Investigations on the logical aspects of ecosystems for programmers in Lingua-V

(a working version)

Andrzej Jacek Blikle

October 4th, 2025

Contents

1 Introduction

This paper is addressed to readers familiar with the techniques of validating programming originated by Andrzej Blikle at the turn of the 1970s and 1980s (see [2], [3], [4], and [5]) and currently explored on a theoretical basis in the **Lingua** project [6]. From the perspective of the pursued goal, this approach is similar to the idea of developing programs that are *correct-by-construction* suggested by Edsger Dijkstra in the years 1969/70 (see [7] and [8]), and currently elaborated in the **Dafny** project (see [11], [10], [9] and [10]). Both these approaches share the idea that a program should be developed in a step-by-step process where each step guarantees the correctness of the current program. In both methods, a program (syntactically) includes its specification in the form of pre- and postconditions, along with some internal assertions. However, technically and mathematically, these approaches are significantly different.

The technical core of the **Lingua** project is a virtual (so far) programming language **Lingua-V** (V for "validating"), which includes a "standard" (also virtual) programming language **Lingua**, plus its extension by *metaprograms*. The latter syntactically consist of **Lingua** programs plus their specifications. A metaprogram is said to be *correct* if its program component is totally correct with clean termination (generates no errors) for its specification.

The category of *correct metaprograms* constitutes a special case of a category of *valid metaconditions*. The latter are developed from other correct metaconditions through *construction rules* that include proof rules in Dijkstra's style (but with three-valued predicates) plus some additional rules. Based on a denotational model of **Lingua-V**, these rules are proven sound, meaning that given valid metaconditions as inputs, they return valid metaconditions as outputs. It is essential to emphasize in this place that the development of correct metaprograms can't be described as a process where we build exclusively metaprograms from other metaprograms. As has been shown in Sec. 9 of [6], arbitrary metaconditions must be included in this game.

The primary mathematical difference between the **Lingua** and **Dafny** projects is that Lingua has a denotational model (semantics), and on this basis, we prove the soundness of our construction rules. In **the Dafny project, only the syntax is formally defined,** and program-construction rules are essentially assumed to be sound. On the other hand, it is worth noting that **Dafny** offers an implemented language. More about the differences between these two approaches can be found in Sec. 7

2 Preliminaries

2.1 Introductory remarks about an ecosystem for Lingua-V programmers

An ecosystem for **Lingua-V** programmers should help them create correct metaprograms and, in fact, valid metaconditions. We assume that the latter will be developed in a bottom-up manner, starting with some previously created valid metaconditions, i.e.:

- correct metaprograms,
- correct metadeclarations,
- correct metainstructions,
- other valid metaconditions,

and combining or transforming them into new valid metaconditions using sound construction rules. An ecosystem including the following components will support the work of programmers in **Lingua-V**:

- a *repository* dedicated to storing:
  - valid metaconditions and, in particular, the axioms of a formalized theory of denotations, a **D-theory**, based on a formalized language **Lingua-D**,
  - sound program-construction rules,
  - conditions,
- an *engine,* to construct new valid metaconditions, including:
  - an intelligent *text editor* with a **Lingua-D** parser to ensure the syntactical correctness of created metaconditions,
  - a *composer* providing procedures, called *actions*, for creating new valid metaconditions from the ones stored in the repository,
  - a *theorem prover* to prove the validity of these metaconditions that can't be generated by the composer.

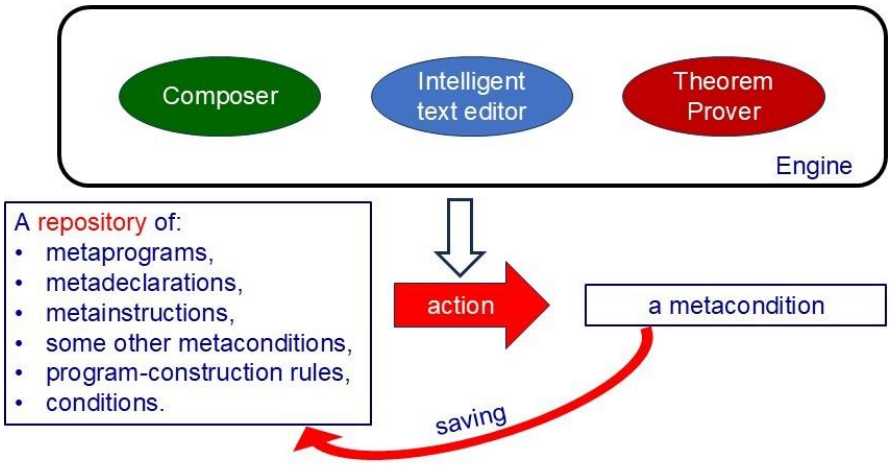


Fig. 2.1-1 An ecosystem for programmers

Storing conditions in the repository serves as a technical vehicle, enabling programmers to use concise names for complex conditions during program development (see Sec. 6.5 for examples).

We assume that the ecosystem will be dynamically developed by two categories of users:

- *programmers* — using the current repository to derive new metacomponents out of previously stored,
- *superusers* — authorized to add new axioms and new construction rules to the repository and to update the text editor whenever **Lingua-D** is enriched.

Developing correct metaprograms within an ecosystem can be seen as proving the validity of formulas in **D-theory**. This theory will be formalized as a triple consisting of:

- a formalized language **Lingua-D** derived from (including) the source language **Lingua-V**,
- a set of axioms, i.e., formulas written **Lingua-D**,
- a set of classical inference rules (substitution, detachment, etc.) along with some rules specific to the underlying language **Lingua-D**.

