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

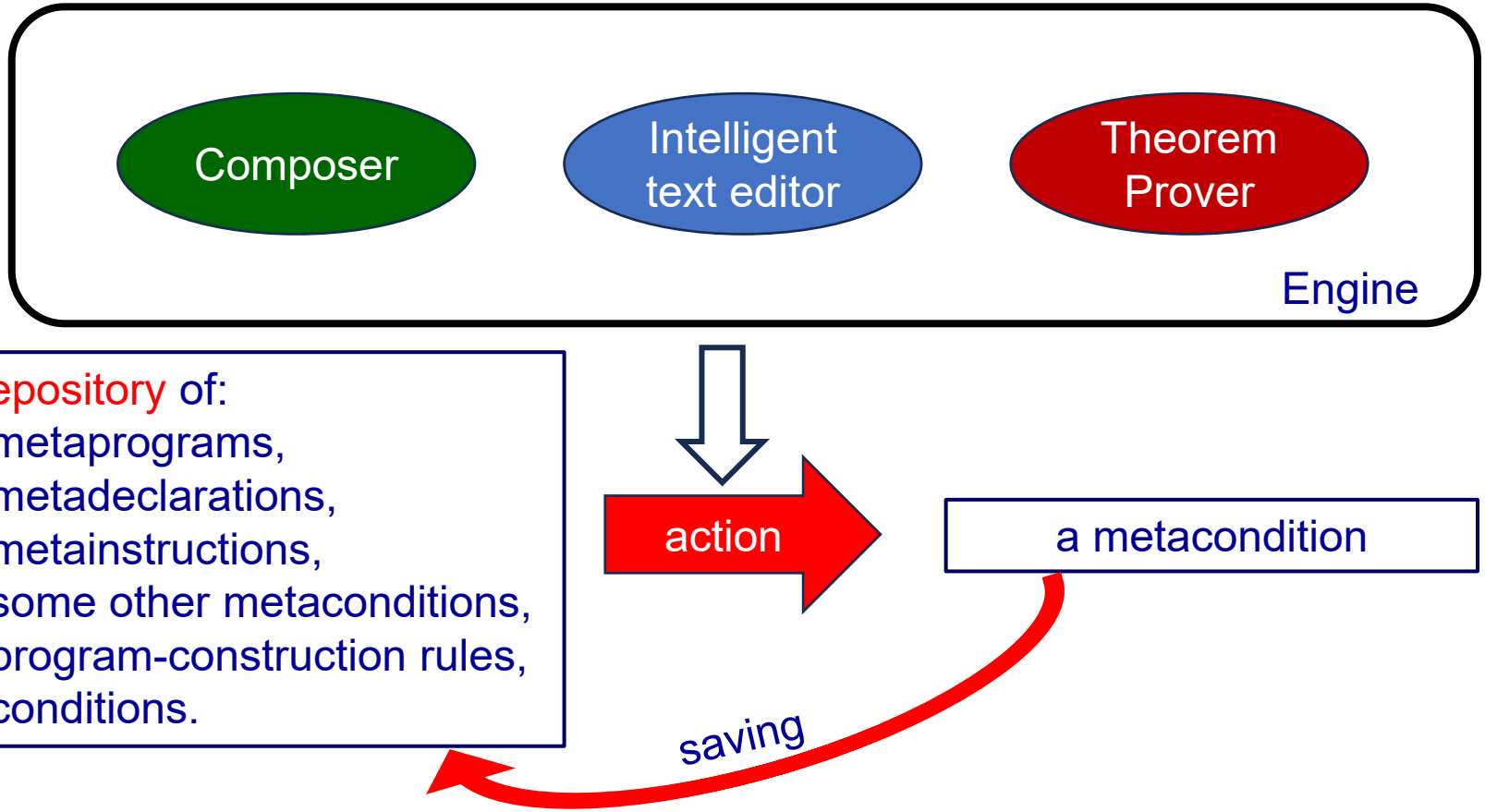
Andrzej Jacek Blikle

October 4th, 2025

(short version)

