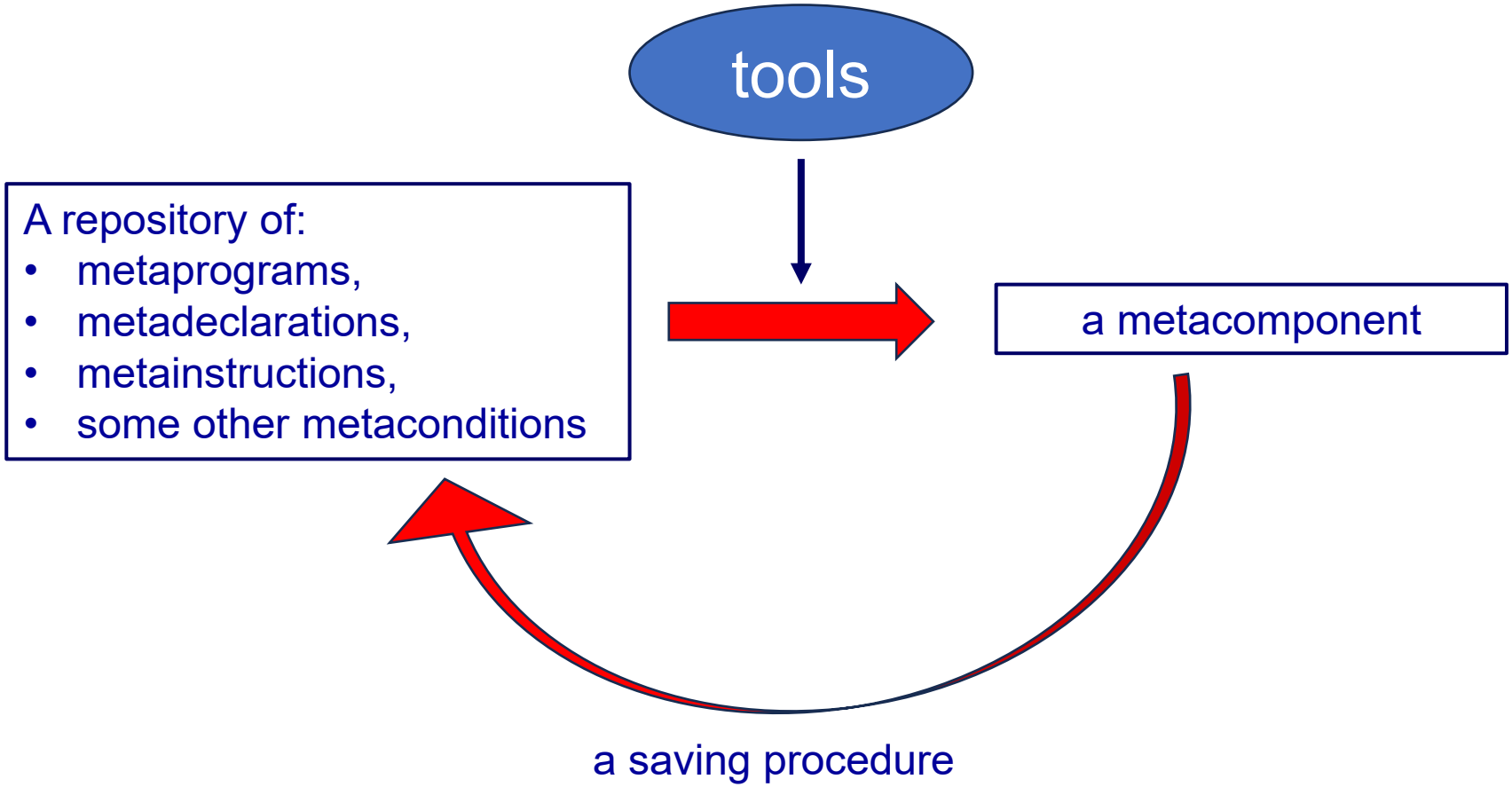
Investigations on ecosystems for Lingua-V programmers