In this paper, we investigate the general problem of developing a formalized theory for a programming language with a denotational model. We shall illustrate our proposed solution by developing **Lingua-D** and **D-theory** for **Lingua-V**. Still, we will also try to formulate some general recommendations applicable to a class of languages with denotational models.

To discuss the main subjects of our investigations — i.e., **Lingua-D** and **D-theory** — we will need a metalanguage and a corresponding metatheory. We take **MetaSoft** as the former, and our metatheory will encompass all mathematical theories used in [6], i.e., set theory, the theory of relations, the theory of formal languages, and the theory of CPOs, among others. We will refer to it as **M-theory**, and (so far?), we shall not formalize it.

The last language we will introduce in this paper is Lingua-E, which will represent the ecosystem itself. It will include procedures, referred to as *actions*.

Summing up, on our way from **Lingua** to an ecosystem, we shall investigate and develop a hierarchy of languages **—** all of them with denotational models:

- **Lingua** **—** a source programming language,
- **Lingua-V** **—** a language of validating programming; an extension of **Lingua**,
- **Lingua-D** **—** a language of a formalized theory of the denotations of **Lingua-V**,
- **Lingua-E** **—** a language of actions executed by the ecosystem.

At the end of this section, one methodological remark is in order about the development of correct metaprograms in our approach. The development of correct metaprograms, and in general valid metaconditions, may be seen as proving theorems in **D-theory**. This task can be conducted in two regimes:

- *ex-ante constructive proofs* **—** when we use a composer to build valid metaconditions from earlier proved ones; in this case, the proof of validity is developed simultaneously with the construction of the metacondition,
- *ex-post analytic proofs* **—** when we use a theorem prover; in this case, a metacondition to be proved is presented first, and its proof is developed (discovered) later.

We expect that most of our programmers' work will involve using a composer to develop ex-ante proofs, although they may also occasionally use a theorem prover to find an ex-post proof. It is important to note here that the theorem prover will not be used to verify the correctness of programs, but only to check the validity of certain “intermediate” metaconditions.

2.2 An example of a program development

Let’s analyze an example of a program development to illustrate (in advance) the method that we are going to study. Assume that we intend to develop the following simple metaprogram:

```
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)
```

We tacitly assume that in the current implementation of **Lingua-V,** the range of integers is such that our program will not generate an overflow error, and, therefore, for simplicity, we shall omit this aspect in our metaprogram.

We shall describe the development of our metaprogram as a sequence of compound steps, each consisting of several elementary steps. We assume that at the end of every compound step, the resulting metacomponent is saved in the repository. In the description of each step, we first specify the intended target metaprogram, and then we explain the process of its construction. As in [6], metaprograms and their components are typed in Ariel narrow; however, we make an exception for metavariables, which are typed in Arial underlined. This rule is formalized and explained in Sec. 4. Actions in **Lingua-E** are typed in Arial Narrow blue. We shall use the same font to type the names of the elements stored in the Repository. All elements of the Repository will be referred to as *lemmas*.

Step 1. The development of the declaration of x:

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

This metacomponent is generated from the following atomic construction rule (Sec. 9.4.4 of [6]), which must be stored in the repository as an axiom:

A1 : **pre** (ide **is free**) **and-k** (tex **is type**)
let ide **be** tex **tel**
post var ide **is** tex

We use composer to execute the action

substitute(**A1**, [ide/x, tex/integer], **P1**)

which applies the indicated substitution to **A1** and stores it in **P1**.

Step 2. The elimination of the tautology condition (integer **is type**) from **P1**.

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

This step is based on the following metaconditions and construction rules that must be present in the Repository (error transparency of con means that, for states carrying an error, the denotation of con returns this error).

A2 **error-transparent**(con) **implies** con ⇨ **NT** — this is a definitional axiom of **NT**; denotationally, **NT** is a condition, to be read "nearly true", that is satisfied for all states that do not carry errors, whereas for states carrying errors they return these errors,

A3 (**error-transparent**(con1) **and** (**NT** ⇨ con2)) **implies** (con1 **and-k** con2 ⟺ con1),

A4 (con1 ≡ con2) **implies** (con1 ⇨ con2),

A5 (con1 ≡ **NT**) **implies** ((con1 **and** con2) ≡ con1)),

A6 integer **is type** ≡ **NT**,

A7 **error transparent**(ide **is free**),

A8
pre prc **:** sin **post** poc
prc ⟺ prc-1
---
pre prc-1 **:** sin **post** poc

Using lemmas **A2** to **A7**, we derive metacondition (L for "lemma")

L1 (x **is free**) **and-k** (integer **is type**) ⟺ (x **is free**)

and then we derive of **P2** from **P1** by **A8**.

Step 3. The development of the metadeclaration

P3 : **pre** (y **is free**)
let y **be** integer **tel**
post (**var** y **is** integer)

This step is analogous to the development of **P2**.

Step 4. The enrichment of **P2** by **(y is free)** and **P3** by **(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)**

P5 : **pre (var** x **is** integer**) and-k (**y **is free)**
let y **be** integer **tel**
post (var x **is** integer**) and-k (var** y **is** integer**)**

To perform these transformations, we need the following lemmas in the Repository (for **irrelevant for,** see Sec. 9.3.4 of [6]):

L2 **different(**ide1, ide2**) implies ((**ide1 **is free) irrelevant for (let** ide2 **be** tex **tel))**,

L3 **different(**ide1, ide2**) implies ((**ide1 **is** tex1**) irrelevant for (let** ide2 **be** tex2 **tel)**,

L4
pre prc **:** sin **post** poc
con **irrelevant for** sin
---
pre prc **and-k** con **:** sin **post** poc **and-k** con

Step 5. The sequential composition of **P4** and **P5**

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 (var** y **is** integer**)**

Here we use Lemma 9.4.3-5 (Sec. 9.4.3 of [6]), which must be in the Repository.