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:

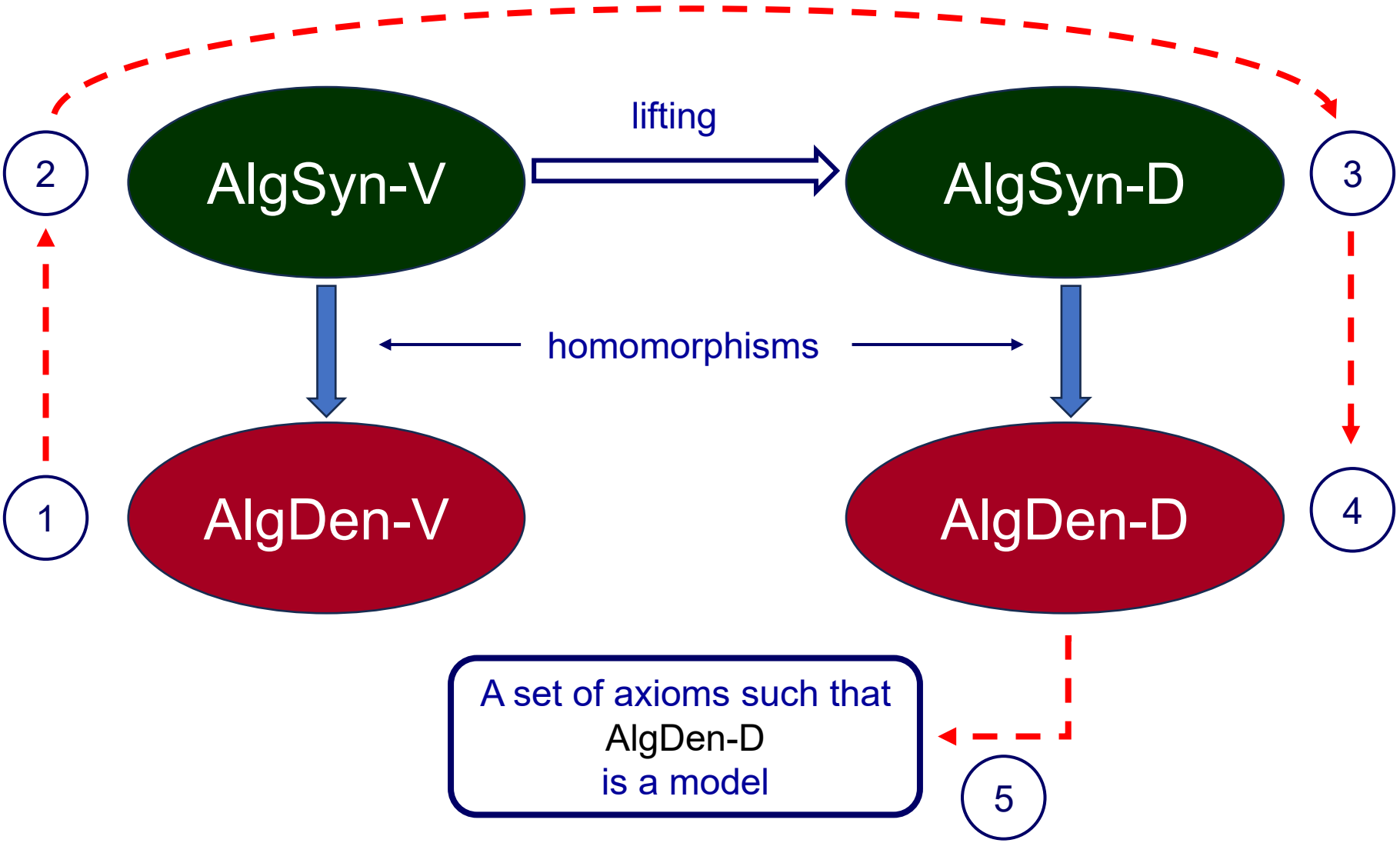
Lingua — a source programming language,
Lingua-V = **Lingua** + conditions + metaconditions
Lingua-D = **Lingua-V +** variables
Lingua-E — **Lingua-DA**-based language of ecosystem

running over denotations of **Lingua-V**

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

Building Language-D for Language-V (1)