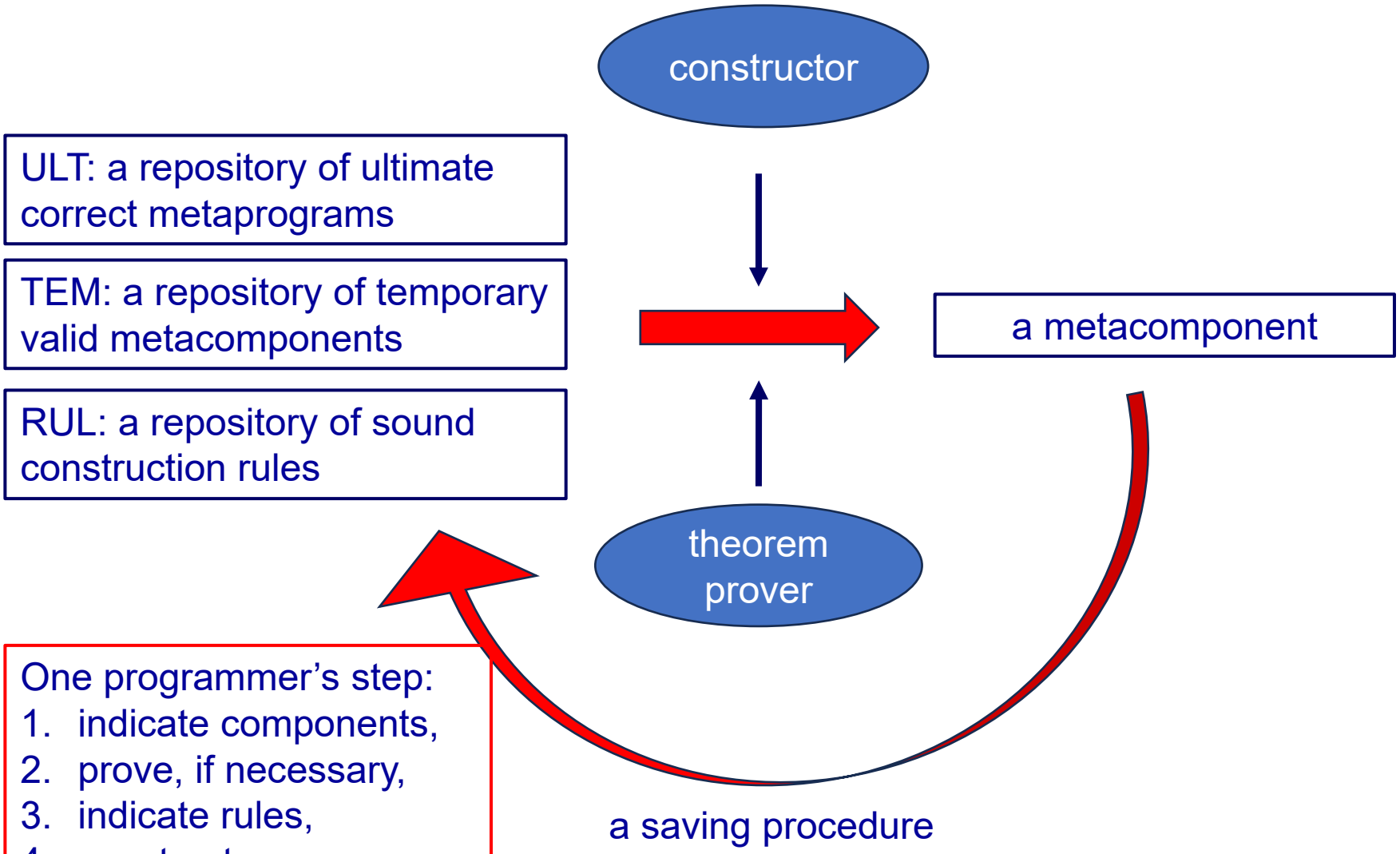
Andrzej Jacek Blikle

August 9th, 2025

Programs' development cycle in Lingua-V



A closer look to programs' development cycle



One programmer's step:

1. indicate components,
2. prove, if necessary,
3. indicate rules,
4. construct,
5. save

Examples of theorems to be proved

x **is** integer ⇨ x < x + 1

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

largest integer in the implementation

We shall not need to prove the correctness of metaprograms!

Correct metaprograms will be developed.

An example of a program development (1)

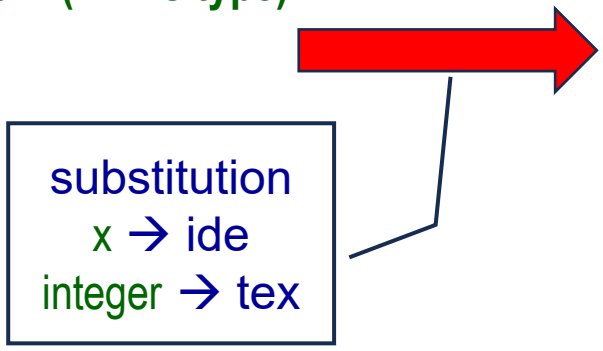
Program to be developed

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

Step 1: synthesize the declaration of x — RPM(1), RCM(2)

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

a rule in RUL




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

An example of a program development (2)

Step 2: remove tautology — RPM(4), RTM(1), RCM(1)

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

Rules to be applied:

- integer **is type ≡ NT**
- (x **is free**) **is error transparent** derived from (ide **is free**) **is error transparent**
- ((x **is free**) **and-k NT**) **≡** (x **is free**) derived from
  con **is error transparent implies** ((con **and-k NT**) **≡** con)