Step 6. The development of a metaprogram

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**)**

To develop this program, we use @-tautology (Sec. 9.4.6.2 of [6])

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

which must be in the Repository. By an appropriate substitution applied to this formula, we create the following (concrete) metaprogram:

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**)**

Next, we find in the Repository the following lemma:

L5: (ide not in vex) **implies**
(ide := vex @ (var ide is tex) and-k (vex is tex) and-k (ide = vex) ⟺ (var ide is tex) and-k (vex is tex))

from which we can derive by appropriate rules

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**)**

Now, we use Lemma 7. (see **P2**

Step 7. The development of a metaprogram

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

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

We use a technique similar to that in Step 6 to derive the weak equivalence.

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

Step 8. The development of a metaprogram

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**)**

We combine sequentially **P6**, **P7,** and **P8**.

Step 9. The development of a metaprogram

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**)**

We proceed similarly to Step 6.

Step 10. The development of a metaprogram

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**)**

In this step, we replace the postcondition of P10 by a weaker one (Sec. 9.4.3 of [6]), and we have to use the theorem prover to prove the following metaimplication:

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

which essentially means that we have to prove the validity of the formula 8 < 10.

Step 11. Our target program is generated by combining programs **P9** and **P11**.

There are two critical observations we can draw from our example. Both are based on the fact that our program-derivation technique involves proving theorems about programs and that our proofs differ from proofs in "usual" mathematics.

First, as we have already mentioned, our proofs are *ex ante rather than ex post*. This fact has a technical consequence regarding the process of conducting proofs. In traditional proofs conducted by theorem provers, users must provide the so-called *tactics*, which are hints on how to carry out proofs. These tactics need to be "known" by users, which is not always straightforward. In our case, the roles of ex-post tactics are played by the ex-ante choices of construction rules and metaconditions in the Repository. These choices are quite natural for programmers who know what program they intend to create.

Our second observation is that, in our case, most of the work in constructing a program is done by the composer rather than the theorem prover. In our example, the only general mathematical hypothesis we needed to prove appeared in Step 10 as the validity of

8 < 10 **≡ NT**

which must be deduced from the basic mathematical axioms of integer arithmetic. At the same time, we have referred to our Repository in using composer about 40 times.

Of course, based on a single toy example, we cannot draw general conclusions; however, this case suggests that there is something underlying this discrepancy. In our view, these preliminary investigations of the process of program development indicate that using a theorem prover in an ex-ante constructive process of program validation may be much less "intensive" than in the ex-post analytical case.

2.3 A recollection on first-order and second-order formalized theories

As we have seen in our example, some steps in the development of a program require proving the satisfaction of formulas that describe facts about the values of variables appearing in the program. Since in a "practical programming", such formulas may be computationally fairly complicated — the number of variables in these formulas may be comparable to the number of variables in a current program — an automatic theorem prover should support programmers in Lingua-V. To build such a prover, or to adapt an existing one to our goal, we first need to establish a formalized theory rich enough to talk about programs and their components, i.e., data, types, values, references, denotations, and so on. In this paper, we outline a general scenario for developing a theory for a programming language with a denotational model. We begin with a brief review of the concepts of first-order and second-order formalized theories.

In *first-order theories,* we talk about the elements of a set Uni, usually called the *universe,* and about many-argument *functions* and *predicates* on this set, i.e.:

fun	: $\text{Uni}^{cn} \mapsto \text{Uni}$	for n ≥ 0	functions
pre	: $\text{Uni}^{cn} \mapsto \text{Bool}$	for n ≥ 0	predicates

where

boo : Bool = {tt, ff}

The language of a first-order theory includes three syntactic categories:

- *variables* running over Uni,
- *terms* that represent functions,
- *formulas* that represent predicates.

To define them, we are given four mutually disjoint sets of symbols:

var	: Variable	— a possibly infinite set of variables
fn	: Fn	— a finite set of function names,
pn	: Pn	— a finite set of predicate names,
sep	: Separator	— a finite set of separators such as parentheses, colons, etc.

The union of all these sets is called an *alphabet*:

Alphabet = Variable | Fn | Pn | Separator

Every functional and predicative symbol has an *arity* — a non-negative integer indicating the number of arguments of this functional or predicative symbol, respectively. We thus define a function:

arity : Fn | Pn ⟼ {0, 1, 2,…}

We assume that zero-ary functional symbols, called *constants*, represent the elements of Uni, and zero-ary predicative symbols represent the logical values *true* and *false*. Based on these assumptions, we define the sets of variables, terms, and formulas using the following grammar.

```
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 and the argument tuples with n elements

for : Formula =
   true                           |
   false                          |
   pn(Term, … ,Term)              |  for all pn : Pn with arity.pn = n and the argument tuple with n elements
   not(Formula)
   and(Formula, Formula)          |
```

or(Formula, Formula) |
implies(Formula, Formula) |[1]
(∀ Variable) Formula |
(∃ Variable) Formula

In this grammar, we have introduced a (meta)notational convention such that:

1. the names of functions and predicates are printed in green Arial Narrow,
2. separators are printed in green Arial Narrow,
3. variables like x, y, z… are printed in black Arial,
4. metavariables like "Variable", "Term", "Formula" are printed in black Arial.

This convention slightly modifies the one introduced in Sec. 7.2 of [6], where all terminal symbols of grammars are written in Arial Narrow. In this paper, we make an exception for variables, which are printed in Arial. This decision will be explained in Sec. 4.4.