D – denotations



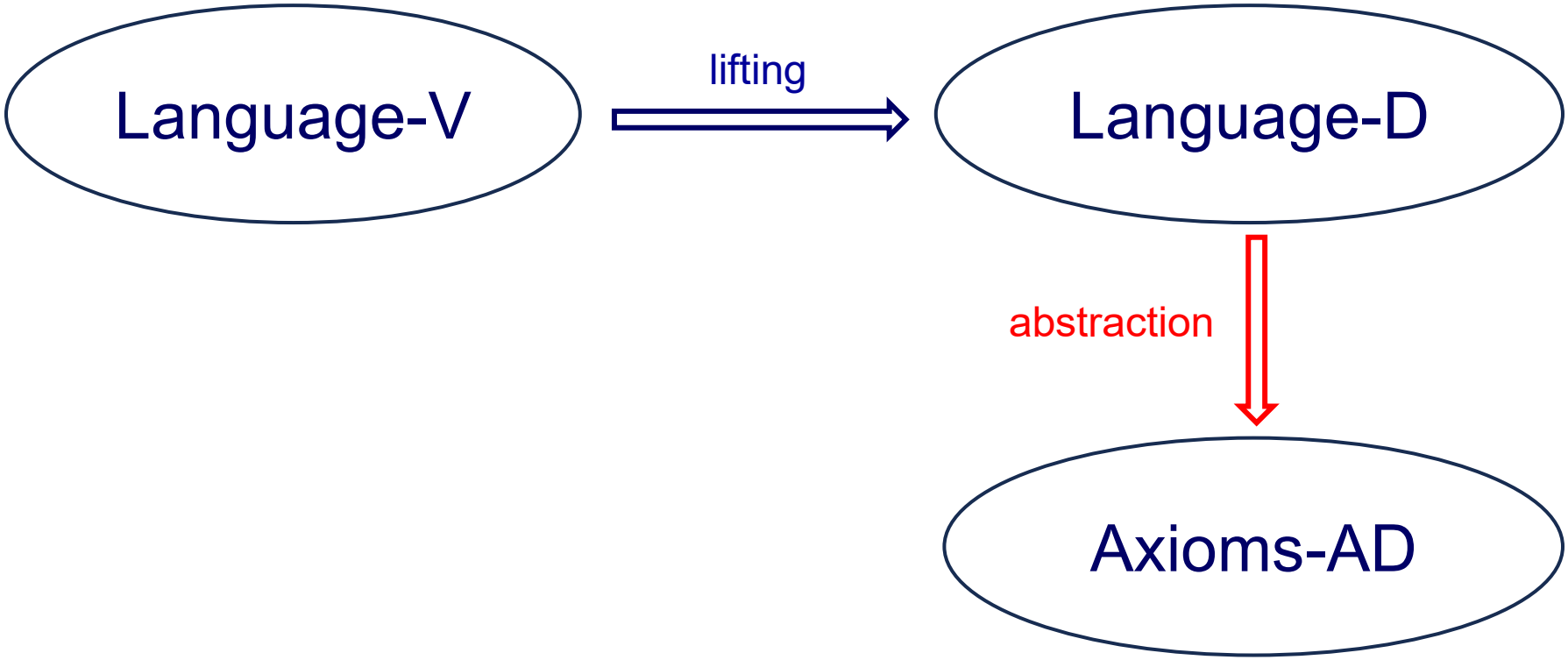
Lifting Language-V to Language-D (2)

