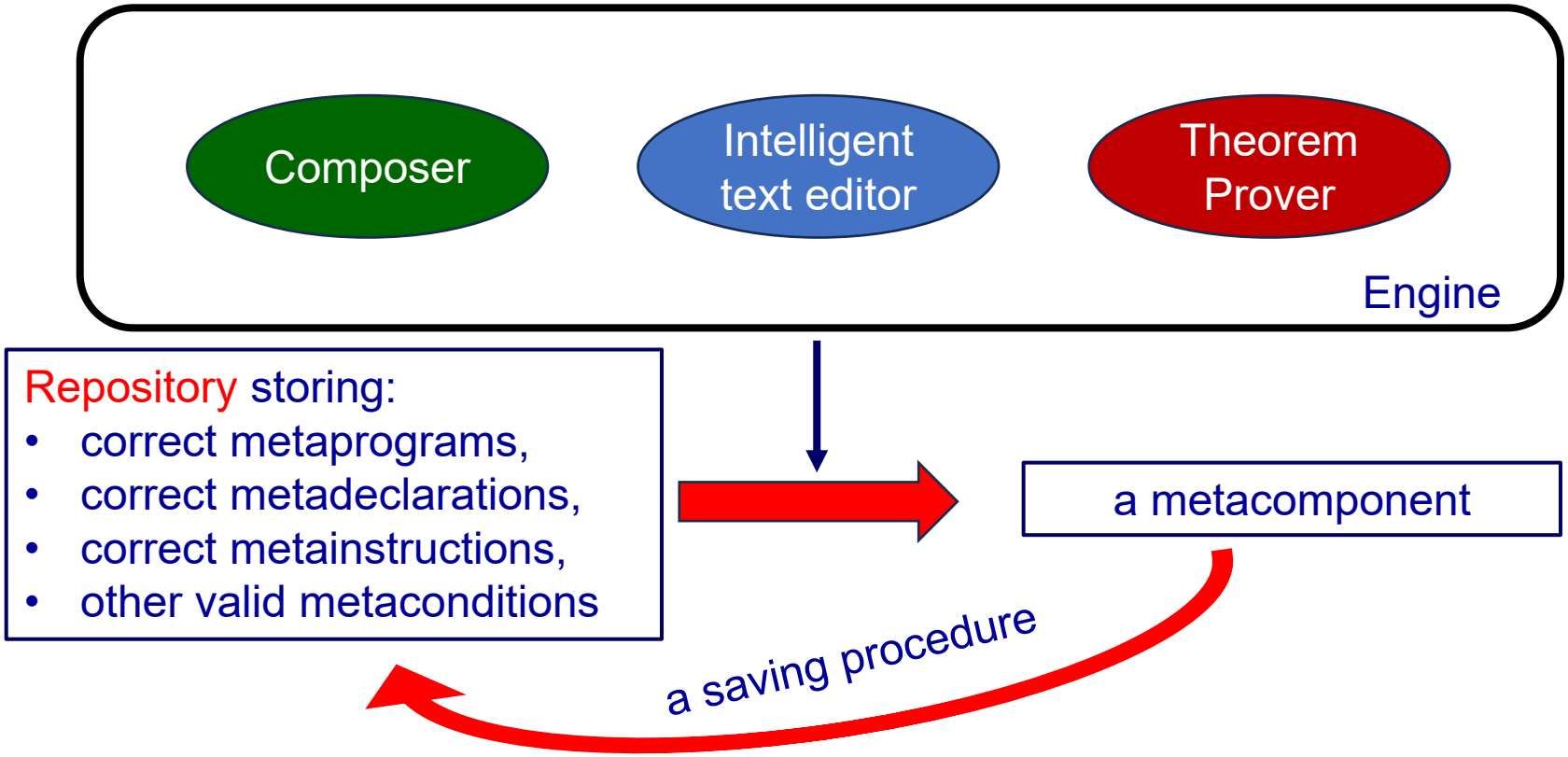
Investigations on ecosystems for Lingua-V programmers

Andrzej Jacek Blikle

August 23rd, 2025

An ecosystem of programmers in Lingua-V



Steps of program-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 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-FT**:
   a. lifting **AlgSyn-V** to **AlgSyn-FT**,
   b. lifting **Alg-Den-V** to **AlgDen-FT**.
3. Choosing an axiomatic framework for **Lingua-FT**:
   a. extending **Lingua-FT** to **Lingua-AFT** (axiomatic FT),
   b. establishing a set of axioms for which **AlgDen-FT** is a model.
4. 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-FT = **Lingua-V** + variables
Lingua-AFT = **Lingua-FT** + axioms-oriented-elements
Lingua-E — **Lingua-AFT**-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-AFT,**
- a set of axioms,
- a set of inference rules.

