Ecosystems for programmers in Lingua-V

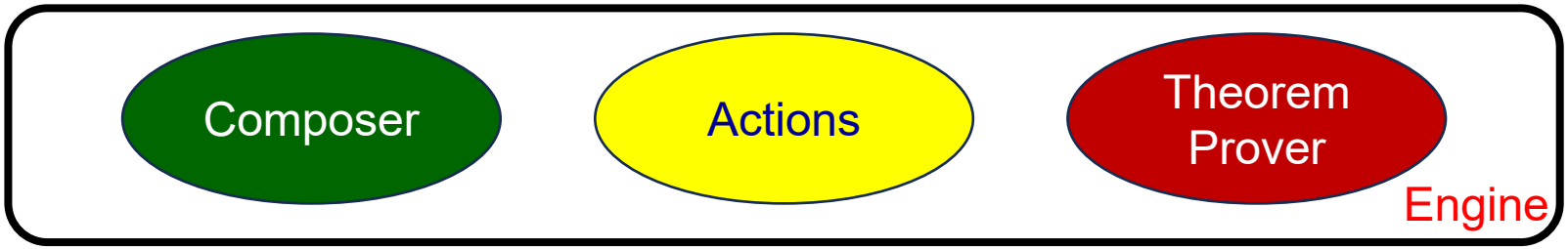
Andrzej Jacek Blikle

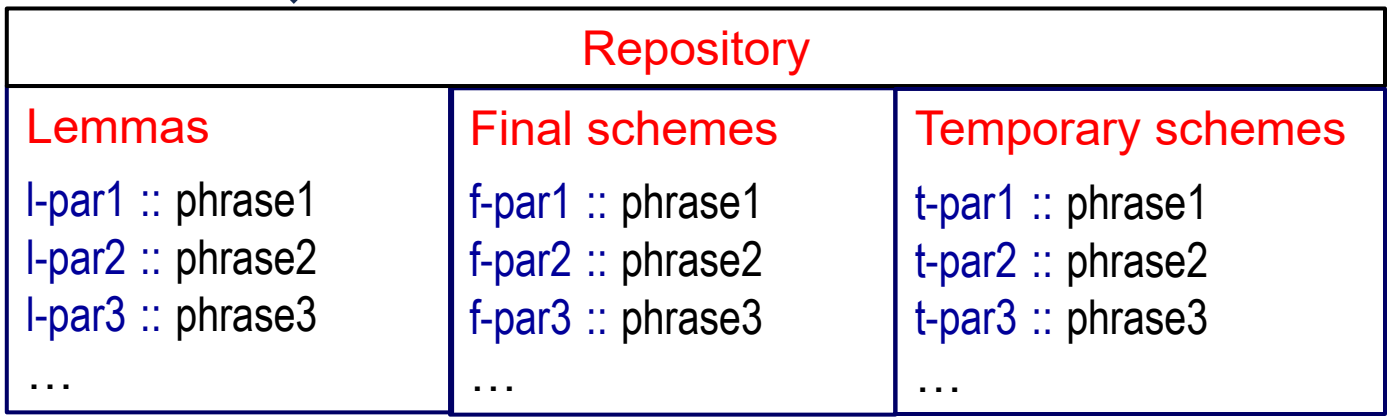
January 11th, 2026

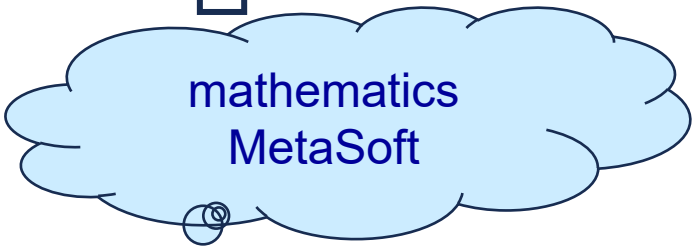
Preliminaries of ecosystems

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

finals ⊆ patterns ⊆ schemes

We need of 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. standard rules – expressible as lemmas; convenient
   c. non-standard rules – not expressible as lemmas; convenient +significant (?)

we shall concentrate on group 2.d

Two dichotomies

1. A small set of axioms (logic) versus dynamically developed 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 ecosystems

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

patterns

add to all categories:
$ art $ Label

```
art : Article =
   con   |
   mco   |
   act   |
   …
```

not a grammatical category

Lingua-S = Lingua-T + Parameters

```
vex : ValExp =
   % vex % Label
   $ vex $ Label      |
   … as in Lingua
```

schemes

add to all categories:
% art % Label

We assume that single parameters are also regarded as schemes.

