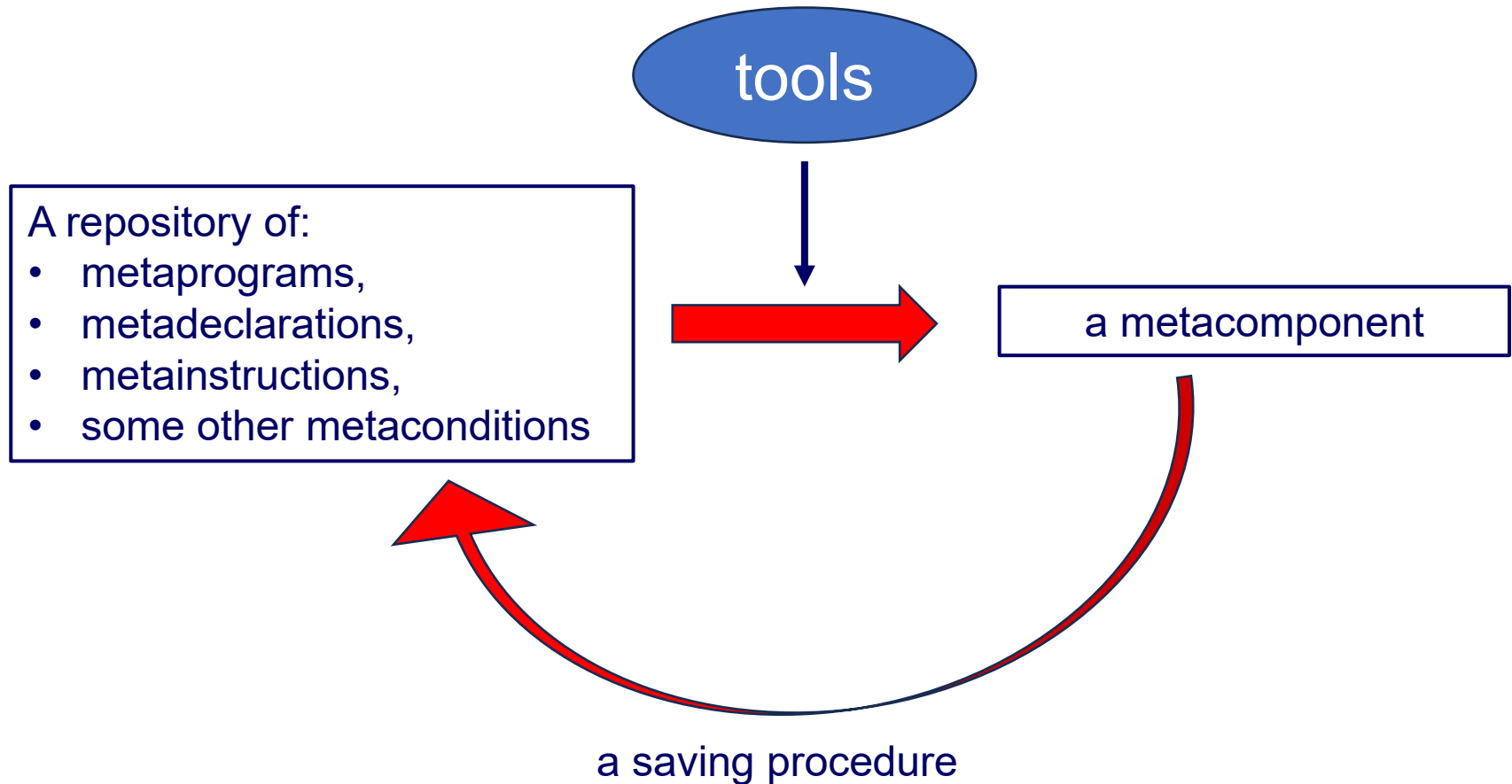


Preliminary thoughts on ecosystems for Lingua programmers

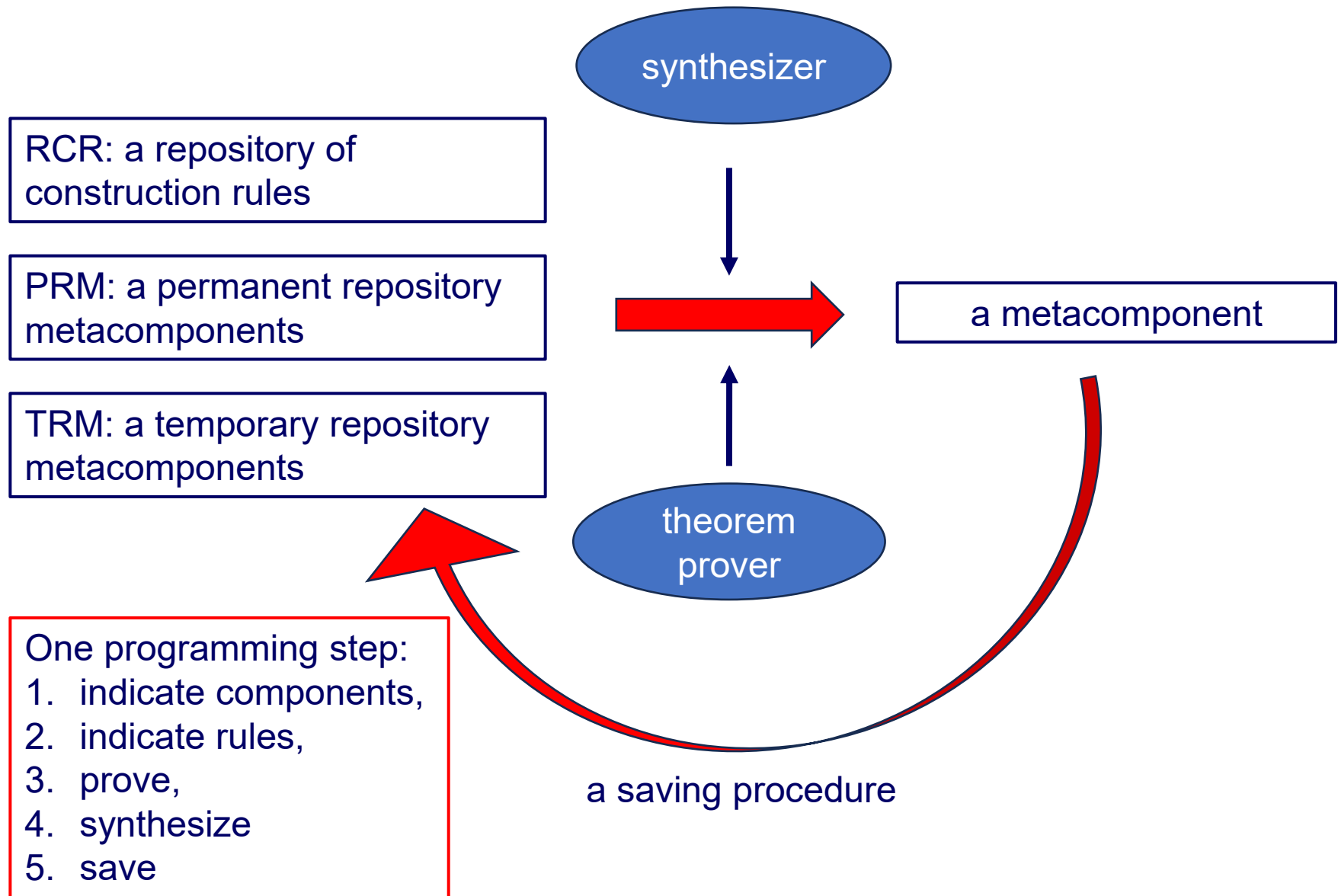
Andrzej Jacek Blikle

July 19th, 2025

Programs' development cycle in Lingua-V



A closer look to programs' development cycle



Examples of theorems to be proved

$x \text{ is integer} \Rightarrow x < x + 1$

$(x+1 \leq \text{isrt}(n))$

\equiv

$((x+1)^2 \leq n)$

whenever $(x, k \text{ is integer})$ **and-k** $(x, y \geq 0)$ **and-k** $((\text{isrt}(n)+1)^2 \leq M)$ **and-k** $(x \leq \text{isrt}(n))$

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

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

a rule in RCR

substitution
x → ide
integer → tex

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

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** \equiv **NT**
- **(x is free) is error transparent** derived from **(ide is free) is error transparent**
- **((x is free) and-k NT) \equiv (x is free)** derived from
 con is error transparent implies ((con and-k NT) \equiv con)

pre prc : sin post poc
prc \Leftrightarrow 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)

Rules to be applied:

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

pre prc : sin **post** poc
con **resilient to** sin

pre prc **and-k** con : sin **post** poc **and-k** con

An example of a program development (4)