An ecosystem as a set of tools

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)          |          ground formulas  – no variables: 1< 2
   or(Formula, Formula)           |          free formulas    – with variables: x < x+1
   implies(Formula, Formula)      |
   (∀ Variable) Formula           |
   (∃ Variable) Formula
```

A recollection of formalized theories (3)
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; $F[fn] : Uni^{cn} \longmapsto Uni$ for arity.fn = n
$F[fn] : \longmapsto Uni$ for arity.fn = 0

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

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

vlu : Valuation = Variable $\longmapsto$ Uni

The semantics of variables, terms and formulas:

SV : Variable $\longmapsto$ Variable
ST : Term $\longmapsto$ Valuation $\longmapsto$ Uni
SF : Formula $\longmapsto$ Valuation $\longmapsto$ Bool

A recollection of formalized theories (4)
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 (5)
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 |- zer ≠ suc(zer)

A recollection of formalized theories (6)
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

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

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

Rule of generalization

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

x free in for(x)

Rule of detachment

$$\frac{A \models for1 \qquad A \models for1 \rightarrow for2}{A \models for2}$$

In second-order theories we have also
- functional variables,
- predicational variables.

Gödel's completeness theorem
In every first-order theory with axioms A
A |- for iff A |= for

Gödel's adequacy theorem