Variables appearing in formulas under the signs of quantifiers are said to be *bound*, and variables that are not bound are called *free*. In the set of formulas, we distinguish four categories:

- *open formulas* — at least one free variable, e.g., x < 1 or (∀x)(x < y),
- *closed formulas* — all variables are bound, e.g., (∀x)(∃y)(x < y),
- *ground formulas* — no variables in such formulas, e.g., 1 < 2,
- *free formulas* — some unbound variables in such formulas, e.g., x < 2 or (∀x)(x < y).

In the set of terms, we distinguish only two categories:

- *ground terms* — no variables,
- *free terms* — with variables.

Let's consider now a *first-order arithmetic of natural numbers* (non-negative integers) as an example of a first-order theory. Let

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

where

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 terms in this theory may be:

zer, suc(zer), suc(suc(zer)),.., suc(x), suc(suc(y)), …

and 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))).

Given the language of our theory, we can define axioms that express the intended meanings of functions and predicates represented in the language by functional and predicational symbols. For better readability, we shall write:

(ter-1 = ter-2) instead of equ(ter-1, ter-2),
(for-1 and for-2) instead of and(for-1, for-2),

[1] Of course, while we have negation and alternative, the remaining connectives may be defined, but we introduce them as primitive connectives for convenience.

(pre-1 → pre-2) instead of implies(pre-1, pre-2).

We will also omit parentheses when this does not cause ambiguity. The following axioms specify the expected properties of the equality predicate:

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

Axioms (1) – (3) describe the fact that equality is an equivalence relation and two remaining (schemes of) axioms — that it is a congruence for all functions and predicates. The next group of axioms describes the intended properties of the meanings of num, zer, and suc[2]:

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

A formal language together with axioms constitutes an *axiomatic theory*. On the grounds of such a theory, we can define the concepts of the *validity* of formulas and of a *model* of the theory. We shall define these concepts in an abstract case of an arbitrary first-order theory. We start by defining an *interpretation* of the underlying language as a triple:

Int = (Uni, F, P)

where

- Uni is a set called *universe*, and its elements are called *primitive elements* of the interpretation,
- F is a function that, with every functional symbol fn of arity n ≥ 0, assigns a n-ary function F[fn] : Uni^{cn} ⟼ Uni, and for n = 0, F[fn] : ⟼ Uni,
- P is a function that, with every predicative symbol pn of arity n ≥ 1, assigns a n-ary predicate P[pn] : Uni^{cn} ⟼ Bool; we assume that P[true] = tt and P[false] = ff

Note that F and P are functions that belong to the metalevel of our theory, rather than to the theory itself. In contrast, fn and pn belong to the theory level — more precisely, to the theory's language. By a *valuation,* we mean a total function that assigns elements of Uni to variables:

vlu : Valuation = Variable ⟼ Uni[3]

Now, for every interpretation of our theory, we can define the *semantics* of variables SV, of terms ST, and of formulas SF, respectively:

SV : Variable ⟼ Variable
ST : Term ⟼ Valuation ⟼ Uni
SF : Formula ⟼ Valuation ⟼ Bool

such that for any variable var

SV[var] = var the denotation of a variable is the variable itself,

and for every vlu : Valuation

ST.[mk-term(var)].vlu = vlu.var for every var : Variable
ST.[fn(ter-1,…,ter-n)].vlu = F[fn].(ST.[ter-1].vlu,…,ST.[ter-n].vlu) where n = arity.fn, n ≥ 0

SF.[true].vlu = tt
SF.[false].vlu = ff

[2] These axioms were formulated by an Italian mathematician Giuseppe Peano (1858 – 1932).

[3] We introduce a metavariable vlu rather than val, since the latter is used in [6] for values.

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

where **and**, **not**,… are classical logical connectives of our metalevel.

It is to be noticed in this place that scripts like pn(ter-1,…,ter-n) where ter-i's represent arbitrary terms formally do not belong to the language of our theory, but only represent its elements at the level of a metalanguage.

We say that a formula for is satisfied in a given interpretation if, for every valuation vlu of this interpretation, the formula evaluates to true, i.e.,:

SF.[for].vlu = tt.

An interpretation is said to be a *model* of an axiomatic theory if all axioms of that theory are satisfied in this interpretation. A formula for is said to be *valid* in a theory with a set of axioms A, which we describe by a metaformula:

A |= for,

if it is satisfied in every model of this theory. In this case, we also say that for is a *semantic consequence* of the set of axioms A. An example of a valid formula in Peano's arithmetic is

not(zer = suc(zer))

which says that zero is different from its successor. Note that

A |= for(x) **iff** A |= (∀x)for(x)

where for(x) symbolically denotes a formula with a free variable x.

Although the concept of validity provides a clear distinction between valid and invalid formulas, we do not use this concept to justify mathematical hypotheses. Instead, we employ a method of *deduction* that allows us to derive formulas from axioms using *inference rules*. If a formula for can be derived by deduction from a set of axioms A, then we call it a *theorem,* and we denote this fact by a metaformula:

A |- for

The three most commonly used rules of inference (we leave out some rules for quantifiers) are the following (not entirely formal):

Rule of substitution

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

This rule states that if in a theorem we replace free variables with arbitrary terms, then the new formula can also be considered a theorem. The second rule is the primary foundation of deduction:

Rule of detachment (modus ponens)

$$\begin{array}{l} A \vdash for1 \\ A \vdash (for1 \textbf{ implies } for2) \end{array}$$

A |- for2

If we prove for-1 and we prove the implication (for-1 → for-2), then we can conclude that for-2 has been proved.

