Ecosystems for programmers in Lingua-V

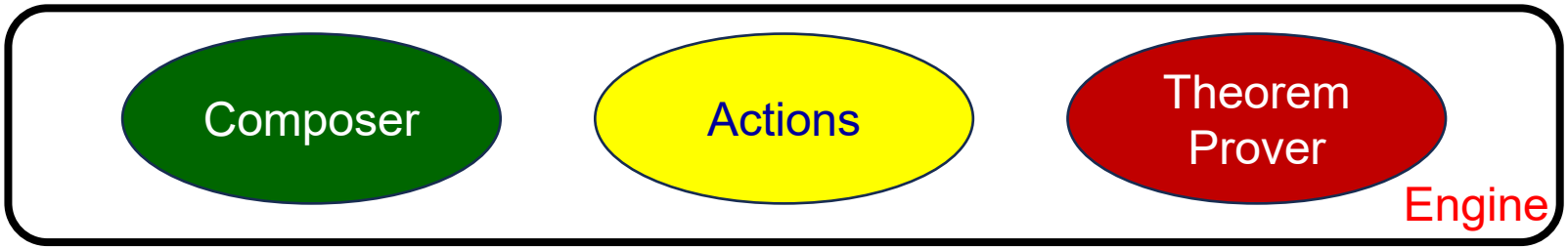
Andrzej Jacek Blikle

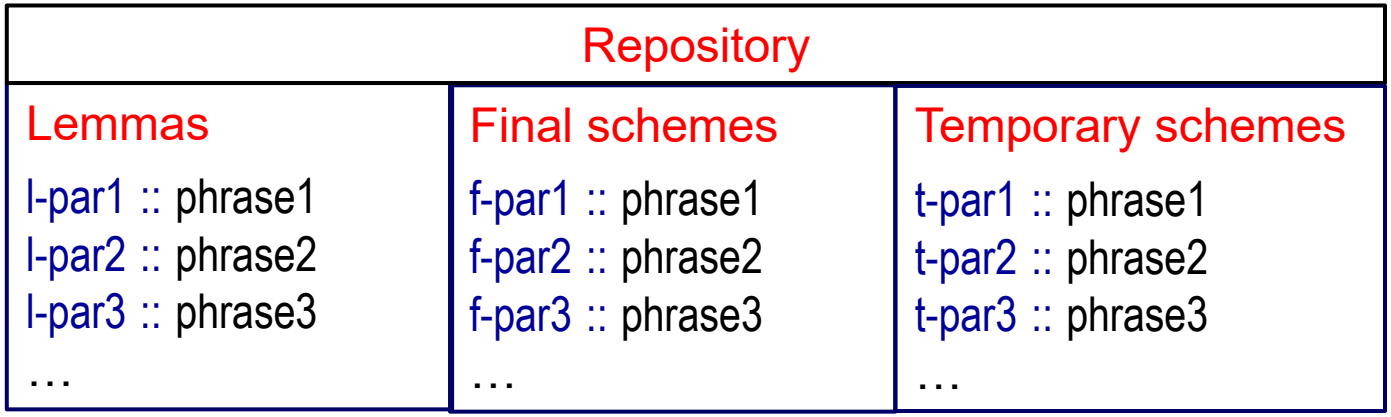
February 29th, 2026

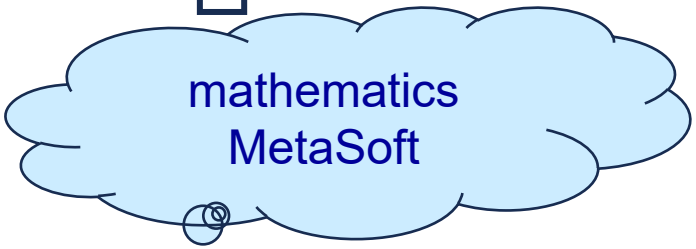
Preliminaries of ecosystems

toolboxes of programmers
based on formalized theories

An ecosystem of programmers in Lingua-V







Two reflections:

- we are building a prototype,
- we are at an „assembler level”.

Categories of phrases stored in repositories

Lingua-V = Lingua + Conditions + Metaconditions — finals

```
loop    :: while x > 0 do x := x-1 od
assign  :: x := x-1
lemma1  :: pre x > 0 : while x > 0 do x := x-1 od post x = 0
```

Lingua-T = Lingua-V + Metavariables — patterns

```
lemma2  :: ((pre con1 : spr post con2) and (con3 ⇨ con1 ) ) implies (((pre con3 : spr post con2) )
lemma3  :: (con1 ⇔ con2) implies (con2 ⇔ con1)
stronger :: (con1 ⇨ con2)
```

Lingua-S = Lingua-T + Parameters — schemes

```
sorting :: pre invariant :
             asr invariant rsa ;
             while arr[j-1] > arr[j] do swap(arr, j - 1, j) ; j:= j-1 od
           post invariant and-k sorted(arr, 0, i)
invariant :: (constant source is array of integer) and-k (var i, j, n is integer) and-k
             (len(arr) = n) and-k (0 ≤ i ≤ j ≤ n) and-k (n > 0) and-k
             (arr sorted from 0 to i but j) and-k permutation(arr, source)
```

Colors reflect future VSC visualization

Lingua ⊆ Lingua-V ⊆ Lingua-T ⊆ Lingua-S

We need a formalized theory of the denotations of Lingua-V
D-Theory

AlgDen-V must be one of the models of **D-Theory**

D-Theory:
- a formal language **Lingua-T,**
- a set of axioms,
- a set of inference rules.