The satisfaction of patterns
(a link to the theory)

By a valuation in **D-Theory** into **AlgDen-V** we mean any many-sorted function vat that assigns denotations of **Lingua-V** to metavariables:

vat.ide : {ide label | label : Label} ⟼ Identifier identity function
vat.tex : {tex label | label : Label} ⟼ TypExpDen
vat.vex : {vex label | label : Label} ⟼ ValExpDen
vat.sin : {sin label | label : Label} ⟼ SpeInsDen
…

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

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)
- (ide **is** integer) **and-k** ((ide+1) **is** integer) ⇨ ide > ide+1

The satisfaction of schemes
(a link to the theory)

A scheme is satisfied if:

- all its parameters are declared in the Repository,
- it becomes a satisfied pattern if all its parameters are replaced by their designators.

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)
```

invariant – declared on slide 4

A pattern is said to be a lemma if it can be proved (derived) in **D-Theory**.

A scheme is said to be a lemma if it becomes a lemma after the replacement of all its parameters by their designators.

Since **AlgDen-V** is a model of **D-Theory**
by Gödel's theorem
all lemmas are satisfied **AlgDen-V.**

Repositories in ecosystems

Repositories – definition

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

A parameter is said to be called in a repository, if it appears in at least one scheme or lemma of this repository.

A parameter is said to be declared in a repository, if it is declared in at least one of its subrepositories, i.e., belongs to its domain

A repository is said to be well-formed if:

1. every parameter is declared in at most one subrepository,
2. every parameter that is called in a repository is declared in it (although not necessarily vice versa)

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 justified within the ecosystem.
3. Operators may add lemmas and actions proved „outside" of the ecosystem, provided that AlgDen-V remains a model of D-Theory!

In repositories parameters are assigned to their designators:

parameter :: designator

1. and 2. guarantee that AlgDen-V remains a model of D-Theory (by Gödel's theorem)

Basic patterns of TemSchRep

Initial content of TemSchRep

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

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

Grammatical categories in grammatical clauses are replaced by corresponding metavariables.

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

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.

Repositories – examples

SchRep

```
con-mev      :: con                                  metavariable of the sort of conditions
con-less-int :: ( vex1 < vex2 )                      condition less-than for integers
sde-var      :: let loi ¥ be ¥ tex tel                specdeclaration of a variable
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)
```

LemRep

```
lem-successor-int   :: (vex is integer) and-k ((vex+1) is integer) ⇨ vex < vex + 1
lem-weak-equ-sym    :: con1 ⟺ con2 implies con2 ⟺ con1
lem-standard-arith  ::  pre (ide1, ide2 is-var-type integer) and-k (ide1 = 0) and-k (ide2 > 0):
                          while ide1 < ide2 do ide1 := ide1 + 1 od
                        post ide1 = ide2
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)
```

Lingua-E
an ecosystem seen
as a programming language

Part I: the creation of schemes

The algebra of denotations

instructions

Carriers of the algebra of denotations AlgDen-E

```
acd : ActDen       = Repository ⟼ Repository
scd : SchCreDen    = Repository ⟼ SchemeE
lcd : LemCreDen    = Repository ⟼ LemmaE
par : Parameter    = {%} Article {%} Label
mev : MetVar       = {$} Article {$} Label
```

action denotations

scheme-creator denotations

lemma-creator denotations

expressions

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:

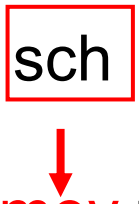
```
get-S       : Parameter                          ⟼ SchCreDen get designator
sch4mev-S   : SchCreDen x MetVar x SchCreDen     ⟼ SchCreDen scheme for mev
par4mev-S   : Parameter x MetVar x SchCreDen     ⟼ SchCreDen par for mev
sch4par-S   : SchCreDen x Parameter              ⟼ SchCreDen scheme for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

Constructors of the denotations of universal actions for schemes:

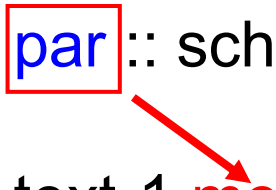
```
assign-S    : Parameter x SchCreDen              ⟼ ActDen
declare-S   : Parameter x SchCreDen              ⟼ ActDen
finalize-S  : Parameter                          ⟼ ActDen
```