In every second-order theory with axioms A
if A |= for then A |- for

Consistency and completeness of formalized theories

Definition

A theory is consistent if it has a model. niesprzeczna, spójna

Theorems:

1. A theory is consistent iff there is no 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.

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!

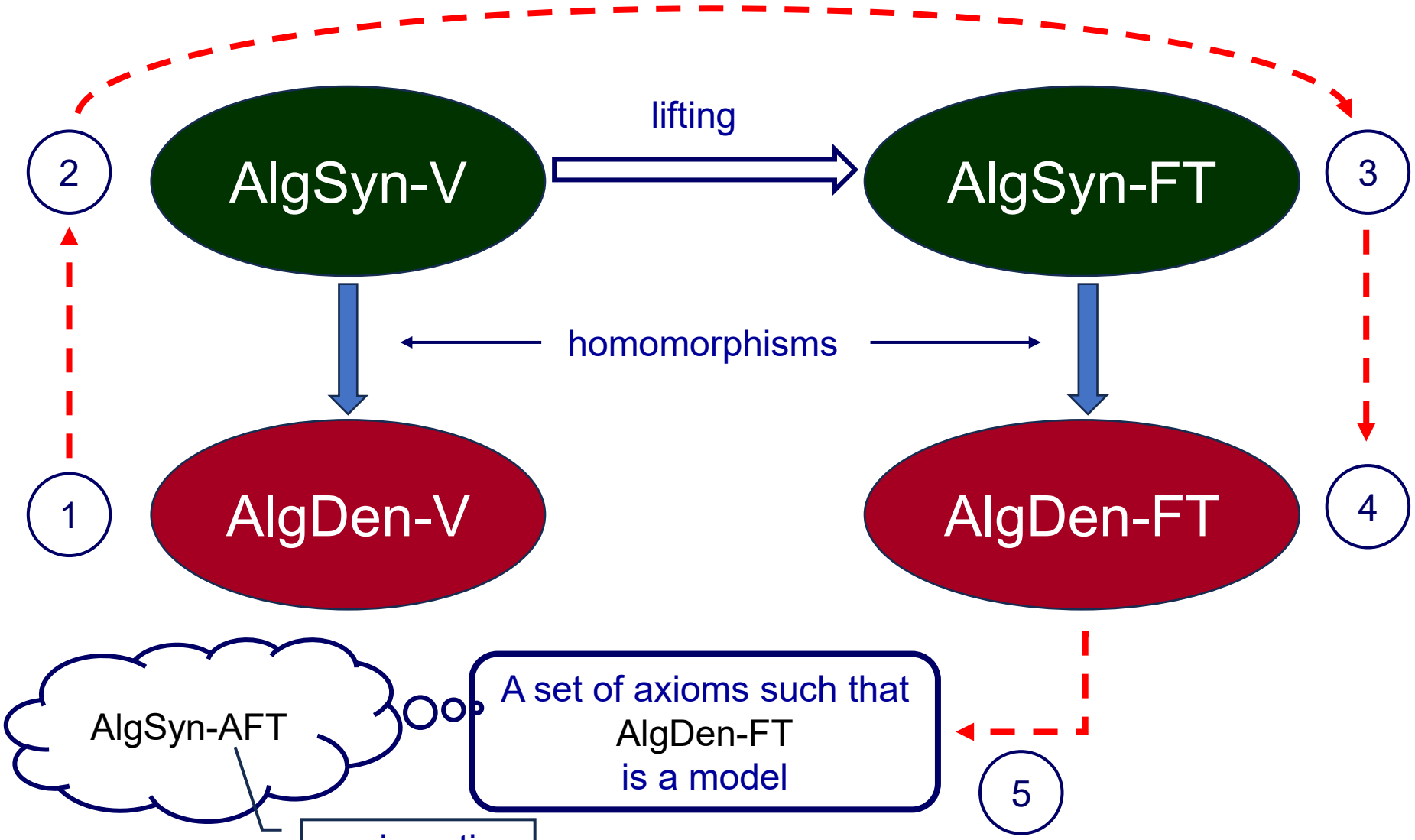
Observation

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

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

Lifting Language-V to Language-FT (1)

FT – formal theory



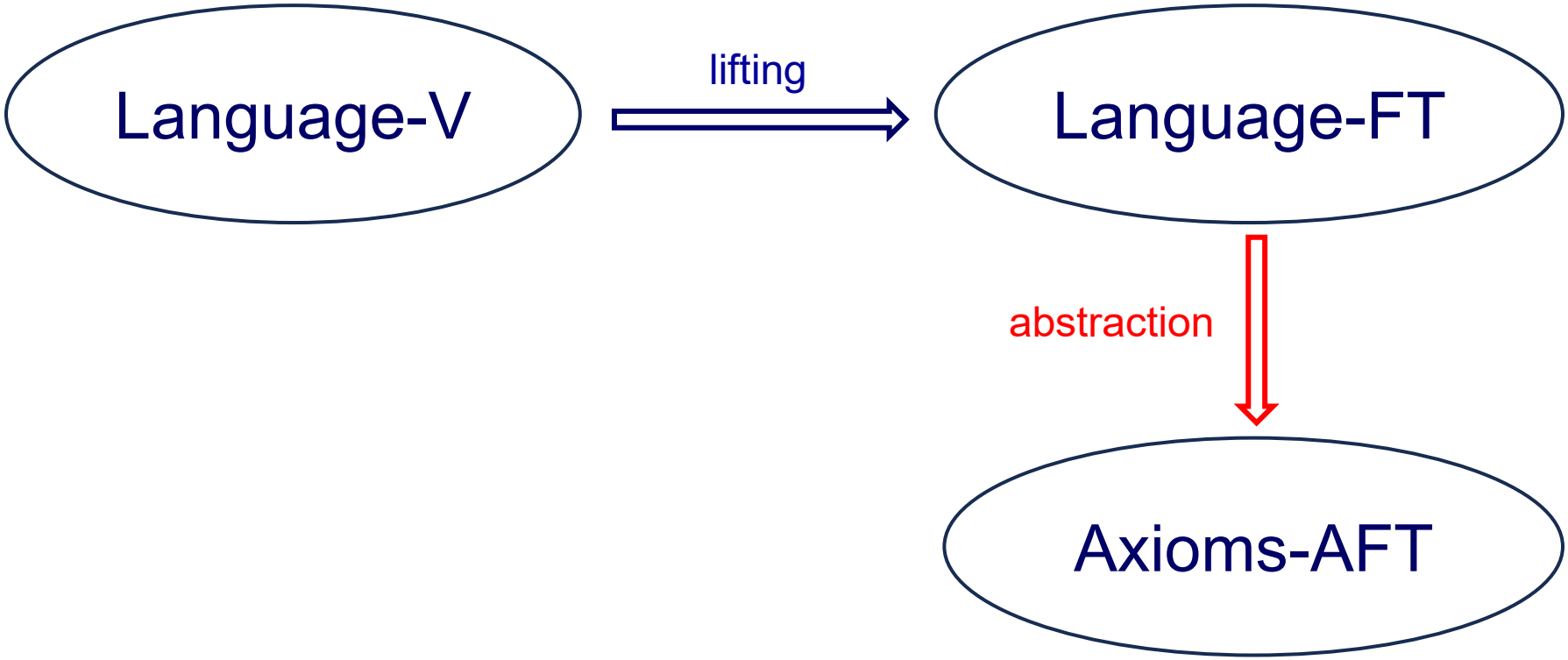
Lifting Language-V to Language-FT (2)

How theories and models are built

mathematical logic: axioms ➔ its models

working mathematician: a model ➔ its axioms

our case



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

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

Individual variables – for all cn : Cn-V

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

Second-order variables:

vacate (so far)

Variables if **Lingua-FT** run over the denotations of **Lingua-V**.

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

value expressions

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

Note the difference:

vex – a metavariable running over domain ValExp-FT

vex – an individual variable of sort „value expression" in **Lingua-FT**

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

specinstructions

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

identifiers

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

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

conditions