A supporting ecosystem with a set of tools

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

1. **Language**: **Lingua-T**
2. **Lemmas (including axioms)**:
   a. logical lemmas : the calculus of predicates
   b. general mathematical lemmas : numbers, sets, functions, …
   c. lemmas on **Lingua-V** values : simple and structural,
   d. lemmas on **Lingua-V** denotations : of expressions, instructions, metaconditions, etc.
3. **Inference rules**:
   a. universal rules: substitution, detachment, generalization, etc.; significant
   b. derivable rules – expressible as implicative lemmas; convenient
   c. non-derivable rules – not expressible as lemmas; convenient + significant (?)

we shall concentrate on group 2.d

Two dichotomies

1. A minimal set of axioms (logic) versus dynamically developed set of lemmas (Lingua)
2. Ex-post proofs (logic) versus ex-ante proofs (Lingua)

proved (or derived)
valid metaconditions
ex-post proofs

(x **is** integer) **and-k** (x+1 **is** integer) ⇨ x < x+1

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

derived correct metaprograms
(valid metaconditions)
ex-ante proofs

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

```
pre x > 0 :
   while x > 0 do x := x-1 od
post x = 0
```

Languages
offered by our ecosystem

Four languages in one – Lingua-S (1)

Lingua

```
vex : ValExp =
   Identifier                                                    |
   (ValExp + ValExp)                                             |
   push ¥ ValExp ¥ to ¥ ValExp ¥ sup                             |
   …
ins : Instruction =
   RefExp := ValExp                                                                  |
   call-pro ¥ Identifier . Identifier ( val ¥ ActPar ¥ ref ¥ ActPar )                |
   …
```

¥ denotes a space

Lingua-V = Lingua + Conditions + Metaconditions

```
con  : Condition  = … new category
mec  : MetCon     = … new category

vex : ValExp      = … as in Lingua
sin : SpeIns      =     changed definition
   asr Condition rsa                                             |
   RefExp := ValExp                                              |
   if ¥ ValExp ¥ then ¥ SpeIns ¥ else ¥ SpeIns fi                |
   …
```

Lingua-V:

- two new categories,
- applicative categories of **Lingua** unchanged,
- imperative categories of **Lingua** modified,

Four languages in one – Lingua-S (2)

Lingua-T = Lingua-V + Metavariables

```
vex : ValExp-T =
   MetVarVex        |
   … as in Lingua
sin : SpeIns-T =
   MetVarSin                                                    |
   asr Condition-T rsa                                          |
   RefExp-T := ValExp-T                                         |
   if ¥ ValExp-T ¥ then ¥ SpeIns-T ¥ else ¥ SpeIns-T fi         |
   …
```

patterns

Lingua-S = Lingua-T + Parameters

```
vex : ValExp-S =
   Parameter        |
   MetVarVex        |
   … as in Lingua-V
```

schemes

Parameter = % Label %

The syntax of metavariables in Lingua-T

Label = (Letter | Digit | _)*

Metavariables of new categories of Lingua-V

MetVarCon = $ con Label $	condition metavariables
MetVarMec = $ mco Label $	metacondition metavariables
MetVarAct = $ act Label $	anchored class-transformer metavariables
MetVarFcc = $ fcc Label $	funding-class creators metavariables

Metavariables of modified categories of Lingua-V

MetVarIns = $ ins Label $	specinstruction metavariables
MetVarSde = $ sde Label $	specdeclaration metavariables
MetVarSct = $ sct Label $	specified-class-transformer metavariables
MetVarSpp = $ spp Label $	specified program-preamble metavariables
MetVarSpr = $ spr Label $	specprogram metavariables

Metavariables of unchanged categories of Lingua-V

MetVarIde = $ ide Label $	identifier metavariables
MetVarCli = $ cli Label $	class indicator metavariables
MetVarPsi = $ psi Label $	privacy statuses indicator metavariables
MetVarTex = $ tex Label $	type-expression metavariables
MetVarVex = $ vex Label	value-expression metavariables

…

The semantics of Lingua-T

Valuations of metavariables:

Val-MetVarIde = MetVarIde ⟼ Identifier — valuations of identifier metav.
Val-MetVarTex = MetVarTex ⟼ TypExpDen — valuations of type-expression metav.
Val-MetVarVex = MetVarVex ⟼ ValExpDen — valuations of value-expression metav.
…
vlu : Valuation = Val-MetVarIde | Val-MetVarTex | Val-MetVarVex | …

valuations of different categories have disjoint ranges

The semantics of specified instructions

SEMsin : SpeIns ⟼ Valuation ⟼ InsDen
SEMsin.[if vex then ins1 else ins2 fi].vlu = ind-if.(vlu.vex, vlu.ins1, vlu.ins2)

constructor of the denotation of if-instructions from Lingua

Instead of the semantics of Lingua-S

Schemes have denotations (meanings) only in the context of a concrete (state of) a repository.

That is the meaning of a pattern resulting from the scheme when all parameters are filled with their designators

A scheme is a lemma if the corresponding pattern is a lemma.