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 – RPM(8), RTM(2), RCM(4)

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

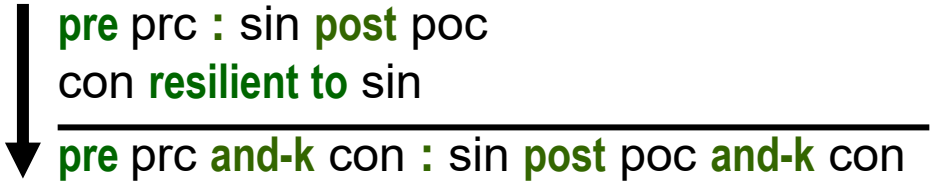
P4 : pre (x **is free**) **and-k** (y **is free**)
let x **be** integer **tel**
post var x **is** integer **and-k** (y **is free**)

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**)

Rules to be applied:

- ide-1 ≠ ide-2 **implies** ((ide-1 **is free**) **is resilient to** (**let** ide-2 **be** tex)),
- ide-1 ≠ ide-2 **implies** ((ide-1 **is** tex-1) **is resilient to** (**let** ide-2 **be** tex-2)),



An example of a program development (4)

Step 5: sequential composition — RTM(2), RUL(1)

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**)

Rule to be applied:

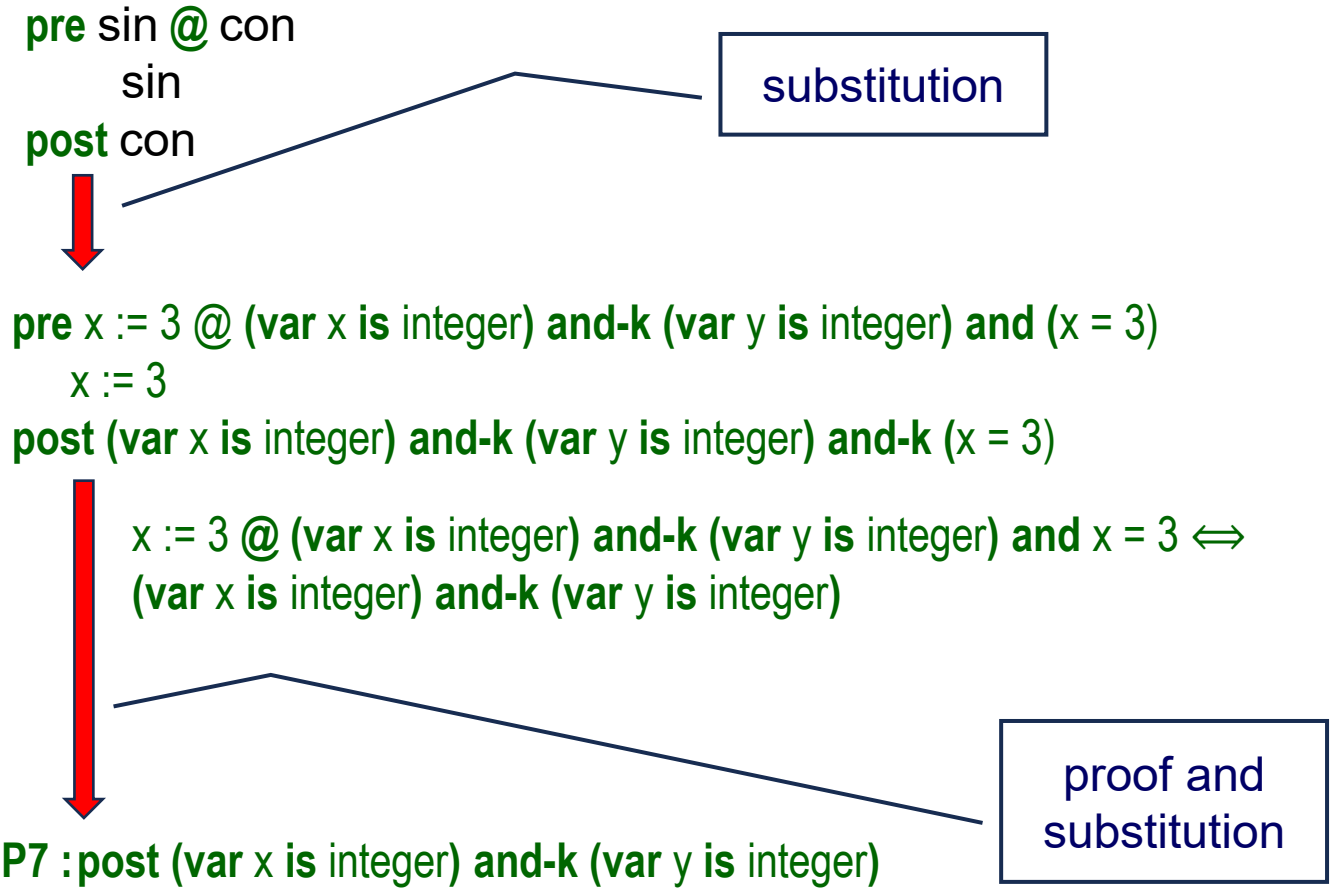
pre prc-1**:** spr-1 **post** poc-1
pre prc-2**:** spr-2 **post** poc-2
poc-1 ⇨ prc-2