```
con : Con-FT =
        con-or-k(Con-FT , Con-FT)                | Kleene's alternative
        con-less-int(ValExp-FT , ValExp-FT)      |
        con-is-value(ValExp-FT)                  |
        con-is-free(Ide-FT)                      |
        con-left-algorithmic(SpeIns-FT , Con-FT) |
        con-right-algorithmic(Con-FT , SpePro-FT)|
        …
```

examples of ground terms (in abstract syntax)

```
sin-assign(x, vex-divide-re(1, z))          x := 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 (note: symbols of variables are underlined)

```
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-FT (5)
syntax – formulas (metaconditions)

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

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

ground formulas in **Lingua-FT** 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-FT** (axioms):

```
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-FT (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-FT 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-FT (8)
denotations – examples of carriers

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

Program construction rules (1st-order axioms)

1. Rules independent of **Lingua**:
   a. formulas describing properties of metarelations
   b. definitional formulas of behavioral metaconditions including the correctness of metaprograms,
2. Rules dependent of **Lingua**:
   a. declaration-oriented rules,
   b. rules concerning temporal metaconditions,
   c. the rule on assignment instruction,
   d. general implicative construction rules,
   e. rules corresponding to structural instructions,

Program construction rules (1st-order axioms)
rules independent of **Lingua –** 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)
(con1 ≡ con2) **implies** (con1 ⊑ con2)
(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))
…

Program construction rules (1st-order axioms)
rules dependent of **Lingua-V** - examples (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 adding an abstract attribute

```
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 adding a type constant