Constructors of creators (2)
(substitutions in schemes)

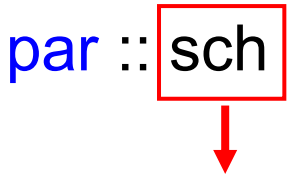
sch4mev-S:



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}
agree-sch	: MetVar x Scheme	⟼ {tt, ff}
agree-par	: MetVar x Parameter	⟼ {tt, ff}
replace-m	: Scheme x MetVar x Scheme	⟼ SchemeE
replace-p	: Scheme x Parameter x Scheme	⟼ SchemeE

parsing involved

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

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

```
get-S : Parameter ⟼ Repository ⟼ SchemeE
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 creators (3)
(substitutions in schemes)

The substitution of a scheme for a variable in a scheme.

sch4mev-S : SchCreDen x MetVar x SchCreDen ⟼ SchCreDen
sch4mev-S : SchCreDen x MetVar x SchCreDen ⟼ Repository ⟼ SchemeE
sch4mev-S.(crd1, mev, crd2).rep =

is-error.rep	➔ rep	
crd1.rep : Error	➔ rep ◀ crd1.rep	
crd2.rep : Error	➔ rep ◀ crd2.rep	
let		
sch1 = crd1.rep		
sch2 = crd2.rep		
not free.(mev, sch2)	➔ 'metavariable not free'	
not agree-sch.(mev, sch1)	➔ 'incompatibility of sorts'	(parsing involved)
true	➔ replace-s.(sch1, mev, sch2)	

Constructors of creators (4)
(substitutions in schemes)

The substitution of a parameter for a variable in a scheme.

par4mev-S : Parameter x MetVar x SchCreDen ⟼ SchCreDen
par4mev-S : Parameter x MetVar x SchCreDen ⟼ Repository ⟼ Scheme | Error
par4mev-S.(par, mev, scd).rep =

is-error.rep	➔ rep
get.par.rep : Error	➔ rep ◄ 'parameter undeclared'
scd.rep : Error	➔ rep ◄ scd.rep
let	
sch = scd.rep	
not free.(mev, sch)	➔ 'metavariable not free'
not agree-sch.(mev, par)	➔ 'incompatibility of sorts'
true	➔ replace.(sch, mev, par1)
	(note that a par1 is a scheme itself)

Parameter substituted for a metavariable must be declared!

Constructors of creators (5)
(substitutions in schemes)

The substitution of a designator for a parameter in a scheme.

sch4par-S : SchCreDen x Parameter ⟼ SchCreDen
sch4par-S : SchCreDen x Parameter ⟼ Repository ⟼ SchemeE

```
sch4par-S.(scd, par).rep =
  is-error.rep              ➔ rep
  scd.rep : Error           ➔ rep ◄ scd.rep
  let
    (ler, fsr, tsr, err) = rep
  fsr.par # ?               ➔ parameter not declared
  let
    sch1 = fsr.par
    sch2 = scd.rep
  not is-in.(par, sch2)     ➔ 'parameter not in the scheme'
  true                      ➔ replace-p.(sch1, par, sch2)
```

For a parameter we may substitute only its designator!

Assignment actions
(assign a created scheme to a parameter)

```
assign-S : Parameter x SchCreDen ⟼ ActDen
assign-S : Parameter x SchCreDen ⟼ Repository ⟼ Repository
assign-S.(par, scd).rep =
  is-error.rep              ➔ rep
  get.par.rep : Error       ➔ rep ◄ 'parameter not declared'
  scd.rep : Error           ➔ rep ◄ scd.rep
  let
    (ler, fsr, tsr, err) = rep
  ler.par # !               ➔ 'modyfied parameter must not point to a lemma'
  fsr.par # !               ➔ 'modyfied parameter must not point to a final scheme'
  let
    sch = scd.rep
  not agree.(par. sch)      ➔ 'incompatibility of sorts'
  true                      ➔ (ler, fsr, tsr[par/sch], err)
```

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

Declaration actions
(declare a created scheme as a parameter)

```
declare-S : Parameter x SchCreDen ⟼ ActDen
declare-S : Parameter x SchCreDen ⟼ Repository ⟼ Repository
declare-S.(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
    sch = scd.rep
  not agree.(par. sch)      ➔ 'incompatibility of sorts'
  true                      ➔ (ler, fsr, tsr[par/sch], err)
```

Parameters to be declared must not be assigned.

Finalization actions
(finalize a declared parameter)