Rule of generalization

$$\frac{A \vdash for(x)}{A \vdash (\forall x)\ for(x)}$$

where x is free in for(x). We will not discuss other rules involving quantifiers at this point, as they are not necessary. Once we add rules of inference to an axiomatic theory, we get a formalized theory.

An Austrian mathematician, Kurt Gödel, proved in his doctoral dissertation in 1929 the following theorem:

> ***Gödel's completeness theorem***. *In first-order theories, every proved formula is valid, and every valid formula can be proved, i.e.,* A |= for *iff* A |- for.

Unfortunately, despite this "highly desirable" property, first-order theories also have a serious flaw. Every first-order theory that has an infinite model also has infinitely many non-isomorphic models. In simpler terms, we could say that in first-order theories, we never fully know what we are talking about. This is easily seen in our example of first-order arithmetic. Although our goal was (supposedly) to create a theory of natural numbers, and although such numbers with suc(x) = x+1 do form a model of our theory, the theory has many other non-isomorphic models. Let's look at two of them:

- In the first model, Uni is the set of all real numbers, num(x) is satisfied for all elements of Uni, and suc(x) = x+1. In this model, there are elements of Uni that are not reachable from zero by a multiple use of successor[4].
- In the second model, Uni includes only natural numbers plus one decimal number, e.g., 0,5. We set num(x) = tt for all elements of Uni, suc(x) = x+1 for all natural numbers, and suc(0,5) = 0,5. In this model, an element may be equal to its own successor.

To address the "ambiguity of axioms" — formally, we say that our theory is *noncategorical*, which means that it has nonisomorphic models — we need to enrich the theory with the following second-order *axiom of induction*, where valuations may assign to variables not only elements of Uni but also predicates in Uni.

10. (P(zer) and (P(x) → P(suc(x)) → (num(x) → P(x))

In this axiom, P is a second-order (predicative) variable of arity 1, which means that valuations may assign to it arbitrary unary predicates in Uni. Axiom 10. says that if (our zero) zer has property P and, if for every element x with property P the successor of this element has property P, then all elements that satisfy num have property P. In other words, num represents the least set that includes zer and all its successors. This axiom guarantees that all models of our new theory are isomorphic with the algebra of all natural numbers, where suc(x) = x + 1.[5]

Another "advantage" of axiom 10 is that in our theory, we can carry out proofs by induction. As a matter of fact, we can do it in every theory which *includes second-order arithmetic*, i.e., which either includes axioms (6) – (10), or where these axioms can be formulated and proved. As an example, let's prove the following theorem

x ≠ suc(x) (2.1-1)

where x ≠ y stands for not(x = y). It is worth noticing that this formula is not valid in the first-order arithmetic.

Let Q(x) be a predicate satisfied if x ≠ suc(x). By axiom (8), Q(zer) is satisfied. Let for a given x, formula x ≠ suc(x) be satisfied. Then, by axiom (9) suc(x) ≠ suc(suc(x)), hence Q(suc(x)). The application of axiom (10) completes the proof.

---

[4] Note that here suc denotes a function that is the meaning of function name suc (green).

[5] Axiom (10) was also formulated by G. Peano.

The main technical difference between first-order and second-order theories is that the latter have three sets of variables, rather than only one, i.e.:

inv	: Iv = {inv-1, … ,inv-p}	individual variables; first-order variables
fuv	: Fv = {fuv-1, … ,fuv-q}	functional variables; second-order variables
prv	: Pv = {prv-1, … ,prv-r}	predicative variables; second-order variables,

and, of course, function arity is now extended to second-order variables, i.e.:

arity : Fn | Pn | Fv | Pv ⟼ {0, 1, 2, …}

We appropriately extend the sets of terms and formulas by adding new clauses to the equations of the former grammar:

ter : Term =
all former clauses
fuv(Term, … ,Term) | for every fuv : Fv with arity.fuv = n and the argument tuples with n elements

for : Formula =
all former clauses
prv(Term, … ,Term) | for every prv : Pv with arity.prv = n and the argument tuples with n elements

Note that for every second-order variable we create an individual grammatical clause, similarly to functional and predicational symbols. At the same time, we do not introduce grammatical equations to generate second-order variables, as is the case for individual variables:

inv : IndVar =
x | y | z | x-1 | y-1 | z-1 |… individual variables

Finally, we add new semantic clauses to the definitions of the functions of semantics of terms and of formulas — again, one clause for every second-order variable:

ST.[fuv(ter-1,…,ter-n)].vlu = (vlu.fuv).(ST.[ter-1].vlu,…,ST.[ter-n].vlu) where n = arity.fuv, , n ≥ 0

SF.[prv(ter-1,…,ter-n)].vlu = (vlu.prv).(ST.[ter-1].vlu,…,ST.[ter-n].vlu) where n = arity.prv, , n ≥ 1

In the future, the theories used by our theorem provers will be second-order and will include arithmetic, thereby offering the possibility of carrying out proofs by induction. There is, however, a price that we have to pay for all these advantages:

Gödel's incompleteness theorem. *In second-order theories with arithmetic, there exist valid formulas that can't be proved, i.e.,* A |= for but not A |- for.

Fortunately, we have yet another theorem:

Gödel's adequacy theorem. *In second-order theories with arithmetic, every proved formula is valid. I.e. if* A |- for *then* A |= for.

This second theorem is satisfied by practically all mathematical theories used by "working mathematicians". It turns out that practically all the valid formulas that we need to prove in these theories are provable.

At the end of this section, we review the definitions of two important concepts related to formalized theories.