---

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

An example of a program development (5)

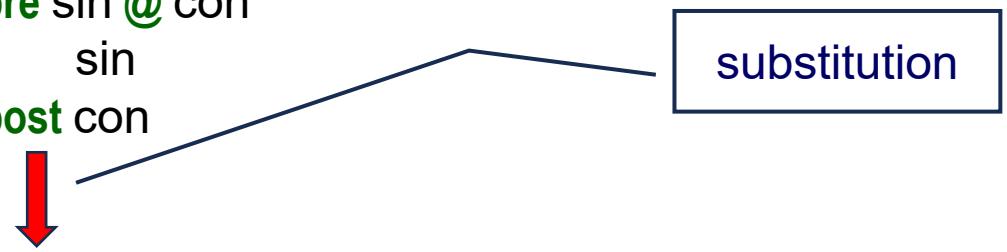
Step 6: the development of assignment — RPM(1), RUL(2)



An example of a program development (6)

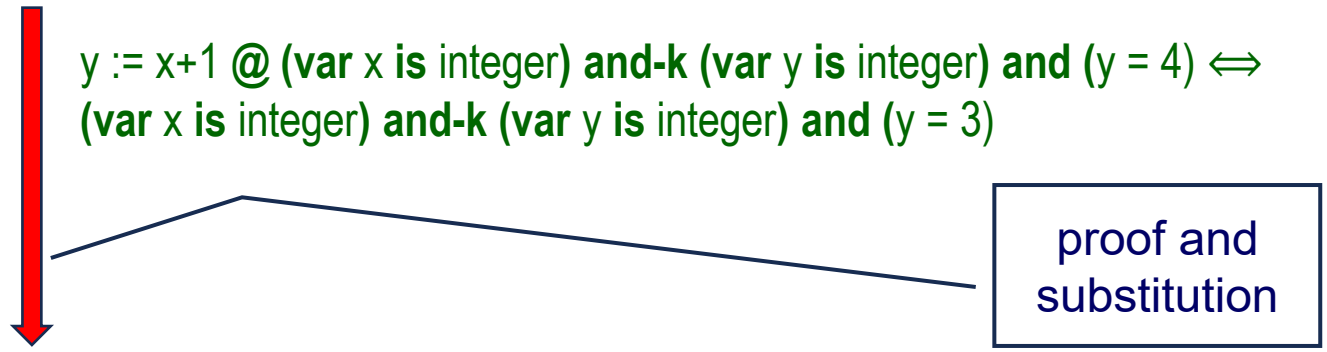
Step 7: the development of assignment — RPM(1), RUL(1)

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 – RTM(3), RUL(2)

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

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

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

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

An example of a program development (8)

Step 9: the development of an assignment — RPM(1), RUL(2)

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

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

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

theorem prover:
8 < 10

Rule to be applied:

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

An example of a program development (8)

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

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

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

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

target program

Numbers of referencings:
RPM – 16
RTM – 10
RUL – 16 total 42
TPR – 1

The need of a formalized theory

We need a formalized theory rich enough to prove lemmas
and the soundness of program-construction rules
in the course of program development in **Lingua-V**

We shall call the language of such a theory
Lingua-FT

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

1. Extending the denotational model of **Lingua** to **Lingua-V**:
   a. extending the corresponding equational grammar and the corresponding algebras, i.e., **AlgSyn** to **AlgSyn-V**
   b. enriching **AlgDen** to **AlgDen-V**.
2. Building a denotational model of **Lingua-FT**:
   a. extending **AlgSyn-V** to **AlgSyn-FT;** working with grammars,
   b. deriving an algebra of denotations **AlgDen-FT** as an enrichment of **AlgDen-V**.
3. Building 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.

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

In first-order theories we talk about:

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

A language of first-order theories includes two syntactic categories

terms — represent functions
formulas — represent predicates

names and separators will be printed in green

variables will be printed in black

Primitives of syntax

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

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

A recollection of formalized theories (2)
First-order theories – the language

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

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

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

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

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

Language

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

with

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

Examples of formulas

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

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

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

Axioms

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

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

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

An interpretation of a language of a formalized theory:

Int = (Uni, F, P)

with

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

F – function; F[fn] : $\text{Uni}^{cn} \longmapsto$ Uni for arity.fn = n
F[fn] : $\longmapsto$ Uni for arity.fn = 0

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

A valuation is a total function that assigns primitive elements 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 (6)
Interpretation and semantics (2)

The semantics of variables:

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

The semantics of terms:

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

The semantics of formulas:

SF : Formula ⟼ Valuation ⟼ Bool

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

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

For a given interpretation:

Int = (Uni, F, P)

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

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

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

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

A |- for

if it is satisfied in every model of this theory.