Step 5: sequential composition

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

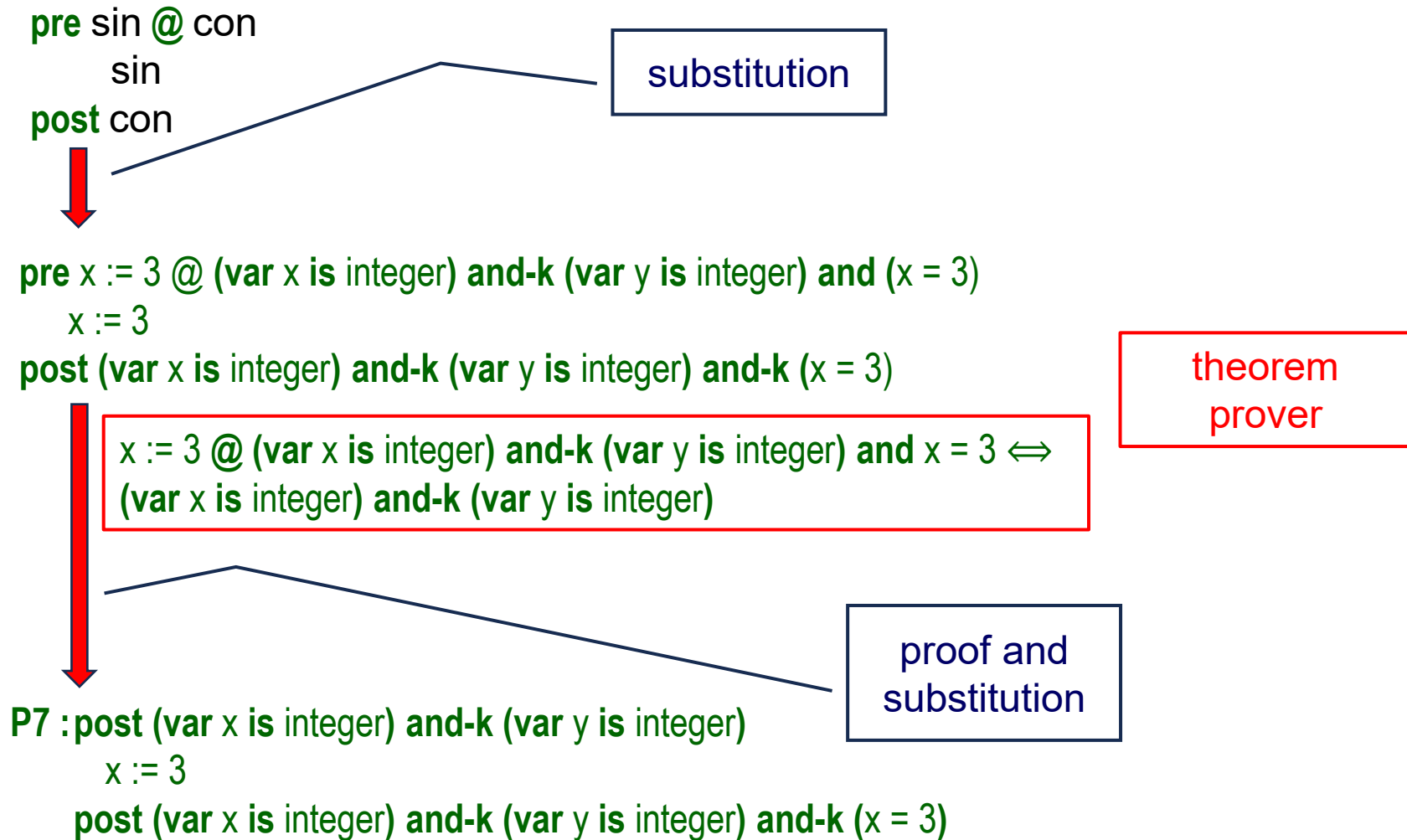
pre prc-2: spr-2 **post** poc-2

poc-1 \Rightarrow prc-2

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

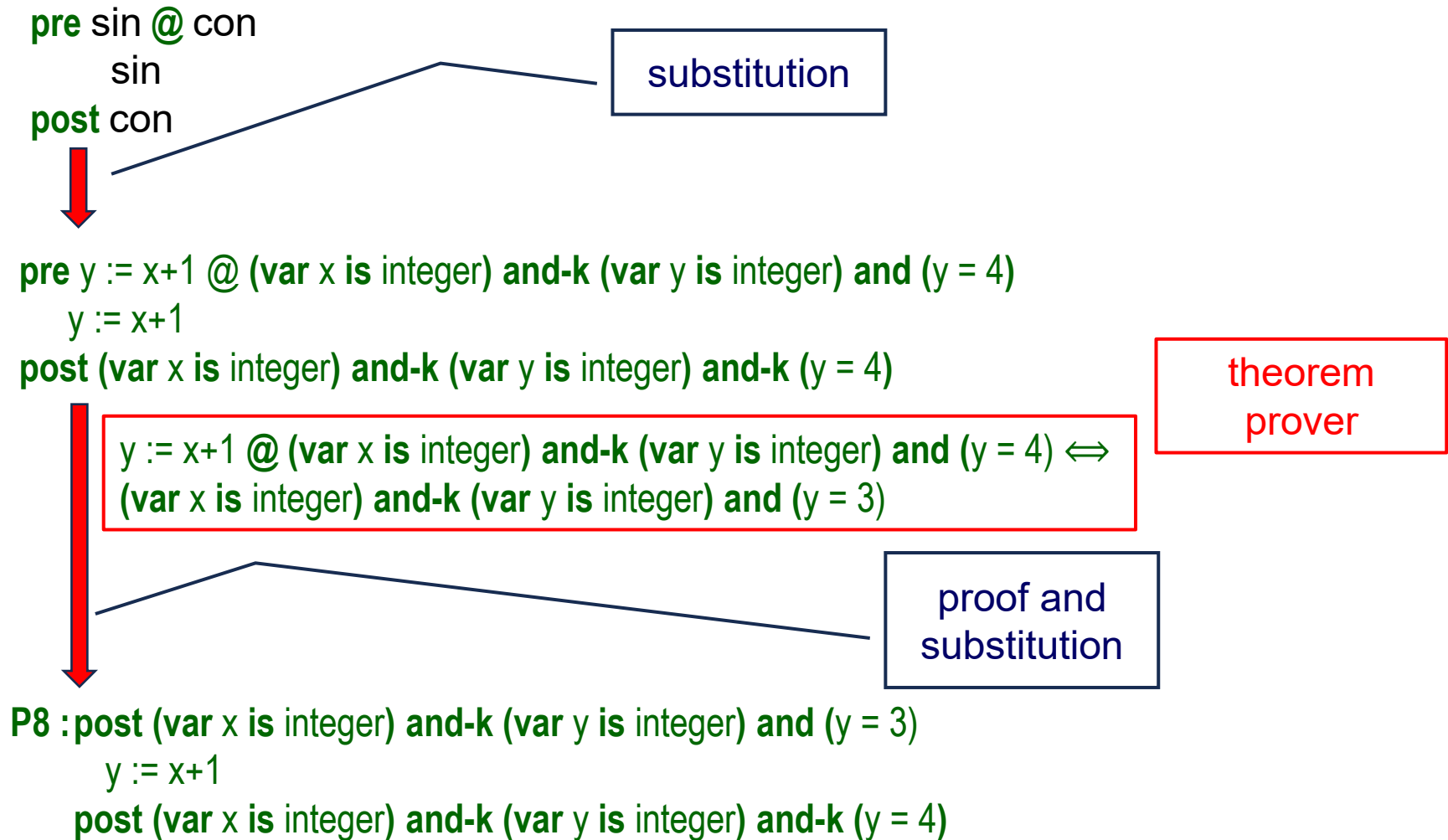
An example of a program development (5)