An example of a satisfied scheme:

```
pre invariant :
        asr invariant rsa ;
        while arr[j-1] > arr[j] do swap(arr, j - 1, j) ; j:= j-1 od
post invariant and-k sorted(arr, 0, i)
```

under the condition that

```
invariant :: (constant source is array of integer) and-k (var i, j, n is integer) and-k
             (len(arr) = n) and-k (0 ≤ i ≤ j ≤ n) and-k (n > 0) and-k
             (arr sorted from 0 to i but j) and-k permutation(arr, source)
```

D-Theory
an ecosystem seen
as a formalized theory

The satisfaction of patterns
(a link to the theory)

D-theory should be defined in this way
that **AlgDen-V** is one of its models

A pattern of a metacondition is said to be satisfied in **AlgDen-V** (or simply satisfied) if it is true (returns tt) for all valuations of metavariables in this algebra.

Examples of satisfied patterns

- ((**pre** con1 : spr **post** con2) **and** (con3 ⇨ con1)) **implies** (((**pre** con3 : spr **post** con2))
- (con1 ⇔ con2) **implies** (con1 ⇨ con2)

Examples of non-satisfied patterns

- (con1 ⇨ con2) satisfied (only) sometimes
- (ide **is** integer) **and-k** ((ide+1) **is** integer) ⇨ ide > ide+1 never satisfied

Our scenario of building a formalized **D-Theory**

Intuitive theories
Collections of definitions and theorems developed dynamically.

to be used

Formalized theories
A "minimal" set of axioms + inference rules.

to be investigated

D-Theory

Initial components of D-Theory
- initial set of lemmas,
- initial set of schemes,
- initial set of inference rules.

programmers

operators

- classical predicate calculus,
- values of **Lingua**,
- denotations of **Lingua**.

- rules derivable from lemmas,
- nonderivable rules,

A taxonomy of denotation-oriented lemmas

1. Lemmas on metapredicates:
   a. relational: ⇨, ⇔, ⊑, ≡,
   b. metaprograms.
   c. behavioral,
   d. temporal,
   e. language-dependent.
2. Implicative lemmas with metaprograms (will be introduced as inference rules).

Slides that follow include only selected examples

Lemmas on relational metaconditions

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)** the weakest error-transparent condition

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

Lemmas on single metaprograms (1)

Definitional axiom

```
(pre con1 : spr post con2) iff con1 ⇨ spr @ con2
```

Standard-integers lemma:

```
pre (ide1, ide2 is-v integer) and-k (ide2 > 0) :
  ide1 := 0;
  while ide1 < ide2
    do ide1 := ide1 + 1 od
post ide1 = ide2
```

This lemma guarantees that all models of **D-Theory** include standard arithmetics.

Algorithmic lemmas stand for second-order axioms

Tautological lemma

```
pre spr @ con :
  spr
post con
```

Lemmas on single metaprograms (2)

Lemma for variable declaration

```
pre (ide is-free) and-k (tex is-type)
   let ide be tex tel
post ide is-v tex
```

Lemma for an abstract attribute declaration

```
pre (ide1 is free) and-k (ide2 is class) and-k (tex is type) :
   let ide1 be tex as pst tel in ide2
post att ide1 is tex in ide2 as pst
```

Lemma for a type constant declaration

```
pre (ide1 is free) and-k (ide2 is class) and-k (tex is type) :
   set ide1 be tex tes in ide2
post ide1 is tex
```

… and analogous lemmas for all categories of declarations

Lemmas on behavioral metaconditions

Axiomatic definition

(con **insures-LR-of** sin) **iff pre** con : sin **post NT**

Lemmas (examples):

(ide1, ide2 **is** integer) **and-k** (ide2 > 0) **insures-LR-of**
ide1 := 0; **while** ide1 < ide2 **do** ide1 := ide1 + 1 **od**

(sin @ con) **insures-LR-of** sin

Axiomatic definition

(con **resilient-to** spr) iff (con @ spr ⇨ con)

Lemmas (examples)

(**pre** con1 : spr **post** con2) **and** (con2 => (ide **is free**)) **implies**
(ide **is free**) **is-resilient** (**pre** con1 : spr **post** con2)

All post-declaration conditions are resilient to all instructions, e.g.,

((ide is-v tex) **resilient-to** sin)

Lemmas on temporal metaconditions

Denotational definition

(con **induced-in** mpr**)** iff con eventually satisfied in the execution of mpr

Lemma (example)

```
(pre con1 : spp1; let ide be tex tel; spp2 : opr ; sin post con)
implies
((ide is tex) induced-in
                    (pre con1 : spp1; let ide be tex tel; spp2 : opr ; sin post con))
```

Denotational definition

(con **hereditary-in** mpr) iff once con is satisfied, it remains satisfied

Lemma (example)

```
(ide1 = 3 hereditary-in
                (pre (var ide1, ide2 is integer) :
                     if ide1 > 0 then ide1 := 3 else ide1 := 2 fi; ide2 := 0
                post ide1 > 0)
```

Lemmas on language-dependent metaconditions

Axiomatic definition (hereditary in all metaprograms)

immunizing(con) **iff** (con **hereditary-in** mpr)

Lemmas on examples of post-declaration conditions

immunizing(tex **is-type**)
immunizing(ide i**s-class**)
immunizing(ide i**s-child-of** cli)
immunizing(ide **not**(**is-free**))