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

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

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

The main deduction rules

Rule of substitution

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

x free in for(x)

ter – an arbitrary term

Rule of detachment

$$\frac{A \models for\text{-}1 \qquad A \models for\text{-}1 \rightarrow for\text{-}2}{A \models for\text{-}2}$$

Rule of generalization

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

x free in for(x)

Gödel's completeness theorem

In every first-order theory with axioms A

A |- for iff A |= for

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

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

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

Three models of Peano arithmetic:

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

standard model

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

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

Second-order Peano's arithmetics:

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

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

Two metatheorems:

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

Proof of 2. by induction:

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

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

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

Gödel's incompleteness theorem

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

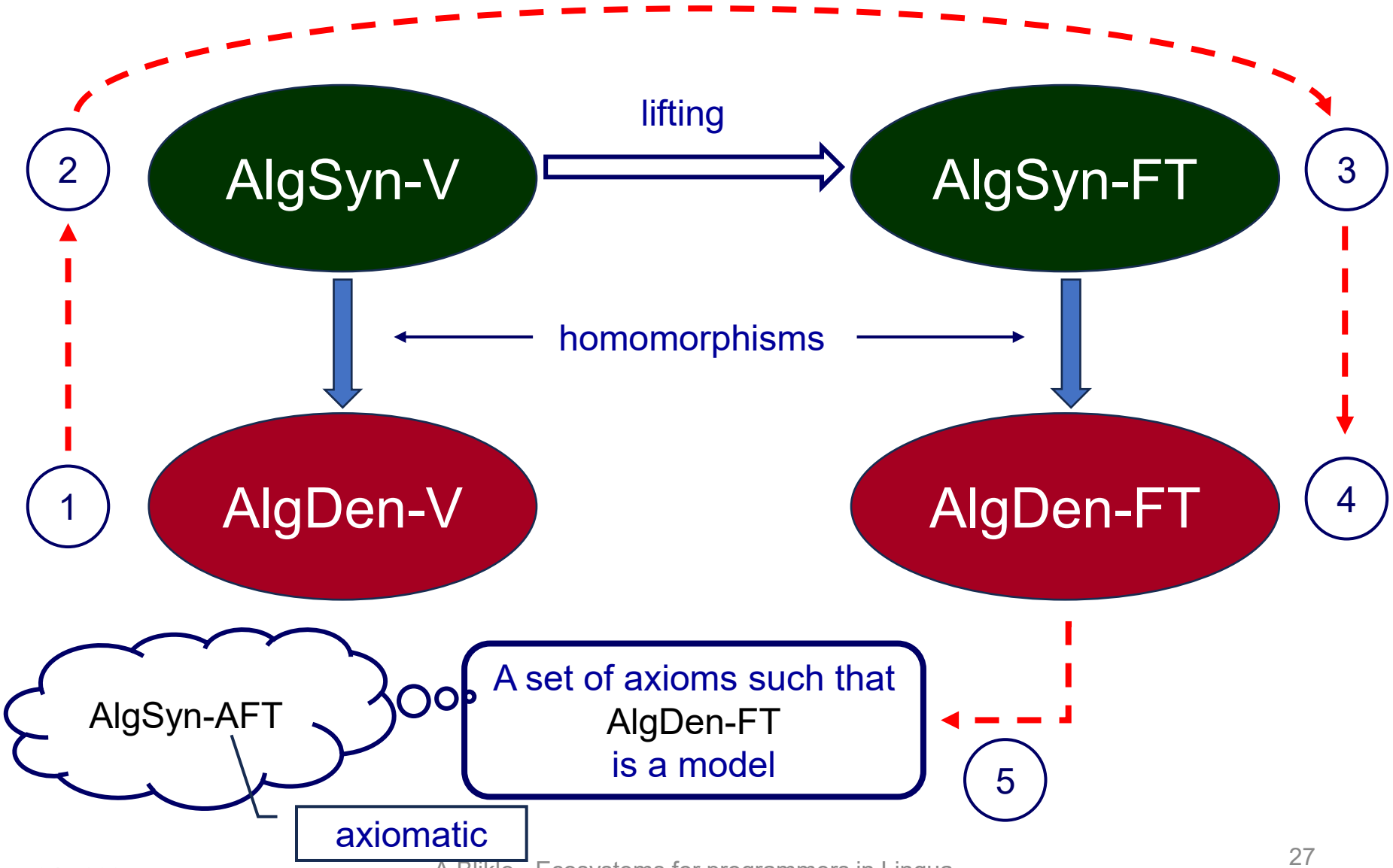
Gödel's adequacy theorem

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

we can trust the theorems that have been proved

Building Language-FT for Language-V (1)