```
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
```

Program construction rules (1st-order axioms)
rules dependent of **Lingua-V** - examples (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 assignment instruction

pre sin **@** con
  sin
post con

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

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, (,), …}
mcn : MetConNam = Character^{c+} the names of metaconditions
rep : Repository = MetConNam ⟹ ValMetCon-FT

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

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

Categories of actions:

- standard actions,
- specific actions,
- logistic actions,
- proving actions.

The meanings of inference rules in the context our ecosystem

Rule of substitution

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

|= mec[inv/ter]

If
mec is in repository and…
then
mec[inv/ter] may be stored in repository

Rule of detachment

|= mec1
|= (mec1 **implies** mec2)

|= mec2

If
mec1 is in repository
and
(mec1 **implies** mec2) is in repository
then
mec2 may be stored in repository

We need corresponding actions to implement both rules

These rules are not formulas in **Lingua-FT**

mec, inv, ter,… are variables in a **MetaLingua-FT**

Denotational models of ecosystems
Constructors of substitution denotations

Auxiliary functions:

free : IndVar-FT x MetCon-FT ⟼ {tt, ff} variable is free in metacondition
matching : IndVar-FT x Text ⟼ {'OK'} | Error the sort of a variable matches the sort of text

Constructors of the denotations of substitutions:

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

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

Denotational models of ecosystems
Constructor of the denotation of a substitution action

Auxiliary functions:

swap : IndVar-FT x Text ⟼ MetCon-FT ⟼ MetCon-FT | Error — single swap
replace : SubDen ⟼ MetCon-FT ⟼ MetCon-FT | Error — many swaps

substitute : MetConNam x SubDen x MetConNam ⟼ ActDen i.e.
substitute : MetConNam x SubDen x MetConNam ⟼ Repository ⟼ Repository | Error
sub-gro.(mcn-s, sud, mcn-t).rep = -s – source, -t – target
  rep.mcn-s = ? ➔ 'no source metacondition'
  rep.mcn-t = ! ➔ 'target name already assigned'
  let
    mec-s = rep.mcn-s
    mec-t = replace.sud.so-mec
  mec-t : Error ➔ mec-t
  true ➔ rep[mcn-t/mec-t]

Rule of substitution

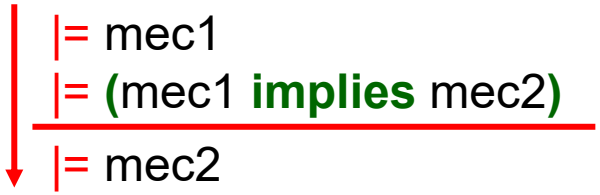
|= mec
inv is free in mec