Denotational definition (not to be a lemma in repository):

immanent(con) iff [con].sta ≠ fv for all sta

Lemmas on immanent conditions (may be an error or undefined)

immanent(vex < vex + 1)
immanent(vex = vex)

Lemmas nearly-true conditions (always true unless state carries an error)

nearly-true(1 < 1+1)
nearly-true(((vex **is** integer) **and-k** (vex+1 **is** integer) **implies** (vex < vex+1))

Lemmas on nonderivable conditions

underivable(ide **is-free)**
underivable(ide ISO/IEC 9075**)** an ISO standard for databases

Inference rules derivable from implicative lemmas (1)

Metatheorem

If mec1, mec2 and mec3 are metaconditions such that

(mec1 **and** mec2) **implies** mec3 i.e.

$$\frac{\text{mec1} \quad \text{mec2}}{\text{mec3}}$$

is a lemma, then the following inference rule is sound

$$\frac{\text{|- mec1} \quad \text{|- mec2}}{\text{|- mec3}}$$

i.e. may be included in our repertoire of actions

Instead of implicative lemmas in the repository, we include lemma creators in the engine.

To prove this metatheorem we use detachment rule and the following lemma:

mec1 **implies** (mec2 **implies** (mec1 **and** mec2))

This metatheorem may be generalized for an arbitrary number of metaconditions in the conjunction (above the line).

Inference rules derivable from implicative lemmas (2) (example)

Since the following implication

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

prc1 ⇨ prc

---

pre prc : spr **post** poc

is a lemma, the following inference rule is sound:

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

|- prc1 ⇨ prc

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

For every such inference rule we may define a corresponding creator.

Expressed in **Lingua-T**

Expressed in **MetaSoft**

Non-derivable inference rules (1)

assignment-instruction rules

A universal lemma for assignment

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

A non-derivable construction rule for assignment:

pre is-reference(ide**) and-k is-value(**vex**) and compatible(**ide**,** vex**) and-k** con[ide/vex] **:**
    ide := vex
post con

All logical inference rules like substitution, detachment, etc. are non-derivable

requires a nontrivial inductive definition

Non-derivable inference rules (2)

further 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
```

Lingua-E
an ecosystem seen
as a programming language

Part I:
repositories

Repositories – definition

rep	: Repository	= LemRep x FinSchRep x TemSchRep x Error	
ler	: LemRep	= Parameter ⟹ TypLem	lemma repositories
fsr	: FinSchRep	= Parameter ⟹ TypSch	final-scheme repositories
tsr	: TemSchRep	= Parameter ⟹ TypSch	temporary-scheme repositories
tsc	: TypSch	= Article x Scheme	typed schemes
tle	: TypLem	= Article x Lemma	typed lemmas
lem	: Lemma	= …	lemmas – satisfied metaconditions
sch	: Scheme	= …	schemes – elements of **Lingua-S**
err	: Error	= {'OK'} | ...	error register (like in **Lingua**)
par	: Parameter	= **{%}** Label **{%}**	
art	: Article	= {**ins**, **sde**, **sct**, …}	

A repository is said to be well-formed if:

1. every parameter that appears in a scheme is declared in the repository,
2. every parameter is declared in at most one subrepository,
3. every typed-scheme (art, sch) that is a designator of a parameter is well-typed, i.e., the syntactic cathegory of sch is art (also grammatical correctness); the same rule concerns lemmas.

WfRep – the domain of well-formed repositories

The development of well-formed repositories

Initial content of LemRep

a. logical lemmas : the calculus of predicates
b. general mathematical lemmas : sets, functions,…
c. lemmas on **Lingua-V** values : simple and structural;
d. lemmas on **Lingua-V** denotations : metaconditions, conditions, specprograms,…

Initial content of TemSchRep – all basic patterns of the grammar of **Lingua-V**

Initial content of FinSchRep – empty

Further development – a dynamic approach to "lemmatization"

1. Programmers may modify repositories exclusively by using actions.
2. Programmers may add new actions exclusively derived within the ecosystem as implicative lemmas.
3. Operators may add lemmas and actions proved „outside" of the ecosystem, provided that AlgDen-V remains a model of D-Theory!

instead of an intelligent editor

1. – 3. guarantee that AlgDen-V remains a model of D-Theory (by Gödel's theorem)

Initial content of TemSchRep

For every grammatical clause of Lingua-V we add to SchRep one typed pattern, e.g.:

ConMetVar	→ (con, con)
(ValExp < ValExp)	→ (vex, (vex1 < vex2))
let ¥ LisOfIde ¥ be ¥ TypExp ¥ tel	→ (dse, let loi ¥ be ¥ tex tel)
(SpePro @ Condition)	→ (con, (spr @ con))
(Condition ⇨ Condition)	→ (mec, (con1 ⇨ con2))
immunizing(Condition)	→ (mec, immunizing(con))
(MetCon ¥ and ¥ MetCon)	→ (mec, (mec1 ¥ and ¥ mec2))
while ¥ ValExp ¥ do ¥ SpeIns ¥ od	→ (sin, while ¥ vex ¥ do ¥ sin ¥ od)
SpeProPre ; OpePro ; SpeIns	→ (spr, spp ; opr ; sin))
empty-class	→ (cli, empty-class)
Identifier	→ (cli, ide)

Grammatical categories in grammatical clauses are replaced by corresponding metavariables.

Fact: Initial content of TemSchRep it the least content that allows for the generation of all patterns of **Lingua-T** by the operations of substitution and indexations.

Conclusion: The repository of temporary schemes with substitution and indexation may stand for (serve as) an intelligent grammar-driven editor.

Indexations – actions that add or change indexes of metavariables in schemes.

Scheme repositories
a programmers' view in VSC

(programmers do not see articles)

TemSchRep