How theories and models are built

mathematical logic: axioms ➔ its models

working mathematician: a model ➔ its axioms

our case



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)            |
  ...
```

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.

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**))** – 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

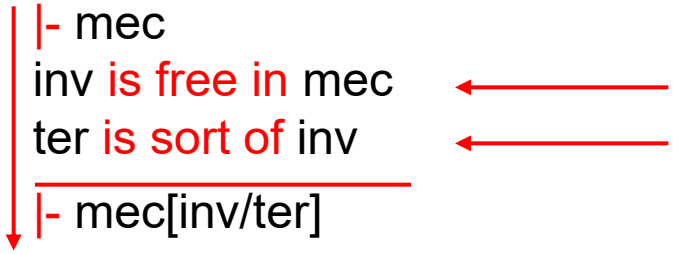
|- pre prc1 : spr **post** poc

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 (due to detachment) 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
```

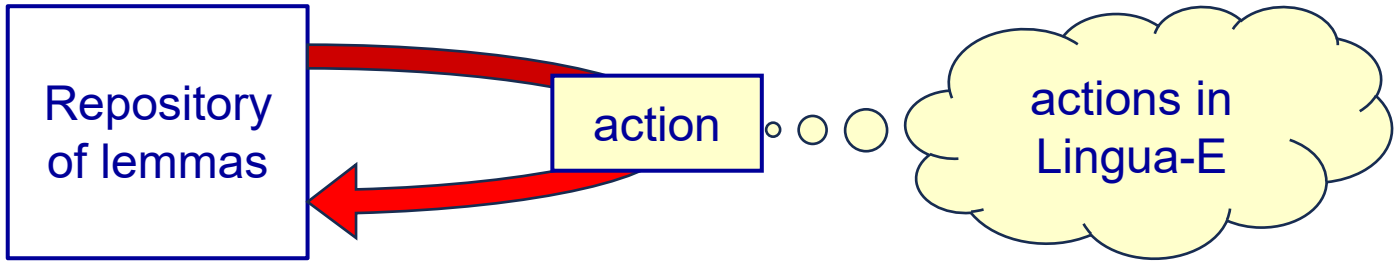
The replacement of a condition in an assertion by a weakly equivalent one

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

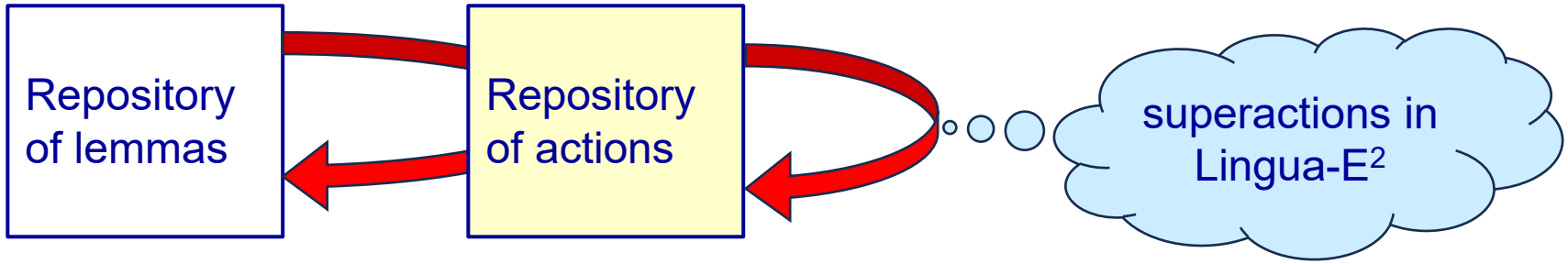
Building a denotational model of an ecosystem for programmers

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.

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

Universal 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

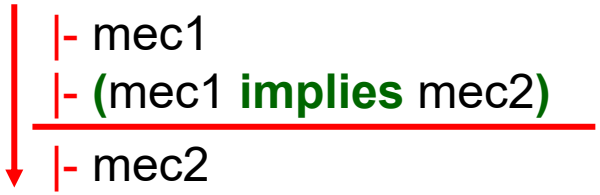
|- mec[inv/ter]