FT – formal theory



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

Source algebra (Language-V)

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

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

Assumptions on **AlgDen-V**:

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

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

Three algebras to be derived:

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

Building Language-FT for Language-V (4)
From AlgDen-V to AlgSyn-V

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

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

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

Building Language-FT for Language-V (3)
From AlgDen-V to AlgSyn-V introducing variables

Sets of variables:

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

Sorts and arities of variables:

sort : IndVar ⟼ Cn-V

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

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

Sorted domains of variables:

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

Term- and formula-creating functions

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

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

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

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

⇒ mk-term-cn(IndVar.cn) |

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

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

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

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

for : Formula =

	mk-formula(IndVar.boo)	
	pn(Term.cn-1,…,Term.cn-n)	for all pn : PreNam with arity.pn = (cn-1,…,cn-n)
	prv(Term.cn-1,…,Term.cn-n)	for all pv : PreVar with arity.pv = (cn-1,…,cn-n)
	and(Formula, Formula)	
	or(Formula, Formula)	
	implies(Formula, Formula)	
	not(Formula)	
	($\forall i$ IndVar) Formula	
	($\exists i$ IndVar) Formula	
	($\forall f$ FunVar) Formula	
	($\exists f$ FunVar) Formula	
	($\forall p$ PreVar) Formula	
	($\exists p$ PreVar) Formula	

instead of Term.boo

AlgSyn-FT is implicit in this grammar

Building Language-FT for Language-V (8)
The common signature of AlgSyn-FT and AlgDen-FT

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

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

Building Language-FT for Language-V (7)
From AlgSyn-FT to AlgDen-FT

Valuations

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

Well-formed valuations

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

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

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

Domains of denotations

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

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

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

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

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

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

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

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

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

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

Building Language-FT for Language-V (9)
Defining the interpretation funDen-FT of functional symbols (2)

(5) Formula-making function mk-formula

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

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

(8) Conjunction

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

interpretation from Language-V

(9) General quantifier

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

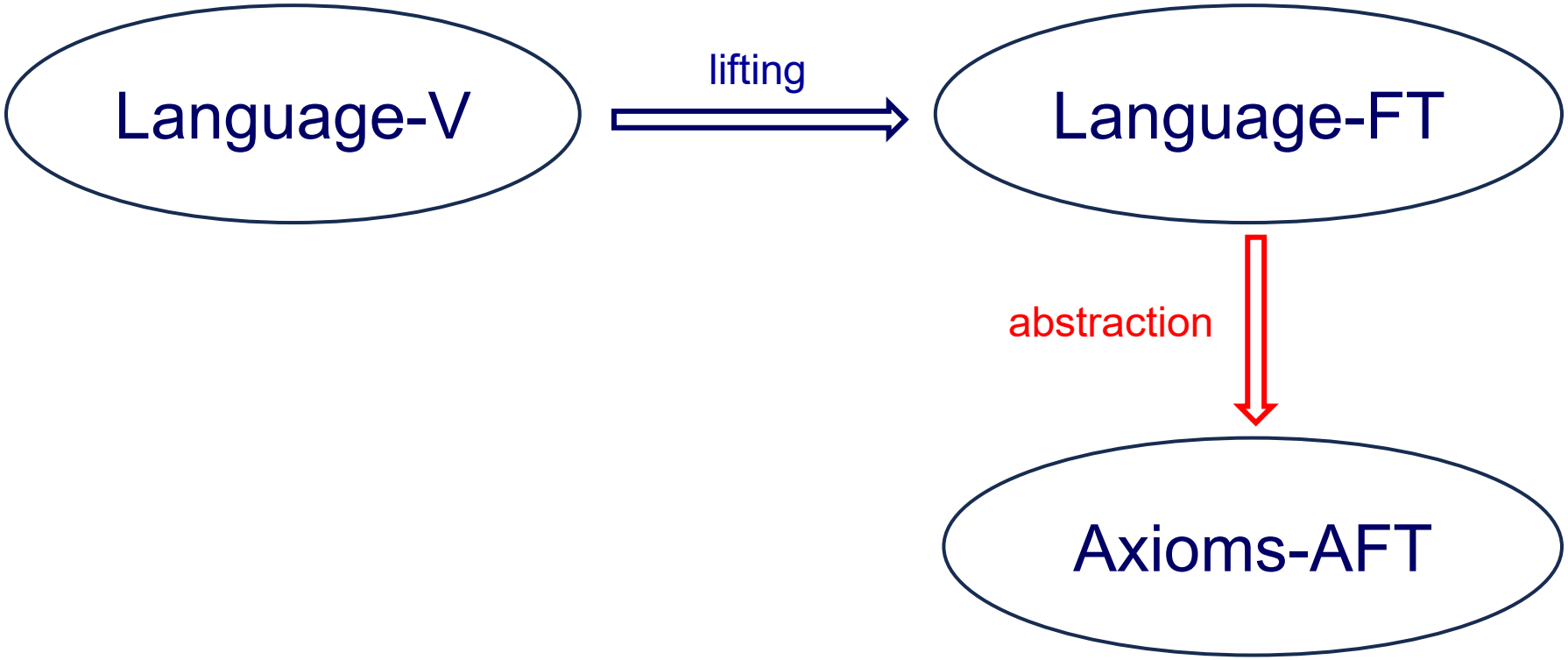
Building Language-FT for Language-V (10)

