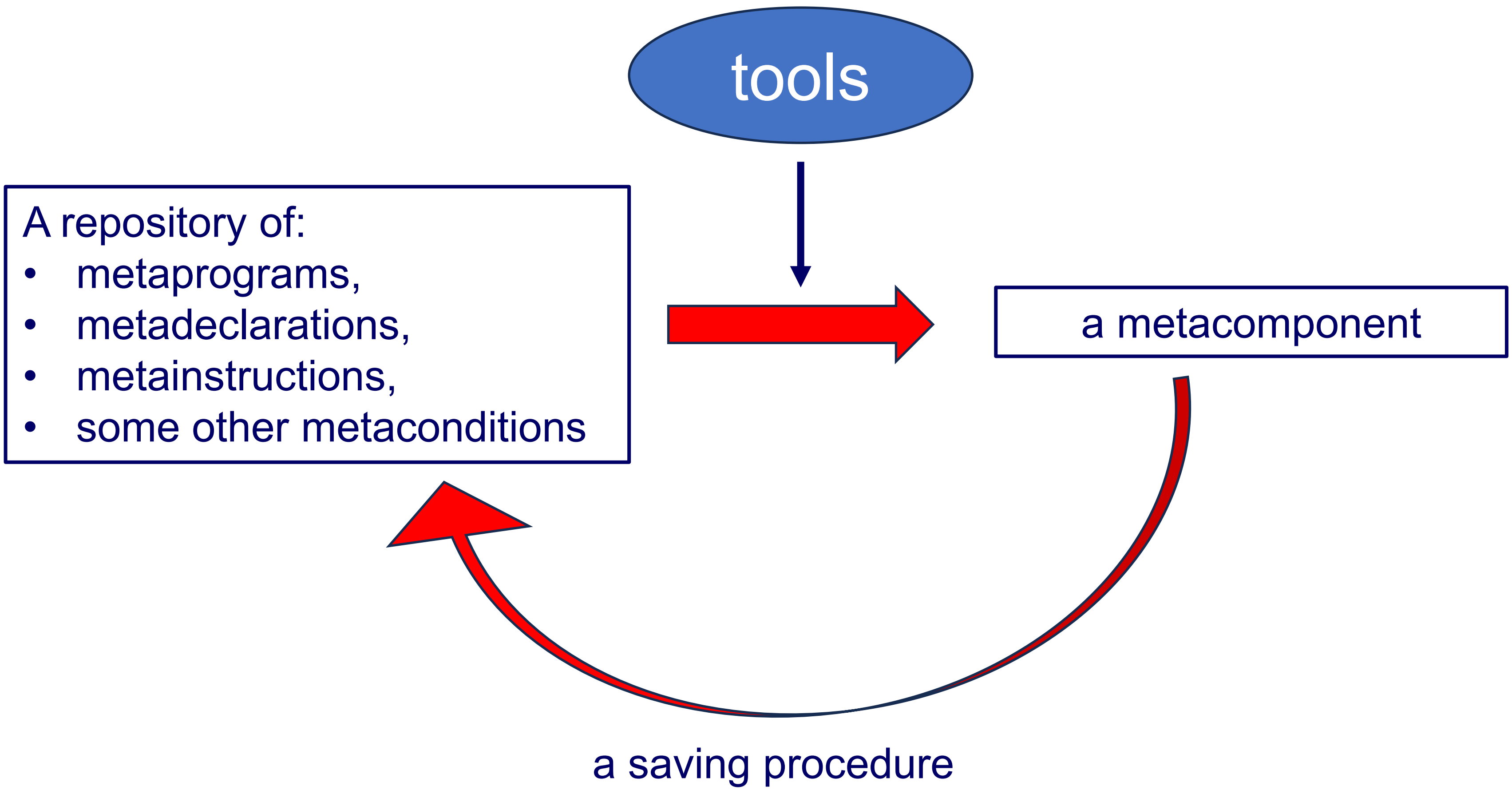
Preliminary thoughts on ecosystems for Lingua programmers