```
con-less-int     :: ( vex, ( vex1 < vex2 ) )                    condition less-than for integers
sde-var          :: ( dse, let loi ¥ be ¥ tex tel )               specdeclaration of a variable
mec-left-algor   :: ( con, ( spr @ con ) )                          condition; left algorithmic
mec-met-pro      :: ( mec, ( pre ¥ con1 : spr ¥ post ¥ con2 ) )                 metaprogram
mec-stronger     :: ( mec, ( con1 => con2 ) )                    metacondition; stronger than
```

FinSchRep

```
invariant        :: (constant source is array of integer) and-k (var i, j, n is integer) and-k
                    (len(arr) = n) and-k (0 ≤ i ≤ j ≤ n) and-k (n > 0) and-k
                    (arr sorted from 0 to i but j) and-k permutation(arr, source)
```

Lingua-E
an ecosystem seen
as a programming language

Part II:
creators of schemes

The algebra of denotations of Lingua-E

Carriers of the algebra of denotations AlgDen-E

expressions

scd : SchCreDen = WfRep ⟼ TypSchE
lcd : LemCreDen = WfRep ⟼ TypLemE
acd : ActDen = WfRep ⟼ WfRep
mev : MetVar = {$} Article Label {$}
par : Parameter = {%} Label {%}

scheme-creator denotations
lemma-creator denotations
action denotations

instructions

where (non-carriers)

art : Article = con | mco | act | fcc | ins | …
lab : Label = Character^{c+}
cha : Character = Letter | Digit | - | _

reminder:
DomainE = Domain | Error

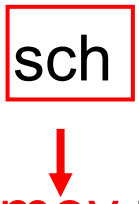
Constructors of scheme-creators' denotations:

get-S	: Parameter	⟼ SchCreDen get designator
sch4mev-S	: SchCreDen x MetVar x SchCreDen	⟼ SchCreDen scheme for mev
par4mev-S	: Parameter x MetVar x SchCreDen	⟼ SchCreDen par for mev
sch4par-S	: SchCreDen x Parameter	⟼ SchCreDen scheme for par

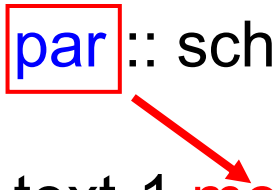
We separate **creators** of (the denotations of) schemes and lemmas, from **actions** that store them in repositories.

Constructors of scheme creators (2)
(substitutions in schemes)

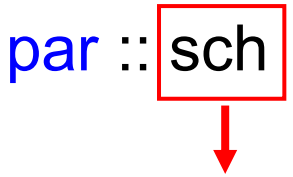
sch4mev-S:



par4mev-S:



sch4par-S:



All constructed creators
check sort compatibility
to preserve
grammatical correctness
of schemes.

Auxiliary functions

free	: MetVar x Scheme	⟼ {tt, ff}	
is-in	: Parameter x Scheme	⟼ {tt, ff}	
artOfMev	: MetVar	⟼ Article	
schAsArt	: Scheme x Article	⟼ {tt, ff}	parsing
parameters-final	: Scheme	⟼ {tt, ff}	
replace-m	: Scheme x MetVar x Scheme	⟼ Scheme	
replace-p	: Scheme x Parameter x Scheme	⟼ Scheme	

unconditional replacements of all occurrences of a given metavariable/parameter by a scheme in a scheme

replace-m.(sch1, mev, sch1)
all occurrences of variable mev are replaced by scheme sch1 in scheme sch2

Constructors of scheme creators (1)
(get the designator of a parameter)

```
get-S : Parameter ⟼ WfRep ⟼ TypSchE
get-S.par.rep =
  is-error.rep   ➔ error.rep
  let
    (ler, fsr, tsr, err) = rep
  ler.par # !    ➔ ler.par
  fsr.par # !    ➔ fsr.par
  tsr.par # !    ➔ tsr.par
  true           ➔ 'parameter not declared'
```

Constructors of scheme creators (3)
(substitutions in schemes)

The substitution of a scheme generated by scd1 for a variable mev in a scheme generated by scd2.

```
sch4mev-S : SchCreDen x MetVar x SchCreDen ⟼ SchCreDen
sch4mev-S : SchCreDen x MetVar x SchCreDen ⟼ WfRep ⟼ TypSchE
sch4mev-S.(scd1, mev, scd2).rep =
  is-error.rep              ➔ rep
  scd1.rep : Error          ➔ rep ◄ scd1.rep
  scd2.rep : Error          ➔ rep ◄ scd2.rep
  let
    (art1, sch1) = scd1.rep
    (art2, sch2) = scd2.rep
    art-m        = artOfMev.mev
  not free.(mev, sch2) ➔ 'metavariable not free'
  art1 ≠ art-m         ➔ 'incompatibility of sorts'
  let
    new-sch = replace-m.(sch1, mev, sch2)
  true                 ➔ (art2, new-sch)
```

Constructors of scheme creators (4)
(substitutions in schemes)

The substitution of a parameter par for a metavariable mev in a scheme created by scd.