```
finalize-S : Parameter ⟼ ActDen
finalize-S : Parameter ⟼ Repository ⟼ Repository
finalize-S.par.rep =
  is-error.rep   ➔ rep
  let
    (ler, fsr, tsr, err) = rep
  tsr.par # ?    ➔ 'parameter must be temporary'
  let
    sch = tsr.par
  true           ➔ (ler, fsr[par/sch], tsr[par/?], err)
```

Parameters to be finalized must be stored in temporary repository
and are moved to final repository.

Lingua-E
an ecosystem seen
as a programming language

Part II: the creation of lemmas
(i.e., metaprograms in particular)

Constructors of lemma-creators' denotations

lcd : LemCreDen = Repository ⟼ LemmaE

Universal L-constructors (our inference rules):

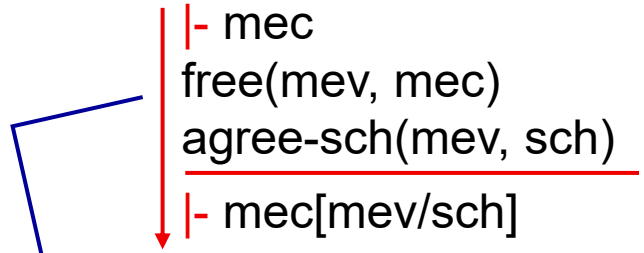
get-L	: Parameter	⟼ LemCreDen get designator
sch4mev-L	: SchCreDen x MetVar x LemCreDen	⟼ LemCreDen scheme for mev
par4mev-L	: Parameter x MetVar x LemCreDen	⟼ LemCreDen par for mev
sch4par-L	: LemCreDen x Parameter	⟼ LemCreDen scheme for par
detach-L	: Parameter x Parameter	⟼ LemCreDen

…

Justification of soundness

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

Definitions analogous as for S-constructors



The rule of substitution

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

Constructors of lemma-creators' denotations

lcd : LemCreDen = Repository ⟼ LemmaE

Lingua-V-oriented 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
…

Constructors of lemma-creators' denotations
(an example)

The strengthening of a precondition (a half-formal definition):

```
strengthen-pre : Parameter x Parameter ⟼ LemCreDen
strengthen-pre : Parameter x Parameter ⟼ Repository ⟼ LemmaE
strengthen-pre.(par-str, par-mpr).rep
  is-error.rep                                  ➔ error.rep
  let
     (ler, fsr, tsr, err) = rep
  ler.par-str # ?                               ➔ 'no such stronger-than lemma'
  ler.par-mpr # ?                               ➔ 'no such metaprogram'
  let
    str-lem   = ler.par-str
    mpr-lem = let.-par-mpr
  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

First example
of a metaprogram derivation

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 ;
post (x is integer) and-k (y is integer) and-k (x=3) and-k (y=4)
```

P - program
A - axiom
L - lemma

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)

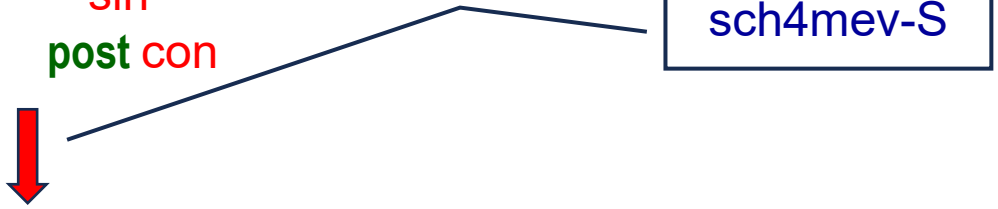
replacement of a precodition by a weakly equivalent one

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



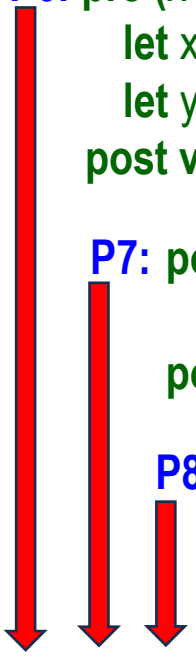
pre y := x+1 **@** **(var** x **is** integer**) and-k (var** y **is** integer**) and** (y = 4**)**
y := x+1
post (var x **is** integer**) and-k (var** y **is** integer**) and-k** (y = 4**)**

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