Andrzej Jacek Blikle

July 12th, 2025

Programs' development cycle in Lingua-V

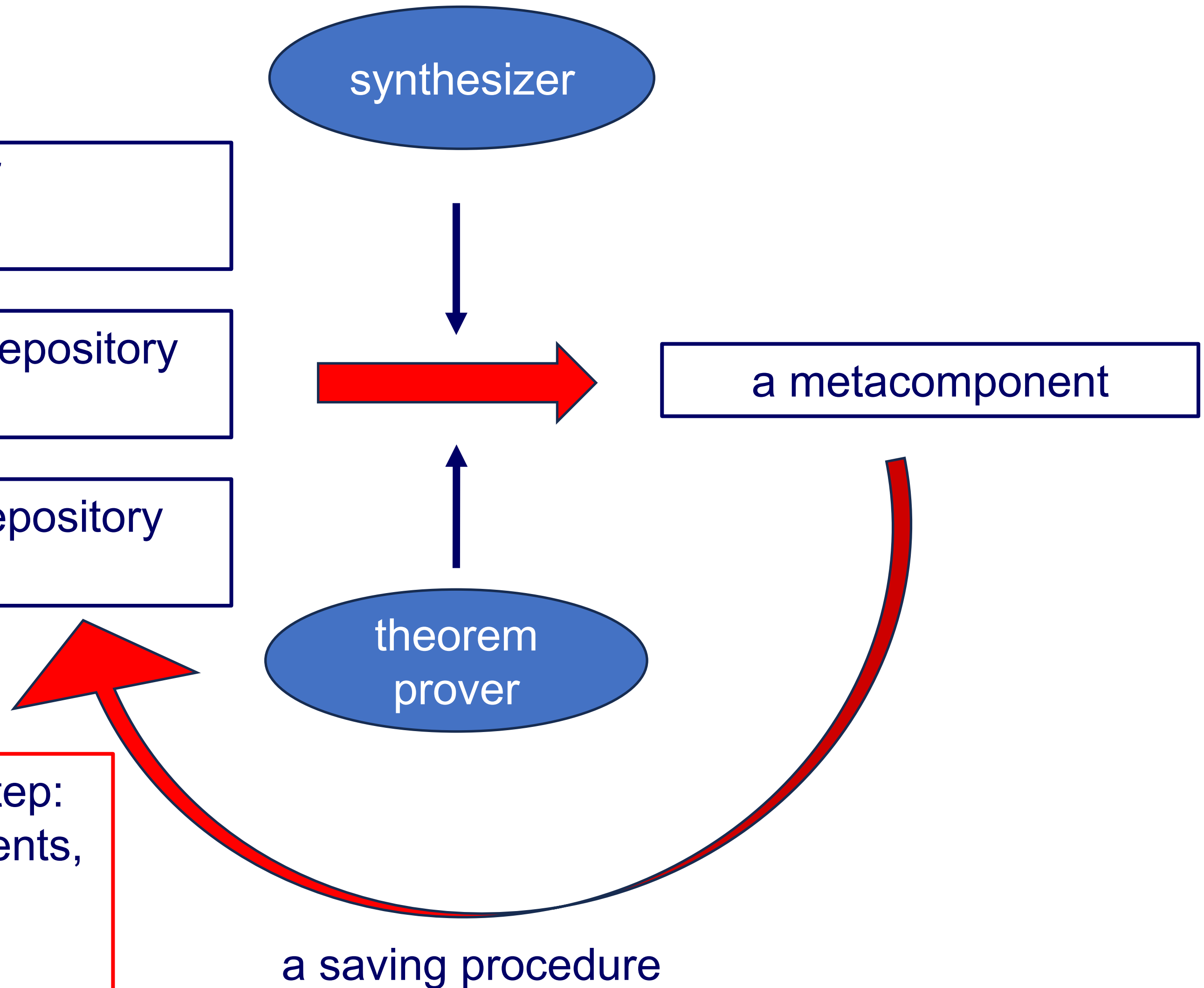


A closer look to programs' development cycle

RCR: a repository of construction rules

PRM: a permanent repository metacomponents

TRM: a temporary repository metacomponents



One programming step:
1. indicate components,
2. indicate rules,
3. prove,
4. synthesize
5. save

Examples of theorems to be proved

x **is** integer ⇨ x < x + 1

(x+1 ≤ isrt(n))

≡

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

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

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 ≡ 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

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)),

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 ⇨ 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

pre sin **@** con
sin
post con

substitution

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)

theorem prover

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**)**

proof and substitution

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

pre sin **@** con
  sin
post con

substitution

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**)**

theorem prover

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**)**

proof and substitution

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**)**

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) ⇨ **(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 ⇨ 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 **M-language**:
   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 **M-language**:
   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:

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

A language of first-order theories includes two syntactic categories

terms — represent functions
formulas — represent predicates

Primitives of syntax

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 $\mapsto$ {0, 1, 2,…} — 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
(∀var)for	: Formula	for all var : Variable and for : Formula
(∃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,…, x-1, x-2,…}, variables may have indices,
Fn = {zer, suc)
Pn = {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))),
(∀x) (equal(x, 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 → 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

nat(zer) zero is a natural number,
nat(x) → nat(suc(x)) the successor of a nat. num. is a nat. num.,
nat(x) → not (suc(x) = zer) the successor of a nat. num. 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; $F[fn] : Uni^{cn} \longmapsto Uni$ for arity.fn = n
$F[fn] : \longmapsto Uni$ for arity.fn = 0

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

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

val : Valuation = Variable $\longmapsto$ Uni

The semantics of terms and formulas:

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 terms:

ST : Term ⟼ Valuation ⟼ Uni

ST.[var].val = val.var, var : Variable
ST[fn(ter-1,…,ter-n)].val = F[fn].(ST[ter-1].val,…,ST[ter-n].val) arity.fn = n

The semantics of formulas:

SF : Formula ⟼ Valuation ⟼ Bool

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

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

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

For a given interpretation:

Int = (Uni, F, P)

A formula for is satisfied in Int if:

SF[for].val = tt for every val : 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.: not(zer = suc(zer)) is valid in Peano's arithmetics.

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

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

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: (X(zer) **and** (X(x) → X(suc(x)) → (nat(x) → X(x))

X – a predicative variable

Two metatheorem:

1. All models of 2-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


Thank you for
your attention