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

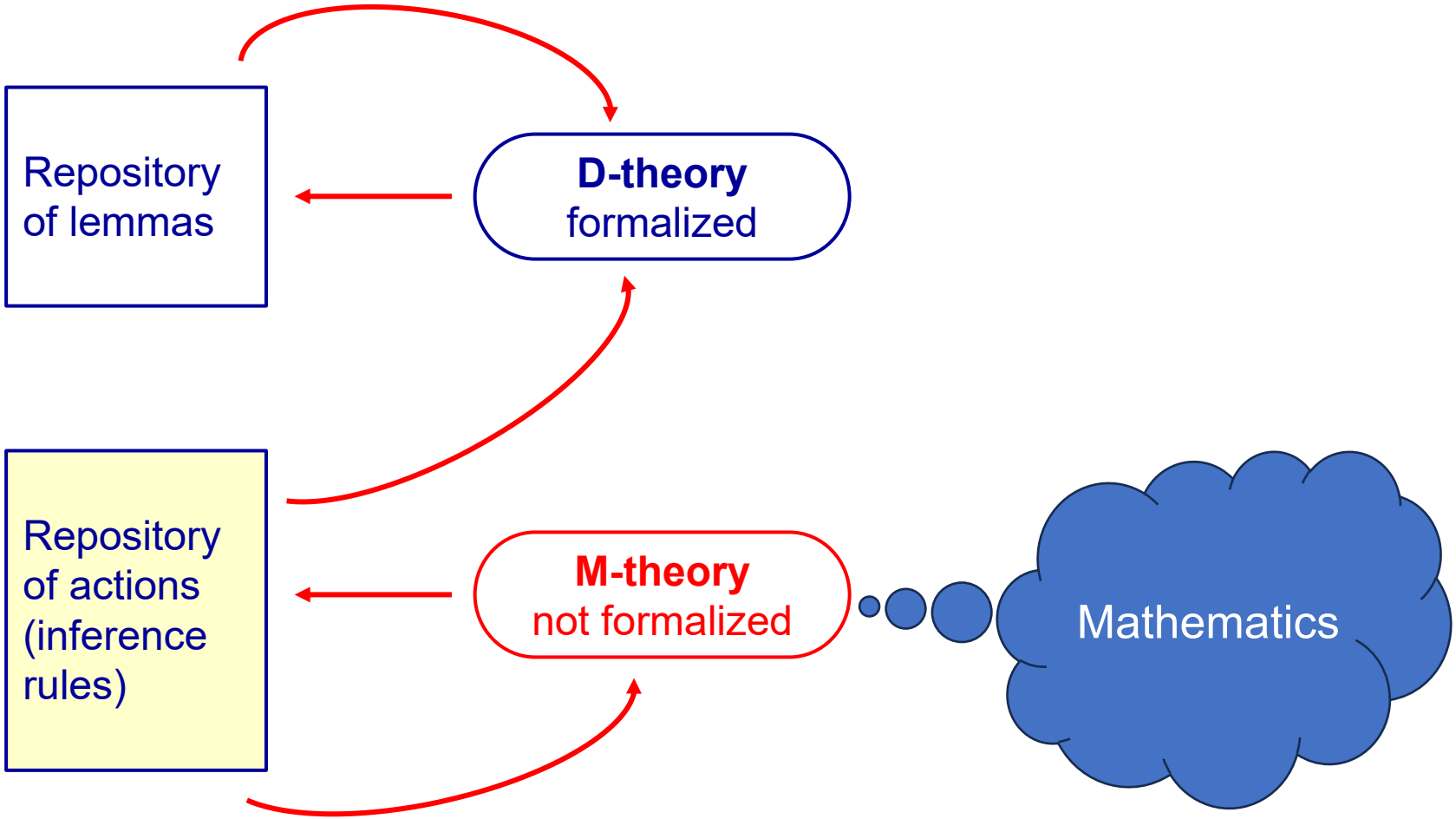
Actions
Non-standard actions – prover-activation action

prove : MetCon-D x Name ⟼ Repository → Repository | Error
prove.(mec, nam).rep =
not is-metacondition.mec ➔ „metacondition expected”
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