Step 6: the development of assignment



An example of a program development (6)

Step 7: the development of assignment



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)

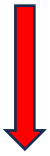
let x be integer tel

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)



An example of a program development (8)

Step 9: the development of an assignment

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) \Rightarrow (var x is integer) **and-k** (x < 10)

theorem
prover

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)

Rule to be applied:

pre prc: spr **post** poc

poc \Rightarrow prc-1

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

An example of a program development (8)

Step 9: sequential composition

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**

The need of a formalized theory

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

We shall call it a **M-theory (Master Theory)** and its language – a **M-language**

Our way to M-theory

1. Building an abstract denotational framework of a language of a formalized theory:
 - a. building an equational grammar,
 - b. building the algebras of syntax and denotations and a corresponding function of semantics.
2. Building a denotational framework of **Language-FT**:
 - a. building an equational grammar as an extension of **Lingua-V** grammar,
 - b. building an algebra of syntax as an extension of **Lingua-V** syntactic algebra,
 - c. deriving an algebra of denotations from **Lingua-V** denotational algebra.
3. Building an axiomatic framework for **Language-FT**:
 - a. defining a standard interpretation,
 - b. defining a set of axioms for which the standard interpretation constitutes a model.

A recollection of formalized theories (1)

First-order theories

In first-order theories we talk about:

$\text{ele} : \text{Uni}$ — **elements** of a set called a **universe**
 $\text{fu} : \text{Uni}^{\text{cn}} \mapsto \text{Uni}$ — **functions** with $n \geq 0$
 $\text{pr} : \text{Uni}^{\text{cn}} \mapsto \text{Bool}$ — **predicates** with $n \geq 0$

A language of first-order theories includes two syntactic categories

terms — represent functions
formulas — represent predicates

Primitives of syntax

$\text{var} : \text{Variable}$ — **variables** (running over Uni)
 $\text{fn} : \text{Fn}$ — function names
 $\text{pn} : \text{Pn}$ — predicate names
 $\text{sep} : \text{Separator}$ — separators, e.g.: „ (” , „) ” , „ , ...
 $\text{Alphabet} = \text{Variable} \mid \text{Fn} \mid \text{Pn} \mid \text{Separator}$
 $\text{arity} : \text{Fn} \mid \text{Pn} \mapsto \{0, 1, 2, \dots\}$ — **arity** of names

A recollection of formalized theories (2)

The language of first-order theories

ter : Term – the least language over Alphabet such that:
var : Term for all var : Variable
fn() : Term for all fn with arity.fn = 0
fn(ter-1,...,ter-n) : Term for all fn with arity.fn = n and ter-i : Term for i = 1,...,n

for : Formula – the least language over Alphabet such that:
true, false : Formula
pn(ter-1,...,ter-n) : Formula for all pn with arity.pn = n and ter-i : Term
not(for) : Formula for all for : Formula
and(for-1, for-2) : Formula for all for-1, for-2 : Formula
or(for-1, for-2) : Formula for all for-1, for-2 : Formula
implies(for-1, for-2) : Formula for all for-1, for-2 : Formula
(\forall var)for : Formula for all var : Variable and for : Formula
(\exists var)for : Formula for all var : Variable and for : 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 arithmetics (1)

Language

Variable = $\{x, y, z, \dots, x-1, x-2, \dots\}$, variables may have indices,

F_n = {zer, suc}

P_n = {nat, equ}

with

arity.zer = 0 zer() or just zer represents number zero

arity.suc = 1 suc(x) is the successor of x

arity.nat = 1 nat(x) means that x is a number

arity.equ = 2 equ(x,y) means that x and y are equal

Examples of formulas

true, nat(zer),

equal(suc(zer), suc(x)),

and(equal(suc(zer), suc(x)), equal(suc(suc(y)), suc(suc(x)))),

$(\forall x) (\text{equal}(x, \text{suc}(x)))$.

