Text = Character^{c+}
nam : Name = Text
svd : SubVecDen = IndVal-D ⟹ Text the denotations of substitution vectors
acd : ActDen = Repository ⟼ Repository | Error the denotations of actions
inv : IndVar-D = IdeVar-D | ValExpVar-D | RefExpVar-D… individual var.
ide : IdeVar-D = ide | {ide-tex| tex : Text} ‚ide' – variables with indices
vex : ValExpVar-D = vex | {vex-tex | tex : Text}
rex : RefExpVar-D = rex | {rex-tex | tex : Text}
sin : SpeInsVar-D = sin | {sin-tex | tex : Text}
con : ConVar-D = con | prc | poc |
| {con-tex | tex : Text} |
| {prc-tex | tex : Text} |
| {poc-tex | tex : Text}

So far, we do not introduce second-order variables.

Denotational models of ecosystems
Constructors of substitution denotations

Auxiliary functions:

free : IndVar-D x MetCon-D ⟼ {tt, ff} variable is free in metacondition

matching : IndVar-D x Text ⟼ {'OK'} | Error the sort of a variable matches the sort of text

(an intelligent **Lingua-D**-oriented editor)

Constructors of the denotations of substitution vectors:

create-sub : IndVar-D x Text ⟼ SubVecDen
create-sub(inv, tex) =
  not matching(inv, tex) ➔ 'matching not satisfied'
  true ➔ [inv/tex]

expand-sub : SubVecDen x IndVar-D x Text ⟼ SubVecDen
expand-sub.(sub, inv, tex) =
  sub.inv = ! ➔ 'variable already assigned'
  not matching(inv, tex) ➔ 'matching not satisfied'
  true ➔ sub[inv/ter]

Actions
Their taxonomy

- Universal actions – derived from universal inference-rules.
- Standard actions – derived from axioms (program-construction rules).
- Non-standard actions – added as new inference-rules (non-derivable from axioms.
- Proving action – activating theorem-prover

Basic actions

Actions
Inference-rule actions – substitution

Auxiliary functions:

swap : IndVar-D x Text ⟼ MetCon-D ⟼ MetCon-D | Error — single swap
replace : SubVecDen ⟼ MetCon-D ⟼ MetCon-D | Error — many swaps

substitute : Name x SubVecDen x Name ⟼ Action i.e.
substitute : Name x SubVecDen x Name ⟼ Repository ⟼ Repository | Error
sub-gro.(nam-s, svd, nam-t).rep = -s – source, -t – target
  rep.nam-s = ? ➔ 'no source metacondition'
  rep.nam-t = ! ➔ 'target name already assigned'
  let
    mec-s = rep.nam-s
    mec-t = replace.svd.mec-s
  mec-t : Error ➔ mec-t
  true ➔ rep[nam-t/mec-t]

Rule of substitution

|- mec
inv is free in mec
ter is of sort of inv

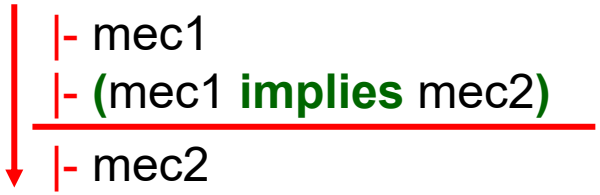
|- mec[inv/ter]

they are not lemmas!

Due to the rule of substitution, constructor substitute is sound.

Actions
Inference-rule actions – detachment

Rule of detachment



detach : Name x Name x Name ⟼ ActDen
detach : Name x Name x Name ⟼ Repository ⟼ Repository | Error

Due to the rule of detachment,
constructor detach is sound.

Standard actions

Strengthening preconditions

|- **pre** prc **:** spr **post** poc
|- prc1 ⇨ prc

|- **pre** prc1 **:** spr **post** poc

strengthen-pre : Name x Name x Name ⟼ Repository ⟼ Repository | Error

pre prc **:** spr **post** poc

prc1 ⇨ prc

pre prc1 **:** spr **post** poc

Actions
Standard actions – strengthening precondition

strengthen-pre : Name x Name x Name ⟼ Action
strengthen-pre : Name x Name x Name ⟼ Repository ⟼ Repository | Error
strengthen-pre.(nam-m, nam-s, nam-t).rep = -m – metapr., -s – stronger, -t – target

rep.nam-m = ?	➔ 'no prerequisite metaprogram'
rep.nam-s = ?	➔ 'no stronger-than metacondition'
rep.nam-t = !	➔ 'target name already assigned'
not is-metaprogram.(rep.nam-m)	➔ 'metaprogram expected'
root.(rep.nam-s) ≠ **stronger**	➔ 'a stronger-than metacondition expected'
let	
pre prc **:** spr **post** poc = rep.nam-m	
stronger(prc1, con**)** = rep.nam-s	
con ≠ prc	➔ 'conclusion not adequate'
let	
new-mec = **pre** prc1 **:** spr **post** poc	
true	➔ rep[nam-t/new-mec).rep