How theories and models are built

mathematical logic: axioms ➔ its models

working mathematician: a model ➔ its axioms

our case



Building Language-FT for Peano Arithmetics (1)

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

constants are printed in green

Carriers

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

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

Functions

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

Grammar

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

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

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

Building Language-FT for Peano Arithmetics (2)

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

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

Grammar

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

no syntactic cathegory of 2nd-order variables!

variables are printed in black

a formula with 2nd-order variable

Building Language-FT for Peano Arithmetics (3)

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

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

The domains of denotations:

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

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

An example of function's interpretation

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

Formalizing Lingua-V (1)
syntax – new categories

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

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

Formalizing Lingua-V (2)
syntax – conditions

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

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

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

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

mco : MetCon-V =

relational metaconditions

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

behavioral metaconditions

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

temporal metaconditions

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

language related metaconditions

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

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

mco : MetCon-V = (cont.)

metaprograms

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

compound metaconditions with classical operators

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

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

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

mco-not(MetCon-V)

Formalizing Lingua-V (5)
denotations

New carriers

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

Signatures of constructors (examples)

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

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

Individual variables – for all cn : Cn-V

IdeVar-FT	= ide	| ide-1	|…	variables corresponding to identifies,
ValExpVar-FT	= vex	| vex-1	|…	variables corresponding to value-expressions,
RefExpVar-FT	= rex	| rex-1	|…	variables corresponding to ref.-expressions,
SpeInsVar-FT	= sin	| sin-1	|…	variables corresponding to specinstructions,
ConVar-FT	= con	| con-1	|…	variables corresponding to conditions,
MetConVar-FT	= mco	| mco-1	|…	variables corresponding to metaconditions.

…

Second-order variables:

vacat (so far)

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)         |
  ...
```

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-make-con(ConVar-FT   )          | variables
        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))
vex-less(y, 0)
sin-while(vex-less-int(x, 0), sin-assign(x, vex-add-int(a, 1)))
sin-skip-ins
```

examples of free terms

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

Lifting Lingua-V to Lingua-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

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

```
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

```
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
```

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
then vlu.vex : ValExpDen-V

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

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

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

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

Program construction rules (1st-order axioms)

1. Rules independent of **Lingua-V**:
   a. formulas describing properties of metarelations
   b. definitional formulas of behavioral metaconditions including the correctness of metaprograms,
2. Rules dependent of **Lingua-V**:
   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-V** (examples)

Dependencies between metapredicates

(con1 ≡ con2**)** **iff** **((**con1 ⊑ con2**)** **and** **(**con2 ⊑ con1**))**
(con1 ⇔ con2**)** **iff** **((**con1 ⇨ con2**)** **and** **(**con2 ⇨ con1**))**
(con1 ≡ con2**)** **implies** **(**con1 ⇔ con2**)**
(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 (at-ide is free) and-k (cl-ide is class) and-k (tex is type) :
   let at-ide be tex with yex as pst tel in cl-ide
post att at-ide is tex with yex in cl-ide as pst
```

Rule for adding a type constant

```
pre (tc-ide is free) and-k (cl-ide is class) and-k (tex is type) :
   set tc-ide be tex tes in cl-ide
post tc-ide 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

The logistics of program development (1)
a denotational model of our ecosystem

Three qualified repositories:

ULT : repository of ultimate correct metaprograms – ground formulas,
TEM : repository of temporary valid formulas – ground and free formulas,
RUL : repository of sound construction rules – free formulas.

mcn : MetConNam = …
rna : RepNam = {ULT, TEM, RUL}

ult : Ultimate = MetConNam ⟹ ValGroMetCon-FT
tem : Temporary = MetConNam ⟹ ValGroMetCon-FT | ValFreMetCon-FT
rul : Rule = MetConNam ⟹ ValFreMetCon-FT

ValGroMetCon-FT – valid ground metaconditions
ValFreMetCon-FT – valid free metaconditions

rep : QuaRep = Ultimate | Temporary | Rule
rep : Repository ⊆ RepNam ⟹ QuaRep

repositories are case-adequate:

rep.ULT : Ultimate
rep.TEM : Temporary
rep.RUL : Rule

The logistics of program development (2)
repositories – indicators (1)

Indicators are used to identify metaconditions in repositories

ind : Indicator = RepNam x MetConNam

get : Indicator ⟼ Repository ⟼ MetCon-FT | Error
get.ind.rep =
  let
    (rna, mcn) = ind
  rep.rna.mcn = ? ➔ 'no such metacondition'
  true ➔ rep.rna.mcn

The logistics of program development (3)
repositories – indicators (2)