Def. *A formalized theory is called* ***consistent*** *if it has a model.*

The following theorem describes two critical properties — in fact, two alternative definitions — of consistent theories.

Theorems about consistency.

(1) A theory is consistent iff there is no valid formula for *in such that* |= for *and* |= ***not*** for.

(2) A theory is consistent iff there exists at least one invalid formula in it.

Inconsistent theories are of no scientific interest because, by (2), all their formulas are valid. In other words, in inconsistent theories, we can't distinguish between the truth and the falsity.

Def. *A formalized theory is called* ***complete*** *if it is consistent and for any formula,* for *either* |- for *or* |- **not** for.

This time, incomplete theories are usually more interesting than the complete ones, because they are more general. For instance, a formalized theory of groups is not complete, since on its grounds, we cannot prove that a group has exactly 15 elements (see [13], p. 292).

In formalized mathematics, and in particular in our investigations, we shall not "struggle" to make our theories complete. However, on the other hand, we should make our theories "sufficiently complete" to be able to prove the correctness of "sufficiently many correct metaprograms".

2.4 Formalized theories in an algebraic framework

One of our goals in this paper is to outline a general method for building a formalized theory of second-order logic, where we could describe the process of developing correct metaprograms in a **Lingua-V**-like language. Creating such a theory involves developing a language that we will call **Lingua-D** ("D" for "denotations"), which is sufficiently expressive to state and verify the correctness of metaprograms in **Lingua-V**. In this section, we will examine this task in an abstract scenario where:

- the source language, called **Language-V,** is given as a pair of algebras — of syntax **AlgSyn-V** and of denotations **AlgDen-V** — sharing a common signature and a unique homomorphism (the semantics) between them,
- the target language, called **Language-D,** is a language of a formalized second-order theory where we can talk about and prove the truth of valid formulas expressed in **Language-V**; also, this language will be identified by two algebras and a homomorphism, i.e., will have a denotational model.

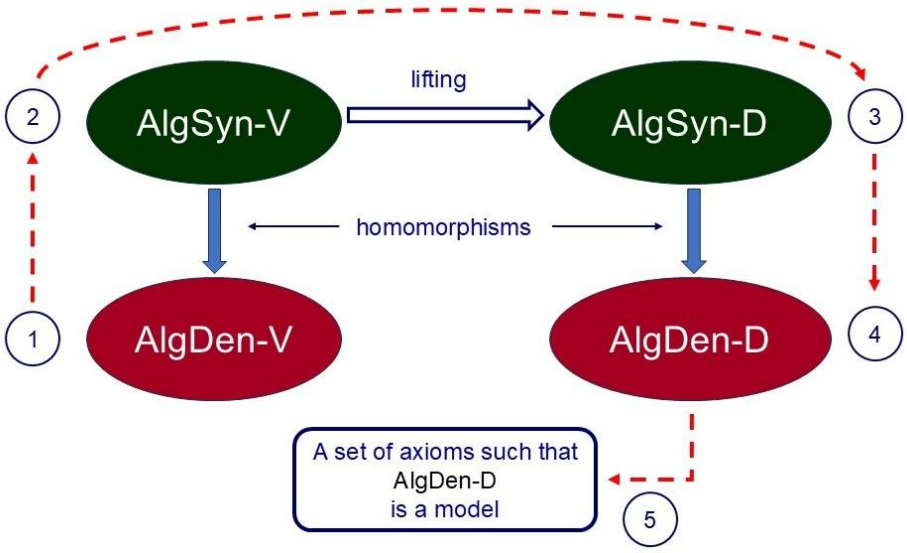


Fig. 2.4-1 The development of a Language-D for a programming Language-V

The transformation of a source language into a target language will be performed in five steps, as shown in Fig. 2.4-1.

1. The identification of an algebra of denotation **AlgDen-V** of some source language.
2. The identification of an algebra **AlgSyn-V** of abstract syntax for **AlgDen-V**. The elements of this language will represent ground terms and ground formulas of the future **Language-D**.
3. The transformation of the abstract syntax of the source language into an abstract syntax of the target language. Here we introduce variables, free terms, and free formulas. This transformation will be referred to as the *lifting* of a language of zero-order — one that only contains ground terms and ground formulas

— to a language of second-order, where variables may range not only over the elements of a particular universe, but also over functions and predicates defined on that universe.

4. The development of an algebra of denotations **AlgDen-D** as an *adequate generalization* of **AlgDen-V**; the concept of adequacy will be explained a little later.
5. The establishment of such a set of axioms that **AlgDen-D** constitutes one of its models.

We assume that all formalized theories investigated in the sequel will be based on standard inference rules sketched in Sec. 2.3. It should also be noticed that our formalized theories will be many-sorted, compared to one-sort theories investigated in Sec. 2.3 (only one universe). Now, instead of one universe, we have a family of carriers of a corresponding algebra of denotations.

Regarding the problem of axiomatizing **Language-D**, there are two basic strategies for building a set of axioms for a lifted language:

A. We formulate all axioms in **Language-D.** In this case, the set of axioms may be pretty large, and we have to make sure that it is consistent and "sufficiently complete". The latter practically means that we can prove the truth of "sufficiently many" valid formulas.

B. We define the carriers and constructors of **AlgDen-D** in some larger formalized theory, e.g., in an axiomatic set theory. In this case, the set of axioms is relatively small and known (from literature) to be consistent, but we have to formulate a large number of definitions. And, of course, we must enrich our lifted language by introducing new concepts, i.e., new carriers and constructors.

At this moment, we do not know (yet) which of these strategies will better fit our needs. We expect that some further research will clarify this point. As we mentioned earlier, we hope to develop our ecosystem, along with its **Language-D**, dynamically, as our experiences accumulate.