```
par4mev-S : Parameter x MetVar x SchCreDen ⟼ SchCreDen
par4mev-S : Parameter x MetVar x SchCreDen ⟼ WfRep ⟼ Scheme | Error
par4mev-S.(par, mev, scd).rep =
  is-error.rep              ➔ rep
  get.par.rep : Error       ➔ rep ◄ 'parameter undeclared'
  fsr.par # ?               ➔ rep ◄ 'parameter must be declared as final'
  let
    (art-p, sch-p)  = get.par.rep
    (art-s, sch-s)  = scd.rep
    art-m           = artOfMev.mev
  not free.(mev, sch)       ➔ 'metavariable not free'
  art-p ≠ art-m             ➔ 'incompatibility of sorts'
  let
    new-sch = replace-m.(par, mev, sch-s)
  true                      ➔ replace.(par, mev, sch-s)
                                (note that par is a scheme itself)
```

Parameter substituted for a metavariable must be declared!

Constructors of scheme creators (5)
(substitutions in schemes)

The substitution of a designator of a parameter par in a scheme created by scd.

sch4par-S : SchCreDen x Parameter ⟼ SchCreDen
sch4par-S : SchCreDen x Parameter ⟼ WfRep ⟼ TypSchE

```
sch4par-S.(scd, par).rep =
  is-error.rep              ➔ rep
  scd.rep : Error           ➔ rep ◄ scd.rep
  get-S.par.rep : Error     ➔ rep ◄ get-S.par.rep
  not is-in.(par, sch-s)    ➔ 'parameter not in the scheme'
  let
    (ler, fsr, tsr, err) = rep
    (art-p, sch-p)     = get-S.par.rep
    (art-s, sch-s)     = scd.rep
  true                      ➔ replace-p.(sch-p, par, sch-s)
```

For a parameter we may substitute only its designator!
We do not need to check if art-p = art-s since
whenever we substitute a parameter for a scheme
the article of the (created) parameter becomes the article of the scheme.

Lingua-E
an ecosystem seen
as a programming language

Part III:
constructors of scheme-oriented actions

The taxonomy of scheme-oriented actions

declare a parameter with created designator

declare-par-SC : Parameter x SchCreDen ⟼ ActDen

declare a parameter with selected designator

declare-par-SS : Parameter x Article x Parameter ⟼ ActDen

declare parameter and replace its designator

declare-par-SSR : Parameter x Article x Parameter ⟼ ActDen

modify a declared parameter

modify-par-S : Parameter x SchCreDen ⟼ ActDen

make a parameter final

finalize-par-S : Parameter ⟼ ActDen

delete a parameter

delete-par-S : Parameter ⟼ ActDen

Parameters that are to be modified
must be assigned to temporary schemes.

Parameter-declaration actions (type SC)
(declare a created (C) scheme as a parameter designator)

```
declare-par-SC : Parameter x SchCreDen ⟼ ActDen
declare-par-SC : Parameter x SchCreDen ⟼ WfRep ⟼ WfRep
declare-par-SC.(par, scd).rep =
  is-error.rep                  ➔ rep
  get.par.rep # !               ➔ rep ◄ 'parameter must not be declared'
  scd.rep : Error               ➔ rep ◄ scd.rep
  let
    (ler, fsr, tsr, err) = rep
    (art, sch)         = scd.rep
  true                          ➔ (ler, fsr, tsr[par/(art, sch)], err)
```

Parameters to be declared must not be declared.

Parameter-declaration actions (type SS)
(declare a selected (S) scheme as a parameter designator)

```
declare-par-SS : Parameter x Article x Parameter ⟼ ActDen
declare-par-SS : Parameter x Article x Parameter ⟼ WfRep ⟼ WfRep
declare-par-SS.(par-n, art-n, par-p).rep =          -n for "new", -p for "parent"
  is-error.rep                          ➔ rep
  get.par.rep # !                       ➔ rep ◄ 'parameter must not be declared'
  get.par-p.rep # ?                     ➔ rep ◄ parent parameter must be declared'
  let
    (ler, fsr, tsr, err) = rep
    (art-p, sch-p)     = get.par-p.rep
    selection          = select.sch-p                    programmer selects a subscheme
  not schAsArt.(selection, art)         ➔ rep ◄ 'incompatibility of sorts'      parsing
  true                                  ➔ (ler, fsr, tsr[par/(art, selection)], err)
```

Parent parameters may be declared in an arbitrary repository.

Parameter-declaration actions (type SSR)

(select and replace (SR) to declare a parameter)

declare-par-SSR : Parameter x Article x Parameter ⟼ WfRep ⟼ WfRep
declare-par-SSR.(par-p, art-n, par-n).rep = -p for “parent par.”, -n for “new par.”
  is-error.rep ➔ rep
  let
    (ler, fsr, tsr, err) = rep
  tsr.par-p # ? ➔ rep ◄ 'parent par. must be temporary’
  get-S.par-n.rep # ! ➔ rep ◄ 'new parameter must not be declared’
  let
    (art-p, sch-p) = get-S.par-p.rep
    head © sel © tail = sch-p programmer selects
  not schAsArt.(sel, art-n) ➔ rep ◄ 'incorrect selection’ parser checks
  let
    new-sch = head © par-n © tail
  true ➔ (ler, fsr[par-n/(art-n, sel], tsr[par-p/new-sch], err)

Parameters to be declared must not be declared.

Parameter-modification actions