ter is of sort of inv

|= mec[inv/ter]

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

Denotational models of ecosystems
Standard actions – detachment

Rule of detachment



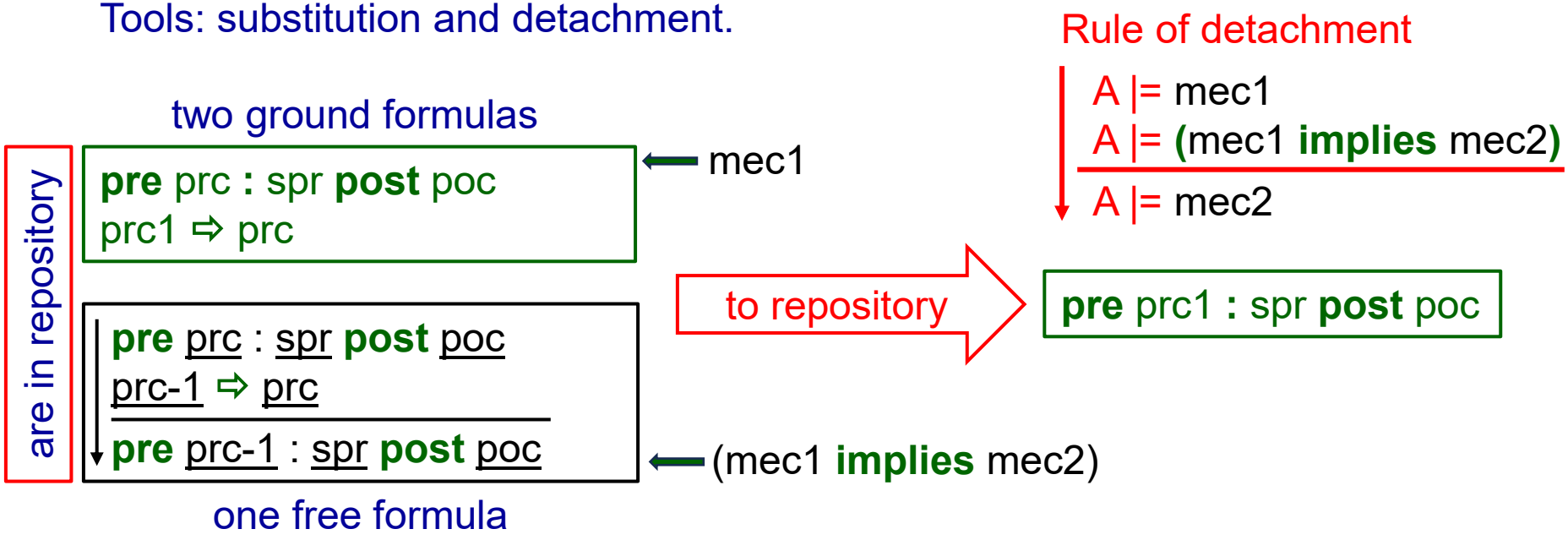
detach : MetConNam x MetConNam x MetConNam ⟼ ActDen
detach : MetConNam x MetConNam x MetConNam ⟼ Repository
⟼ Repository | Error

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

Are substitution- and detachment-action adequate candidates for everyday tools (1)?

A programmer's task: given two metaconditions and a rule generate a new metacondition.

Tools: substitution and detachment.



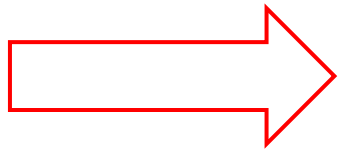
Steps:

1. define and apply to the construction rule a substitution action to get
   ((pre prc **:** spr **post** poc**) and (**prc1 ⇨ prc**)) implies (pre** prc1 **:** spr **post** poc**)**
2. to be able to apply detachment generate and store (combine two formulas)
   ((pre prc **:** spr **post** poc**) and (**prc1 ⇨ prc**))**
3. apply detachment to 1. and 2.

Are substitution- and detachment-action adequate candidates for everyday tools (2)?

A subtask of our programmer: step 2

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



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

Steps:

1. identify in repository an axiom (a free formula):
   (mec1 **implies (**mec2 **implies (**mec1 **and** mec2**))**
2. substitute appropriately to get
   ((pre prc **:** spr **post** poc**) implies ((**prc1 ⇨ prc**) implies**
   ((pre prc **:** spr **post** poc**) and (**prc1 ⇨ prc**)))**
3. detach to get:
   ((prc1 ⇨ prc**) implies ((pre** prc **:** spr **post** poc**) and (**prc1 ⇨ prc**)))**
4. detach again to get
   ((pre prc **:** spr **post** poc**) and (**prc1 ⇨ prc**)))**

7 steps to perform a single replacement of a precondition by a stronger one

Two derived inference rules

If the following inference rule is sound

|= mec1
|= mec2

|= mec1 **and** mec2

due to lemma in repository

|= (mec1 **implies** (mec2 **implies** (mec1 **and** mec2)))

Similarly, the following inference rule is sound

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

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

due to lemma in repository

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

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

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

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