A recollection of formalized theories (4)

An example of a first-order theory of Peano arithmetics (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 \rightarrow pre-2) for implies(pre-1, pre-2).

Axioms

$x = x$

$x = y \rightarrow y = x$

$(x = y \text{ and } y = z) \rightarrow x = z$

$(x-1 = y-1 \text{ and } \dots \text{ and } x-n = y-n) \rightarrow (fn(x-1, \dots, x-n) = fn(y-1, \dots, y-n))$ for all $fn : F_n$

$(x-1 = y-1 \text{ and } \dots \text{ and } x-n = y-n) \rightarrow (pn(x-1, \dots, x-n) = pn(y-1, \dots, y-n))$ for all $pn : P_n$

nat(zer) zero is a natural number,

nat(x) \rightarrow nat(suc(x)) the successor of a nat. num. is a nat. num.,

nat(x) \rightarrow not (suc(x) = zer) the successor of a nat. num. never equals zero,

$x = \text{suc}(y) \text{ and } x = \text{suc}(z) \rightarrow y = z$ suc is a reversible function

A recollection of formalized theories (5)

Interpretation and semantics (1)

An **interpretation** of a language of a formalized theory:

$\text{Int} = (\text{Uni}, F, P)$

with

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

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

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

A **valuation** is a total function that assigns primitive elements to variables:

$\text{val} : \text{Valuation} = \text{Variable} \mapsto \text{Uni}$

The **semantics** of terms and formulas:

$\text{ST} : \text{Term} \mapsto \text{Valuation} \mapsto \text{Uni}$

$\text{SF} : \text{Formula} \mapsto \text{Valuation} \mapsto \text{Bool}$

A recollection of formalized theories (6)

Interpretation and semantics (2)

The semantics of terms:

$ST : \text{Term} \mapsto \text{Valuation} \mapsto \text{Uni}$

$ST.[\text{var}].\text{val}$	$= \text{val}.\text{var},$	$\text{var} : \text{Variable}$
$ST[\text{fn}(\text{ter-1}, \dots, \text{ter-n})].\text{val}$	$= F[\text{fn}].(ST[\text{ter-1}].\text{val}, \dots, ST[\text{ter-n}].\text{val})$	$\text{arity}.\text{fn} = n$

The semantics of formulas:

$SF : \text{Formula} \mapsto \text{Valuation} \mapsto \text{Bool}$

$SF[\text{true}].\text{val}$	$= \text{tt}$	
$SF[\text{false}].\text{val}$	$= \text{ff}$	
$SF[\text{pn}(\text{ter-1}, \dots, \text{ter-n})].\text{val}$	$= P[\text{pn}].(ST[\text{ter-1}].\text{val}, \dots, ST[\text{ter-n}].\text{val}),$	$\text{arity}.\text{fn} = n$
$SF[(\text{for-1 and for-2})].\text{val}$	$= SF[\text{for-1}].\text{val} \text{ and } SF[\text{for-2}].\text{val}$	
$SF[\text{not}(\text{for})].\text{val}$	$= \text{not} SF[\text{for}]$	
$SF[(\forall \text{var})\text{for}].\text{val}$	$= \text{tt iff for every } \text{ele} : \text{Uni},$	$\text{for}.\text{val}[\text{var}/\text{ele}] = \text{tt}$
$SF[(\exists \text{var})\text{for}].\text{val}$	$= \text{tt iff there exists } \text{ele} : \text{Uni}, \text{ such that }$	$\text{for}.\text{val}[\text{var}/\text{ele}] = \text{tt}$

Note: **and**, **not** are metaoperations.

A recollection of formalized theories (6)

Satisfaction, models and validity

For a given interpretation:

$\text{Int} = (\text{Uni}, F, P)$

A formula for is **satisfied** in Int if:

$\text{SF}[\text{for}].\text{val} = \text{tt}$ for every $\text{val} : \text{Valuation}$

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

$A \vdash \text{for}$

if it is satisfied in every model of this theory.

E.g.: $\text{not}(\text{zer} = \text{suc}(\text{zer}))$ is valid in Peano's arithmetics.