parsing techniques required

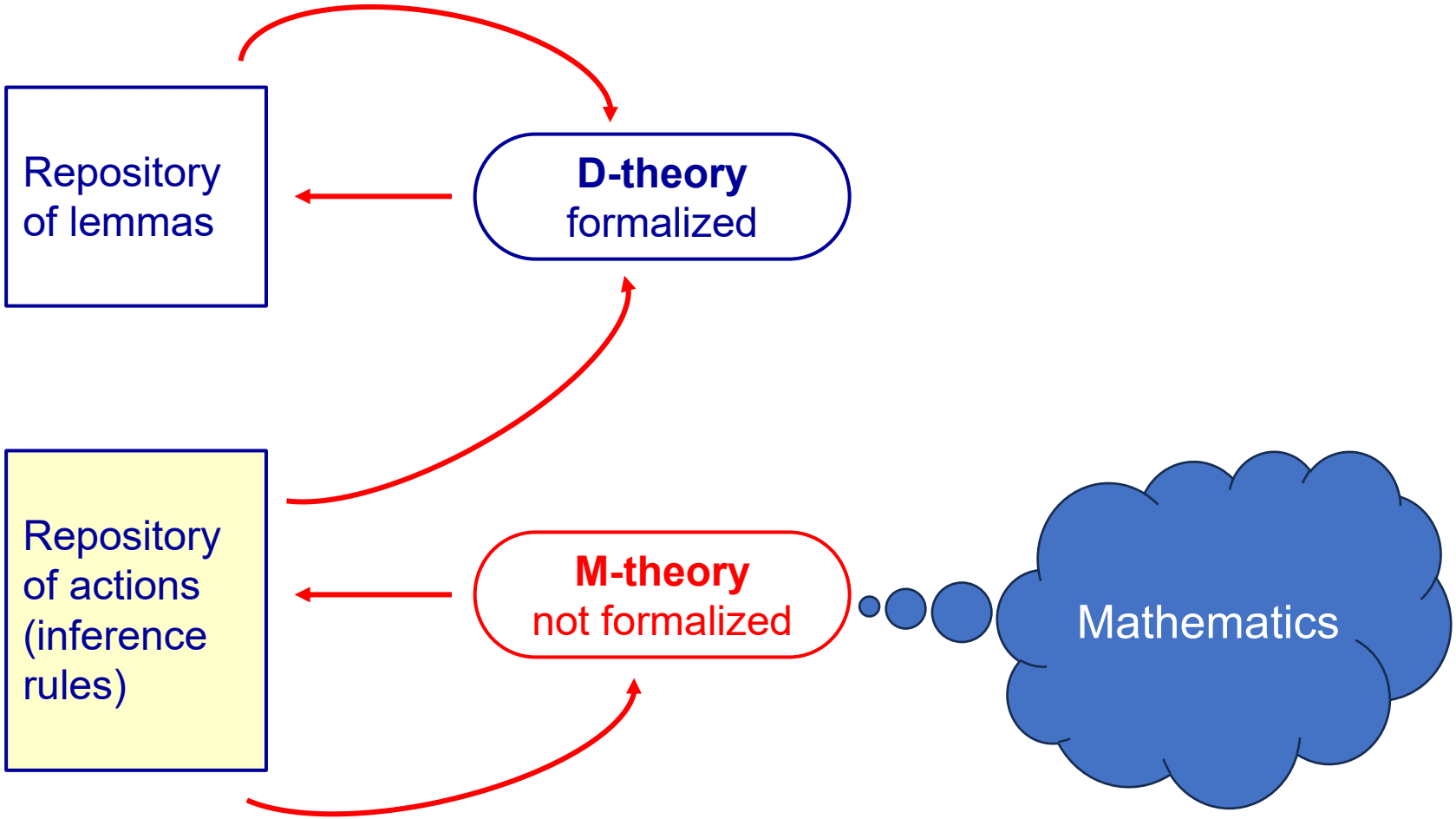
Actions
Non-standard actions – prover-activation action

prove : MetCon-D x Name ⟼ Repository → Repository | Error
prove.(mec, nam).rep =
  valid.mec = ? ➔ ?
  valid.mec = 'NO' ➔ 'metacondition not valid'
  true ➔ rep[nam/mec]

This partial function represents a theorem prover

valid : MetCon-D → {YES, NO] a partial function!

A hybrid scenario of the development of prime repositories




Thank you for
your attention