In this section, we shall concentrate on steps 3 and 4. Let the source language be given by two algebras with a common signature:

AlgSyn-V = (Sig-V, CarSyn-V, FunSyn-V, carSyn-V, funSyn-V) abstract syntax of **Language-V**
AlgDen-V = (Sig-V, CarDen-V, FunDen-V, carDen-V, funDen-V) denotations of **Language-V**
Sig-V = (Cn-V, Fn-V, arity-V, sort-V)

The only assumptions regarding this algebra are the following

boo : Cn-V,
and, or, implies, not : Fn-V; these operators are interpreted as classical connectives,
carDen.boo = Bool where Bool = {tt, ff}

Note that in **Lingua-V,** we use classical connectives in metaconditions, Kleene's connectives in conditions, and McCarthy's connectives in boolean expressions.

It should be noted that, in addition to carrier Bool, the family of carriers of AlgDen-V may include another "Boolean carrier", different from Bool, such as, e.g., BoolE = Bool | Error. However, the sort boo mentioned above is the sort of metaconditions rather than conditions!

Similar to programming languages, D-languages can also have an abstract, concrete, or colloquial syntax. Here, we primarily use abstract syntax because it provides a convenient framework for abstract algebras. Of course, once an abstract syntax of **Language-D** is developed, its syntax can be transformed into a concrete or colloquial version.

Assuming that the source algebras of **Language-V** are given, we build for them two derivative algebras of **Language-D**:

AlgSyn-D = (Sig-D, CarSyn-D, FunSyn-D, carSyn-D, funSyn-D) abs. syntax of **Language-D**
AlgDen-D = (Sig-D, CarDen-D, FunDen-D, carDen-D, funDen-D) den. of **Language-D**

The transformation from **Language-V** to **Language-D** will be called the *lifting* of a language, and the D-algebras will be called *lifted algebras.*

Let's assume that the abstract-syntax grammar of **Language-V** is the following:

Equations generating terms — one equation for every sort cn : Cn-V with cn ≠ boo:

```
ter : Term-V.cn =[6]
        fn(Term-V.cn-1,…,Term-V.cn-n) |  for all fn : Fn-V with
                                            arity.fn = (cn-1,…,cn-n)
                                            sort.fn = cn
```

One equation generating formulas

```
for : Form-V  =                                          we use Form-V instead of Term-V.boo
        pn(Term-V.cn-1,…,Term-V.cn-n)|  for all pn : Fn-V with
                                            arity.pn = (cn-1,…,cn-n)
                                            sort.pn = boo
        and(Form-V, Form-V)             |
        or(Form-V, Form-V)              |
        implies(Form-V, Form-V)         |
        not(Form-V)
```

Of course, for **Language-V** to be not empty, the set Fn-V of function names must include at least one zero-ary functional symbol.

Since our language does not have variables, it includes only ground terms and ground formulas. It may be said to be the language of a *zero-order theory*. As we know from [6], there exists a unique many-sorted homomorphism between our algebras:

SEM-V : **AlgSyn-V** ⟼ **AlgDen-V**

that we refer to as the *semantics* of this language.

To describe the transformation of **AlgSyn-V** to **AlgSyn-D**, let's assume that the future signature of D-language is the following:

Sig-D = (Cn-D, Fn-D, arity-D, sort-D)

The basic difference between **Language-V** and **Language-D** is such that the latter includes variables of three categories:

inv : IndVar — first-order individual variables running over the elements of the carriers of CarDen-V,
fuv : FunVar — second-order functional variables running over functions on such elements,
prv : PreVar — second-order predicational variables running over predicates on such elements.

We assume that each individual variable has a *sort* described by a function:

sort : IndVar ⟼ Cn-V

that indicates a carrier carDen-V.cn whose elements may be assigned to that variable in valuations, and that each second-order variable has an *arity* and a *sort*:

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

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

The functions of arities indicate the arities of functions/predicates that can be assigned to corresponding variables and the functions of sorts — the sort of their values. Now, with every sort cn : Cn-V, we define a family of variables of this sort

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

[6] We write Term.cn rather than Term.cn since the former is regarded as an "indivisible" metavariable in our fixed-point equations, rather than as a function that takes a sort name cn as an argument.

For every individual variable, we define a zero-ary constructor that creates this variable:

civ-cn-inv : ⟼ IndVar
civ-cn-inv.() = inv

Besides, we define two functions that make single-variable terms from variables:

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

Here mk-term-cn is a metaname of a function on strings of characters, whereas mk-term-cn is a concrete string of characters, i.e., it stands for itself. Consequently, mk-term-cn(inv) is a string of green characters followed by an individual variable and ending with a green bracket.

From the grammar of **AlgSyn-V,** we build a grammar of **AlgSyn-D**, or — more precisely — we build a grammar that will indicate that algebra. This new grammar is constructed from the former one by the following extensions:

1. adding one equation for each sort-name cn to generate the domain of individual variables of sort cn,
2. adding to each equation, generating terms of sort cn:
   a. one clause that generates the domain of single-variable terms of sort cn,
   b. for each functional variable fuv : FunVar with sort.fuv = cn one clause to generate a term with fuv as the main operation,
3. adding to the (unique) equation generating formulas:
   a. one clause that generates single-variable formulas,
   b. for each predicational variable prv : PreVar one clause to generate a formula with prv as the principal predicate,
   c. six clauses with quantifiers — two for each of the three categories of variables.