prc1 ⇨ prc

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

The sound action of strengthening precondition

strengthen-pre : MetConNam x MetConNam x MetConNam ⟼ Action
strengthen-pre : MetConNam x MetConNam x MetConNam ⟼ Repository
⟼ Repository | Error

strengthen-pre.(mcn-m, mcn-s, mcn-t).rep = -m – metapr., -s – stronger, -t – target

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

Note! No substitution, no detachment.

Coming back to our programmer's task

Two ground formulas in repository

my-prg → **pre** prc **:** spr **post** poc
my-mec → prc1 ⇨ prc

new-prg → **pre** prc1 **:** spr **post** poc

Activate:
strengthen-pre(my-prg, my-mec, new-prg)

Denotational models of ecosystems
specific actions – the removal of an assertion

A new nonderivable inference rule

The removal of an assertion

|= **pre** prc : head **; asr** con **rsa ;** tail **post** poc

---

|= **pre** prc : head **;** tail **post** poc

This rule can't be derived from a formula in **Lingua-FT,** since scripts

pre prc : head **; asr** con **rsa ;** tail **post** poc

pre prc : head **;** tail **post** poc

are not in MetCon-FT.

They are not metaconditions!

We have to add this rule as an extension of our logic.

Denotational models of ecosystems
specific actions – Lingua-V-oriented

A new nonderivable inference rule

The replacement of an assertion

|= **pre** prc : head **; asr** con **rsa ;** tail **post** poc
|= con ⇔ con1

|= **pre** prc : head **; asr** con1 **rsa ;** tail **post** poc

More such cases:

- the replacement of a boolean value-expression in a program by a strongly equivalent expression,
- the introduction of an assertion block into a program,
- adding a register identifier to a program.

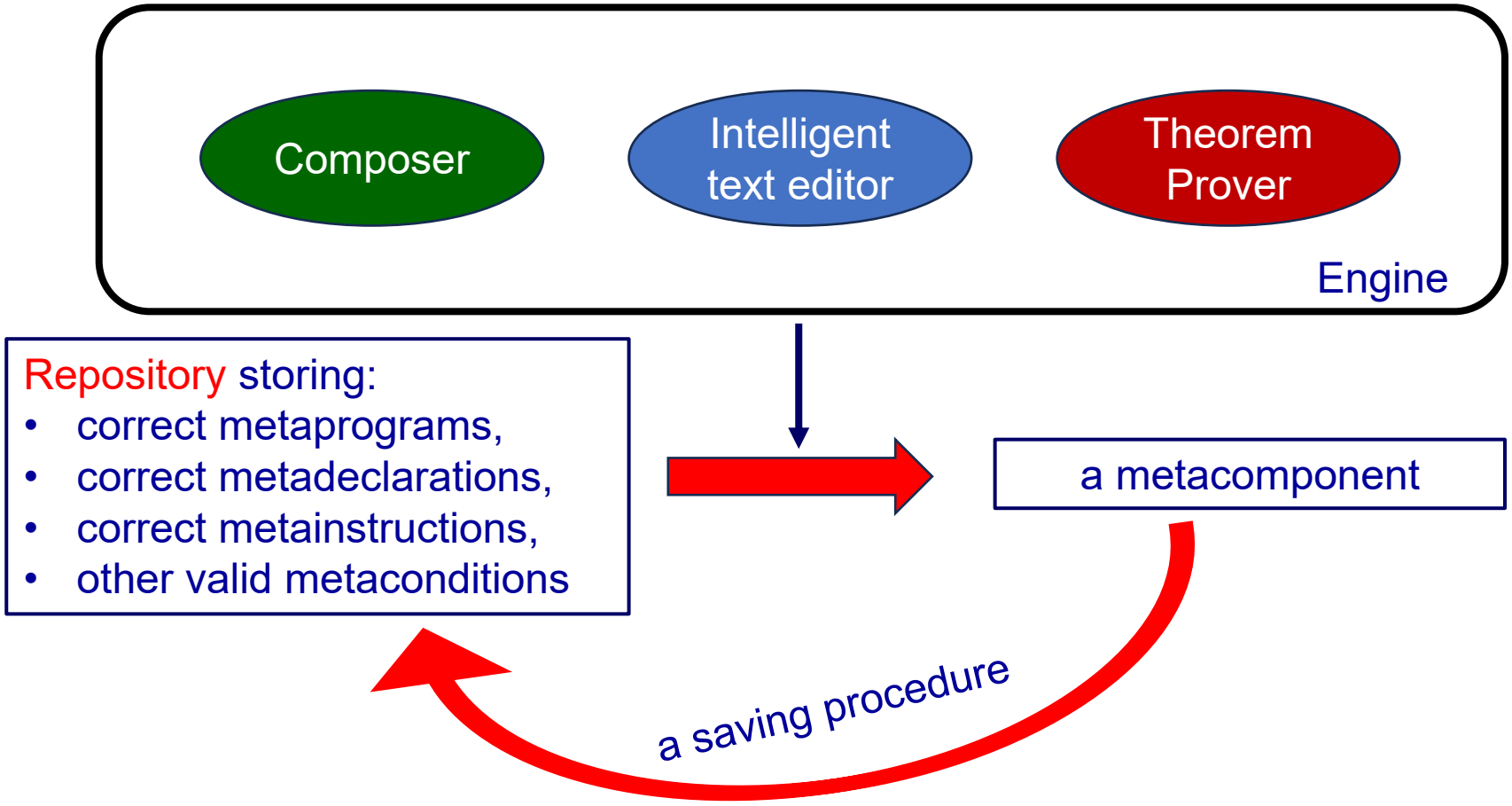
Denotational models of ecosystems
prover-activation action

prove : MetCon-FT x MetConNam⟼ Repository → Repository | Error
prove.(mec, mcn).rep =
  rep.mnc = ! ➔ ‘name already assigned’
  valid.mec : Error ➔ valid.mec
  valid.mec = ? ➔ ?
  valid.mec = ‘NO’ ➔ ‘metacondition not valid’
  true ➔ rep[mcn/mec]

in this definition:

valid : MetCon-FT → {YES, NO} | Error a partial function!


Thank you for
your attention


Composer
Intelligent text editor
Theorem Prover
Engine
Repository storing:
• correct metaprograms,
• correct metadeclarations,
• correct metainstructions,
• other valid metaconditions
a metacomponent
a saving procedure