```
modify-par-S : Parameter x SchCreDen ⟼ ActDen
modify-par-S : Parameter x SchCreDen ⟼ WfRep ⟼ WfRep
modify-par-S.(par, scd).rep =
  is-error.rep        ➔ rep
  scd.rep : Error     ➔ rep ◄ scd.rep
  let
    (ler, fsr, tsr, err) = rep
  tsr.par # ?         ➔ 'modyfied parameter must be declared as temporary'
  let
    (art-p, sch-p)  = tsr.par
    (art-s, sch)    = scd.rep
  art-s ≠ art-p       ➔ 'incompatibility of sorts'
  true                ➔ (ler, fsr, tsr[par/(art-p, sch-s], err)
```

Parameters that are to be modified
must be assigned to temporary schemes.

Parameter-finalization actions
(move a temporary parameter to the final repository)

```
finalize-par-S : Parameter ⟼ ActDen
finalize-par-S : Parameter ⟼ WfRep ⟼ WfRep
finalize-par-S.par.rep =
  is-error.rep                  ➔ rep
  let
    (ler, fsr, tsr, err) = rep
  tsr.par # ?                   ➔ rep ◄ 'parameter must be temporary'
  let
    (art, sch) = tsr.par
  not parameters-final.sch      ➔ 'all parameters in the designator must be final'
  true                          ➔ (ler, fsr[par/(art, sch)], tsr[par/?], err)
```

Parameters to be finalized are
moved from temporary repository to final repository.

Lingua-E
an ecosystem seen
as a programming language

Part IV:
constructors of lemmas
(i.e., metaprograms in particular)

A taxonomy of inference rules in Lingua-E

Two methodological remarks:

1. **Inference rules** – i.e., lemma making rules – are represented in our ecosystem by the **creators of lemmas**.
2. Actions only save our work for a future use.

Three groups of inference rules:

1. Basic logical rules
   a. substitution of a scheme for metavariable
   b. detachment.
   c. quantifier-oriented rules.
2. derivable rules – derivable from lemmas:
   a. sch4mev-L, par4mev-l, sch4par,
   b. rules derivable from implicative lemmas.
3. Non-derivable rules – non -derivable from lemmas

Remark: Basic rules are non-derivable.

Constructors of lemma-creator denotations

lcd : LemCreDen = WfRep ⟼ TypLemE

Universal L-constructors (including logical inference rules):

get-L	: Parameter		⟼ LemCreDen get designator
sch4mev-L	: SchCreDen	x MetVar x LemCreDen	⟼ LemCreDen scheme for mev
par4mev-L	: Parameter	x MetVar x LemCreDen	⟼ LemCreDen par for mev
sch4par-L	: LemCreDen	x Parameter	⟼ LemCreDen scheme for par
detach-L	: Parameter	x Parameter	⟼ LemCreDen

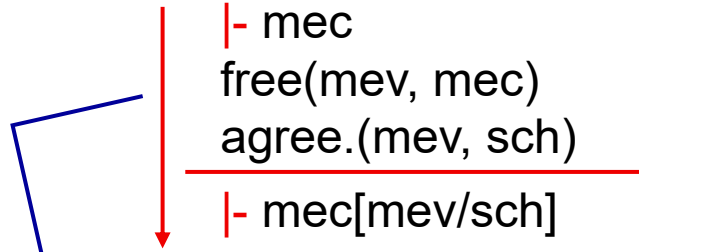
…

Definitions analogous as for S-constructors

Justification of soundness

get-L	- reading from the repository of lemmas
sch4mev-L	- substitution inference rule
par4mev-L	- future substitution
sch4par-L	- filling lemma back with the declared scheme
detach-L	- detachment inference rule

This rule is expressed in **MetaSoft**. E.g. mec (not mec!) is a **MetaSoft** variable and stands for an arbitrary metacondition



The rule of substitution

Constructors of lemma-creators' denotations

lcd : LemCreDen = WfRep ⟼ TypLemE

Selected examples of Lingua-V-oriented L-constructors:

strengthen-pre : Parameter x Parameter ⟼ LemCreDen
weaken-post : Parameter x Parameter ⟼ LemCreDen
compose-sec-1 : Parameter x Parameter x Parameter ⟼ LemCreDen
compose-sec-2 : Parameter x Parameter x Parameter ⟼ LemCreDen
compose-sec-3 : Parameter x Parameter x Parameter ⟼ LemCreDen
compose-seq : Parameter x Parameter x Parameter ⟼ LemCreDen
compose-if : Parameter x Parameter x Parameter ⟼ LemCreDen
compose-while : Parameter x Parameter x Parameter
x Parameter x Parameter ⟼ LemCreDen

…

It is an engineering decision that the constructors of lemma-creators take parameters as arguments rather than other lemma creators.

Experiments with programming will indicate if that was an adequate decision.

Constructors of lemma-creator denotations
(strengthening a precondition; a half-formal definition)