A recollection of formalized theories (7)

Deduction – a way of proving the validity of formulas

$A \models \text{for}$ 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

$$\begin{array}{l} \downarrow \frac{A \models \text{for}(x)}{A \models \text{for}(\text{ter})} \\ x \text{ free in } \text{for}(x) \\ \text{ter} - \text{an arbitrary term} \end{array}$$

Rule of detachment

$$\begin{array}{l} \downarrow \frac{A \models \text{for-1} \\ A \models \text{for-1} \rightarrow \text{for-2}}{A \models \text{for-2}} \end{array}$$

Rule of generalization

$$\begin{array}{l} \uparrow \frac{A \models \text{for}(x)}{A \models (\forall x) \text{for}(x)} \\ \downarrow \\ x \text{ free in } \text{for}(x) \end{array}$$

Gödel's completeness theorem

In every first-order theory with axioms A

$A \vdash \text{for}$ iff $A \models \text{for}$

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

standard model

In first-order Peano arithmetic
 $x \neq \text{suc}(x)$
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: $(X(\text{zer}) \text{ and } (X(x) \rightarrow X(\text{suc}(x))) \rightarrow (\text{nat}(x) \rightarrow X(x))$

X – a predicative variable

Two metatheorem:

1. All models of 2-order Peano's arithmetic are isomorphic to the standard model.
2. $2\text{PA} \models x \neq \text{suc}(x)$

Proof of 2. by induction:

1. $0 \neq \text{suc}(0)$ – is an axiom
2. if $x \neq \text{suc}(x)$ then $\text{suc}(x) \neq \text{suc}(\text{suc}(x))$ – suc is reversible by an axiom
3. $x \neq \text{suc}(x)$ 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)

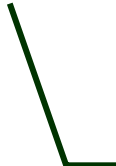
The weaknesses and strengths of second-order theories

Gödel's incompleteness theorem

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

Gödel's adequacy theorem

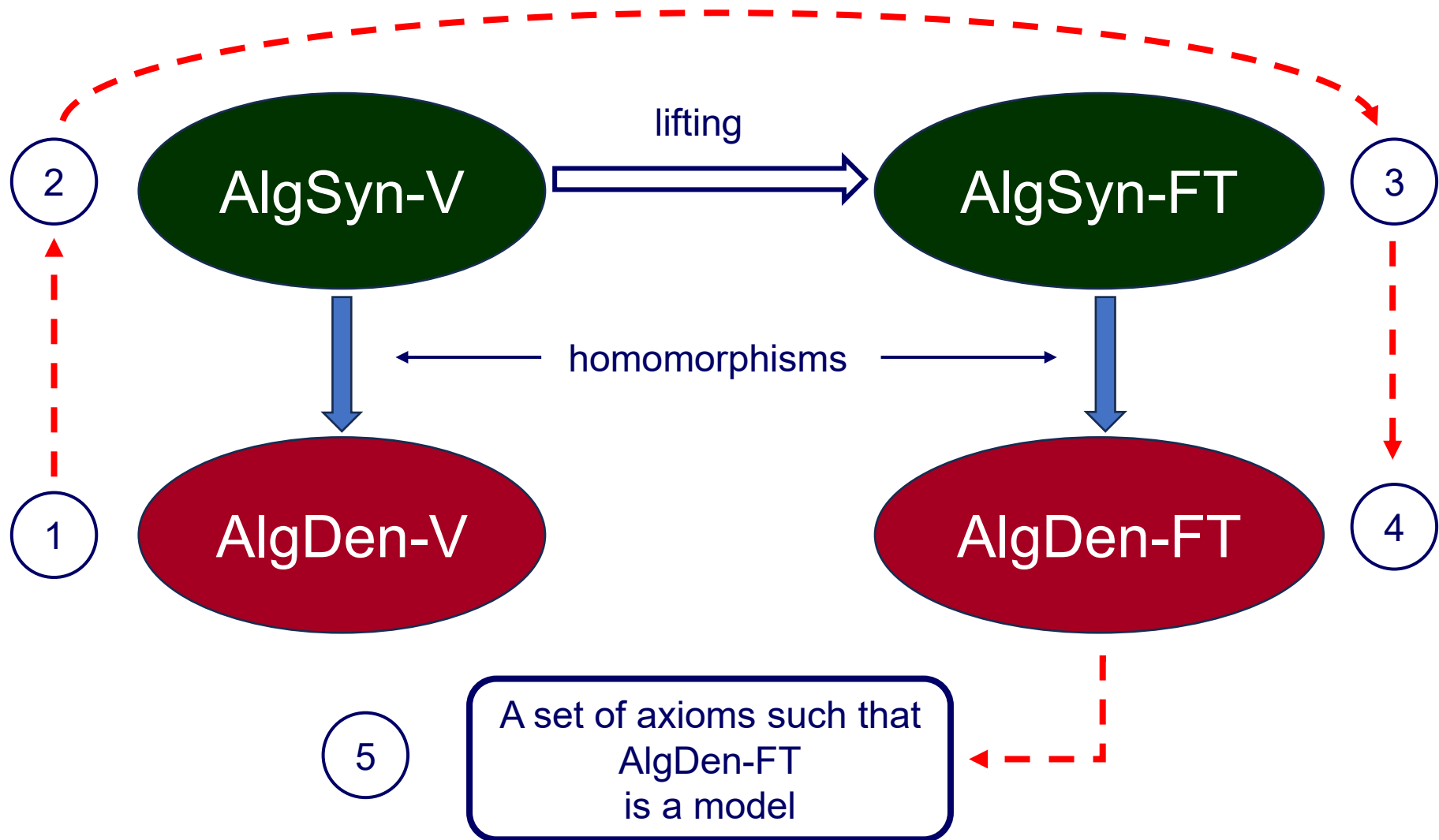
In second-order theories with arithmetic every proved formula is valid i.e. if \vdash for then \models for.