```
store : Indicator x MetCon-FT ⟼ Repository ⟼ Repository | Error
store.(ind, mec).rep =
  let
    (rna, mcn) = ind
  get.ind.rep /: Error  ➔ 'target name must not be assigned'
  ground.mec            ➔                          is a ground metacondition
    rna = RUL ➔ 'ground formula can't be stored as rule'
    let
      new-qre = (rep.rna)[mcn/mec]
    true          ➔ rep[rna/new-qre]
  true                  ➔
    rna = ULT ➔ 'free formula can't be stored as ultimate'
    let
      new-qre = (rep.rna)[mcn/mec]
    true          ➔ rep[rna/new-qre]
```

The logistics of program development (4)
actions (1)

Actions are procedures that given some parameters return repository-modification functions.

Action = Parameter ⟼ Repository ⟼ Repository | Error

Three categories of actions:

- actions based on general inference rules,
- actions based on specific inference rules,
- actions activating a built-in theorem prover,
- suggesting substitutions.

Expected scenario of metaprogram development:

- ❑ programmers start their work with:
  - empty repository TEM,
  - non-empty repositories ULT and RUL
- ❑ programmers' activity = the execution of actions.

The logistics of program development (5)
inference actions – substitutions

var : Variable-FT = IdeVar-FT | ValExpVar-FT | RefExpVar-FT | …
ter : Term-FT = ValExp-FT | SpeIns-FT | Ide-FT | …
sup : SubPar = (Variable-FT x Term-FT)$^{c+}$ substitution parameters

substitute-to-ground : Indicator x SubPar x Indicator ⟼ Repository
⟼ Repository | Error

source

target

Rule of substitution

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

substitute-to-free : Indicator x SubPar x Indicator ⟼ Repository
⟼ Repository | Error

The logistics of program development (6)
inference actions – detachments

Rule of detachment

$$\frac{A \models \text{for-1} \qquad A \models \text{for-1} \rightarrow \text{for-2}}{A \models \text{for-2}}$$

detach-to-ground : Indicator x Indicator x Indicator ⟼ Repository
⟼ Repository | Error

detach-to-free : Indicator x Indicator x Indicator ⟼ Repository
⟼ Repository | Error

The logistics of program development (7)
inference actions – an example of an application

A task of a programmer (ground metaconditions in the example!)

replace **pre** prc1 **:** spr **post** poc by **pre** prc2**:** spr **post** poc
while prc1 ⇔ prc2

Steps to complete the task:

1. Identify **pre** prc1**:** spr **post** poc in ULT | TEM
2. Identify prc1 ⇔ prc2 in TEM
3. Identify rule in RUL
4. Substitute in the rule to match 1. and 2.
5. Execute detachment action
6. Store **pre** prc2**:** spr **post** poc

the only manual activity of programmers

the implication of detachment

pre con1 **;** spr1 **post** con2
con1 ⇔ con3
---
pre con3 **;** spr1 **post** con2

prc1 → con1
spr → spr1
prc2 → con3
poc → con2

ground (concrete) formulas

FT-variables

Rule of detachment

A |= for-1
A |= for-1 → for-2

A |= for-2

The logistics of program development (8)
specific actions – Lingua-V-oriented

An informal descriptions of actions

The removal of an assertion

head **; asr** con **rsa ;** tail

---

head **;** tail

The replacement of an assertion

head **; asr** con1 **rsa ;** tail

con1 ⇔ con2

---

head **; asr** con2 **rsa ;** tail

These actions can't be defined as an application of
inference actions to metaprogram construction rules
since scripts like
head **; asr** con **rsa ;** tail
are not in MetCon-FT.
They are not metaconditions!

The logistics of program development (9)
prover-activation action

prove-ground : MetCon-FT x Indicator ⟼ Repository → Repository | Error
prove-ground.(mec, ind).rep =
  let
    (rna, mcn) = ind
  rna = RUL ➔ 'target repository must be ULT or TEM'
  let
    mec = get.ind.rep
  valid.mec = ? ➔ ?
  valid.mec = 'NO' ➔ 'metacondition not valid'
  true ➔ store.(ind, mec).rep

in this definition:

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


Thank you for
your attention