```
strengthen-pre : Parameter x Parameter ⟼ LemCreDen
strengthen-pre : Parameter x Parameter ⟼ WfRep ⟼ TypLemE
strengthen-pre.(par-s, par-m).rep =
  is-error.rep                                  ➔ error.rep
  let
     (ler, fsr, tsr, err) = rep
  ler.par-s # ?                                 ➔ 'no such stronger-than lemma'
  ler.par-m # ?                                 ➔ 'no such metaprogram'
  let
    (mec, str-lem)   = ler.par-s
    (mec, mpr-lem) = let.par-m
  not str-lem     ::= con1 ⇨ con2               ➔ 'a stronger-than-lemma expected'
  not mpr-lem   ::= pre prc : spr post pos      ➔ 'a metaprogram expected'
  con2 ≠ prc                                    ➔ 'stronger-than-lemma inadequate'
  true                                          ➔ pre con1 : spr post pos
```

(meta) patterns

parsing

Non-derivable quasi-inference rules (2)

proving the satisfaction of a metacondition

prove : Parameter ⟼ WfRep → RepositoryE
prove.par.rep =
  is-error.rep ➔ error.rep
  let
    %art%label = par
    (ler, fsr, tsr, err) = rep
  art ≠ mec ➔ 'parameter inadequate'
  tsr.par # ? ➔ 'no metacondition to be proved'
  let
    mec = tsr.par
  satisfied.mec = ? ➔ ?
  satisfied.mec = 'NO' ➔ 'metacondition not satisfied'
  true ➔ (ler[par/mec], fsr, tsr[mec/?], err)

The theorem-prover is represented by a partial function.

satisfied : MetCon → {YES, NO}

The taxonomy of lemma-oriented actions

declare a parameter with created designator

declare-par-LC : Parameter x LemCreDen ⟼ ActDen

declare a parameter with selected designator

declare-par-LS : Parameter x Article x Parameter ⟼ ActDen

declare par. and replace its designator

declare-par-LSR : Parameter x Article x Parameter ⟼ ActDen

replace a scheme for a parameter

sch4par-L : Parameter x Parameter ⟼ ActDen

modify a declared parameter

modify-par-S : Parameter x SchCreDen ⟼ ActDen

The toolkit of our ecosystem

1. SCHEMES
   a. Creators of schemes:
      i. get the scheme associated to a parameter,
      ii. substitute a scheme for a metavariable in a scheme,
      iii. substitute a parameter for a metavariable in a scheme,
      iv. substitute the designator (of par.) for this parameter in a scheme.
   b. Scheme-oriented actions
      i. declare a new temporary parameter (versions C, S and SR)
      ii. modify the designator of a temporary parameter,
      iii. finalize a temporary parameter,
      iv. delete a temporary parameter (no deletion of final parameters!)
2. ACTIONS
   a. Creators of lemmas:
      i. get the lemma associated to a parameter,
      ii. substitute a scheme for a metavariable in a lemma,
      iii. substitute a parameter for a metavariable in a lemma,
      iv. substitute the designator (of par.) for this parameter in a lemma,
      v. … creators corresponding to all inference rules in the system.
   b. Lemma-oriented actions:
      i. declare a new lemma,
      ii. substitute the designator (of par.) for this parameter in a lemma.
      iii. delete a lemma.

More than just logical rules

First example of a metaprogram derivation

An example of a program development (1)

Program to be developed

P - program
A - axiom
L - lemma

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 (x **is** integer) **and-k** (y **is** integer) **and-k** (x=3) **and-k** (y=4)

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

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

sch4mev-S

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

A2: **error-transparent**(con) **implies** (con ⇨ **NT**) the definitional axiom of **NT**
A3: (**error-transparent**(con1) **and** (con2 ≡ **NT**)) **implies** ((con1 **and-k** con2) ⇔ con1))
A4: (con1 ≡ con2) **implies** (con1 ⇨ con2)
A5: (con1 ≡ **NT**) **implies** ((con1 **and-k** 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**)

L5 :

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

An example of a program development (5)

Step 6: the development of assignment

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

sch4mev-S

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)

elimination of @

L6 : 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)

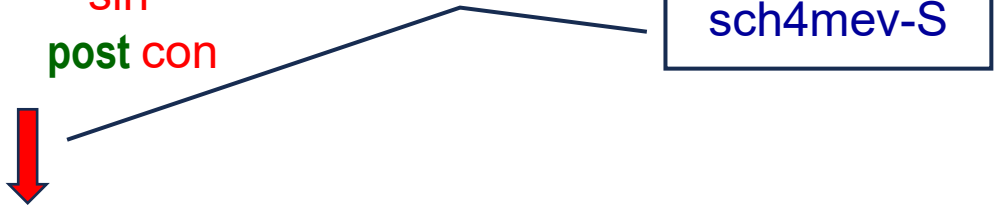
replacement of a precodition by a weakly equivalent one

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



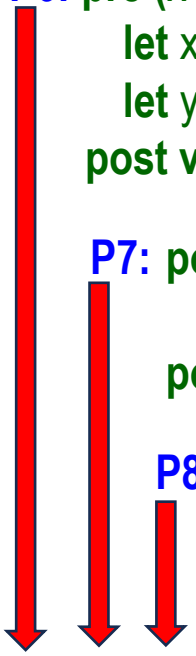
pre y := x+1 **@** **(var** x **is** integer**) and-k (var** y **is** integer**) and** (y = 4**)**
y := x+1
post (var x **is** integer**) and-k (var** y **is** integer**) and-k** (y = 4**)**

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

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)


Thank you for
your attention