we can trust the theorems
that have been proved

Building Language-FT for Lingua-V (1)

FT – formal theory



Building Language-FT for Lingua-V (2)

From AlgDen-V to AlgSyn-V

Source algebra (Lingua-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}

the elements of Cn-V and Fn-V
i.e. names
and corresponding metavariables
are printed in green

Assumptions on **AlgDen-V**:

Bool = {tt, ff}	is in CarDen-V
and, or, implies, not	are in FunDen-V
boo	: Cn-V
and, or, implies, not	: Fn-V

Building Language-FT for Lingua-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 Lingua-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 \mapsto Cn-V

arity : FunVar \mapsto Cn-V^{c*}

sort : FunVar \mapsto Cn-V

arity : PreVar \mapsto Cn-V^{c*}

Sorted domains of variables:

IndVar.cn = {inv : IndVar | sort.inv = cn}

FunVar.cn = {fuv : FunVar | sort.fuv = cn}

PreVar = {prv : PreVar | sort.prv = boo}

Building Language-FT for Lingua-V (5)

From AlgSyn-V to AlgSyn-FT

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 Lingua-V (6)

From AlgSyn-V to AlgSyn-FT

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

for : Formula = instead of Term.boo

→ 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) |

→ (∀i IndVar) Formula

→ (∃i IndVar) Formula

→ (∀f FunVar) Formula

→ (∃f FunVar) Formula

→ (∀p PreVar) Formula

→ (∃p PreVar) Formula

AlgSyn-FT is implicit in this
 grammar

Building Language-FT for Lingua-V (7)

From AlgSyn-FT to AlgDen-FT

Valuations

uni : Universe = $U\{\text{car.cn} \mid \text{cn} : \text{Cn-V}\}$
vlu : IndValuation $\subseteq \text{IndVar} \mapsto \text{Universe}$
vlu : FunValuation $\subseteq \text{FunVar} \mapsto \{\text{fun} \mid \text{fun} : \text{Universe}^{c^*} \mapsto \text{Universe}\}$
vlu : PreValuation $\subseteq \text{PreVar} \mapsto \{\text{pre} \mid \text{pre} : \text{Universe}^{c^*} \mapsto \text{Bool}\}$
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 \mapsto 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 \mapsto carDen-V.boo

Domains of denotations

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

Building Language-FT for Lingua-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 name of mk-term of sort cn
Fn-V	all former names (including boo sort)
FunVar	all functional variables
PreVar	all predicational variables
{mk-formula}	the name of mk-formula
{and, or, implies, not, $\forall i$, $\exists i$, $\forall f$, $\exists f$, $\forall p$, $\exists p$ }	

Building Language-FT for Lingua-V (9)

Defining the interpretation funDen-FT of functional symbols (1)

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

$\text{funDen-FT.civ-cn-inv} : \mapsto \text{IndVar.cn}$ i.e.

$\text{funDen-FT.civ-cn-inv}().\text{vlu} = \text{civ-cn-inv}().\text{vlu}$

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

$\text{funDen-FT.mk-term-cn} : \text{IndVar.cn} \mapsto \text{carDen-FT.cn}$ i.e.

$\text{funDen-FT.mk-term-cn} : \text{IndVar.cn} \mapsto \text{Valuation} \mapsto \text{carDen-V.cn}$ for all $\text{cn} : \text{Cn-V}$

$\text{funDen-FT.mk-term-cn.inv.vlu} = \text{vlu.inv}$

(3) Functional name **fn** : Fn-V with $\text{arity.fn} = (\text{cn-1}, \dots, \text{cn-n})$

$\text{sort.fn} = \text{cn}$:

$\text{funDen-FT.fn} : \text{carDen-FT.cn-1} \times \dots \times \text{carDen-FT.cn-n} \mapsto \text{carDen-FT.cn}$

$\text{funDen-FT.fn}(\text{den-1}, \dots, \text{den-n}).\text{vlu} = (\text{funDen-V.fn})(\text{den-1.vlu}, \dots, \text{den-n.vlu})$

(4) Functional variable **fuv** : FunVar.cn with $\text{arity.fuv} = (\text{cn-1}, \dots, \text{cn-n})$

$\text{sort.fuv} = \text{cn}$:

$\text{funDen-FT.fuv} : \text{carDen-FT.cn-1} \times \dots \times \text{carDen-FT.cn-n} \mapsto \text{carDen-FT.cn}$

$\text{funDen-FT.fuv}(\text{den-1}, \dots, \text{den-n}).\text{vlu} = (\text{vlu.fuv})(\text{den-1.vlu}, \dots, \text{den-n.vlu})$

Building Language-FT for Lingua-V (9)

Defining the interpretation funDen-FT of functional symbols (2)

(5) Formula-making function **mk-formula**

funDen-FT.**mk-formula** : IndVar.**boo** \mapsto carDen-FT.**boo** i.e.

funDen-FT.**mk-formula** : IndVar.**boo** \mapsto Valuation \mapsto 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** \mapsto 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** \mapsto CarDen-FT.**formula**

funDen-FT.**∀i**.(inv, den).vlu =

for any uni : Universe, den.(vlu[inv/uni]) = tt \rightarrow tt

true \rightarrow ff

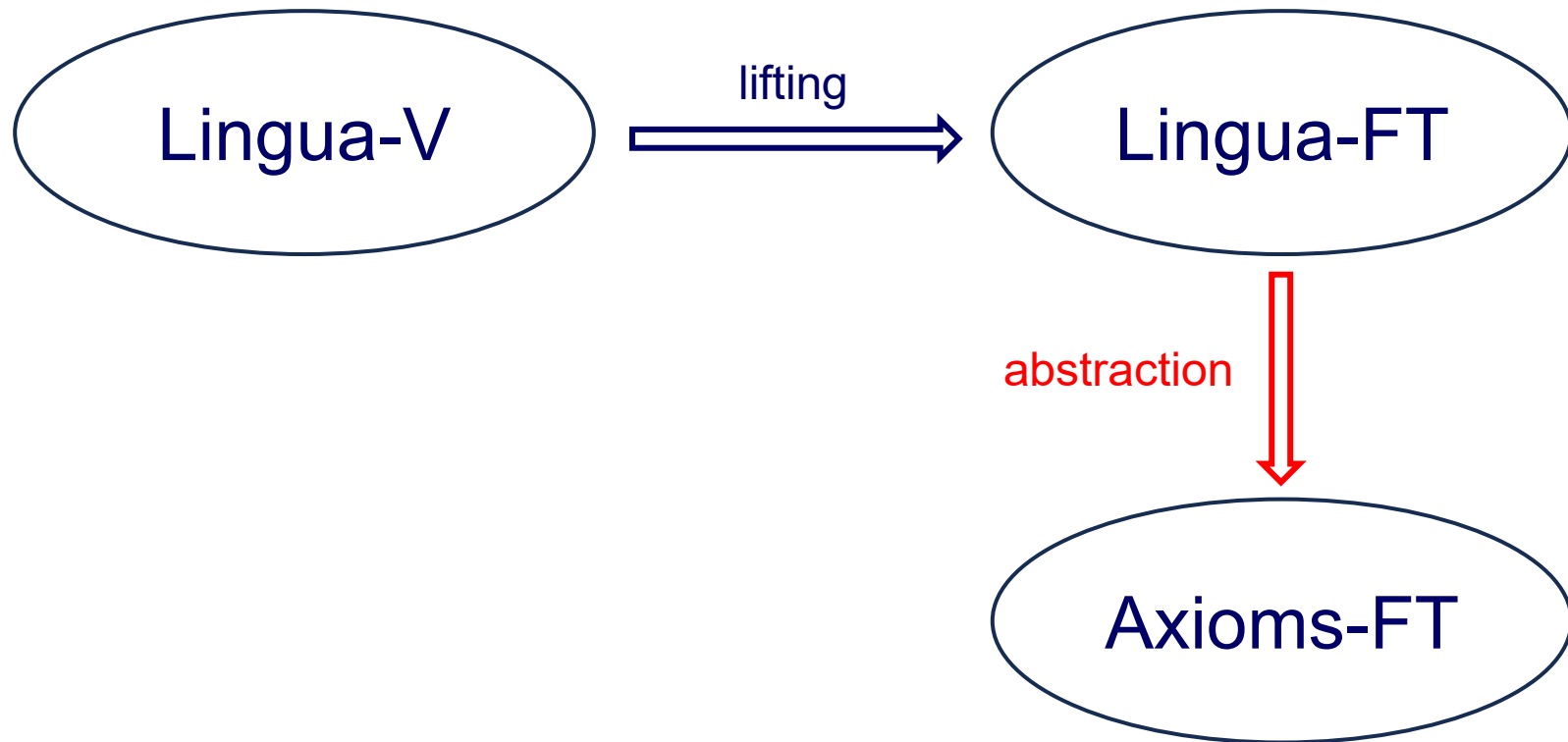
Building Language-FT for Lingua-V (10)

How theories and models are built

mathematical logic: **axioms** → its models

working mathematician: a model → its **axioms**

our case





Thank you for
your attention