As we observe, the new grammar contains all the "content" of the previous one, plus some new equations (for variables) and additional clauses to the equations that generate terms and formulas. This partly explains our earlier claim that **Language-D** should be "an adequate generalization" of **Language-V**. However, this is not the only reason supporting that claim.

The new grammar is the following:

Equations generating individual variables — one equation for every sort name cn : Cn-V:

IndVar.cn =[7]
civ-cn-inv-1.() | civ-cn-inv-2.() | …

Equations generating terms — one equation for every sort name cn : Cn-V with cn ≠ boo:

ter : Term-D.cn =
mk-term(IndVar.cn) |
fn(Term-D.cn-1,…,Term-D.cn-n) | for every fn : Fn-V with arity.fn = (cn-1,…,cn-n) sort.fn = cn
fuv(Term-D.cn-1,…,Term-D.cn-n) | for every fuv : FunVar with arity.fuv = (cn-1,…,cn-n) sort.fuv = cn

One equation generating formulas

for : Form-D = we use Form-D instead of Term-D.boo
fn(Term-D.cn-1,…,Term-D.cn-n) | for every fn : Fn-V with arity.fn = (cn-1,…,cn-n) sort.fn = boo
prv(Term-D.cn-1,…,Term-D.cn-n) | for every prv : PreVar with

[7] We do not write IndVar.cn but IndVar.cn since the latter is regarded as an "indivisible" metavariable in our set of fixed-point equations.

arity.prv = (cn-1,…,cn-n)
sort.prv = boo

and(Form-D, Form-D) |
or(Form-D, Form-D) |
implies(Form-D, Form-D) |
not(Form-D) |
(∀*i* IndVar) Form-D |
(∃*i* IndVar) Form-D |
(∀*f* FunVar) Form-D |
(∃*f* FunVar) Form-D |
(∀*p* PreVar) Form-D |
(∃*p* PreVar) Form-D

In the quantified formulas above, we introduce two quantifiers for each of the three sorts of variables. For example, (∀*i*) is a quantifier for individual variables, hence (∀*i* ide) is understood as an individual-variable quantification of the individual variable ide.

Assume that the **AlgSyn-D** is implicit in this grammar (see Sec. 2.15 of [6]). At this moment, we may identify the common signature of both lifted grammars:

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

Here, IndVar.cn symbolically denotes the name of the carrier IndVar.cn. The set of names of functions is the following:

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

Here, functional/predicational variables are regarded as the names of functions creating terms/formulas in **AlgSyn-D** and the denotations of terms/formulas in **AlgDen-D**. This will be seen a little later.

The last step in our lifting process is the generation of **AlgDen-D**. To do that, we first define the domains of valuations. Let:

```
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
```

We assume that the domains of valuations include all total functions on variables which are *sort-wise well-formed*, which means that for every valuation vlu:

```
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.prv = (cn-1,…,cn-n)
then  vlu.prv : carDen-V.cn-1 x … x carDen-V.cn-n ⟼ carDen-V.boo
```

Next, with every sort cn : Cn-D we associate a corresponding *domain of denotations*, thus defining the function carDen-D:

carDen-D.cn = Valuation ⟼ carDen-V.cn for all cn : Cn-V
carDen-D.formula = Valuation ⟼ carDen-V.boo
carDen-D.IndVar.cn = IndVal.cn for all cn : Cn-V, variables are the denotations of themselves

In this moment, we are ready to define the interpretation function funDen-D. We define it case-by-case:

(1) For every name civ-cn-inv of a variable-creating function, its interpretation is the corresponding variable-creating function:

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

(2) For every name mk-term-cn of the **term-making function**:

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

The denotation of a term that consists of a single individual variable, inv, is a function that, when applied to a valuation, vlu, returns the value assigned to that variable in this valuation.

(3) For every **functional name** fn : Fn-V with arity.fn = (cn-1,…,cn-n) and sort.fun = cn:

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

Note that at the right-hand side of the equation, we refer to the meaning of fn in **Language-V**. That is the second argument to say that **AlgDen-D** is an *adequate generalization* of **AlgDen-V**.

(4) For every **functional variable** fuv : FunVar with arity.fuv = (cn-1,…,cn-n) and sort.fuv = cn:

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

(5) For the name mk-formula of the **formula-making function**:

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

(6) and (7) The cases of the names of predicates and of predicational formulas are analogous to (3) and (4), and, therefore, we shall not repeat them.

The cases concerning propositional operators and quantifiers are routine; therefore, we present just two examples.

(8) For **conjunction**

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

where and is a classical two-valued conjunction.

(9) For a **general quantifier**

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

Our definition identifies a unique homomorphism

SEM-D : **AlgSyn-D** ⟼ **AlgDen-D**

that we call the semantics of **Language-TF**. Now we can easily prove the last fact that justifies calling **Language-TF** an "adequate generalization" of **Language-V**. First, note that every term or formula of **Language-V** is a (ground) term or formula of **Language-D**. The second fact is that for every carrier name cn : Cn-V, every term ter : Term-V.cn, and every valuation val : Valuation.

SEM-D.cn.ter.val = SEM-V.cn.term.

In other words, the denotations of terms in **Language-V** are "compatible" with those in **Language-D**.

At the end of this section, a general observation about our approach to constructing theories and their models is appropriate. In typical textbooks of mathematical logic, the language and axioms of a formalized theory are presented first, and only then are the associated models examined. However, "a working mathematician" usually proceeds in reverse — they construct a model first and often leave the task of its axiomatization to colleagues in formal logic departments. That is also our perspective. We start with a denotational model of a programming language, where the algebra of syntax defines the language of the theory, and the algebra of denotations represents its model. Then, we aim to identify a set of axioms such that our algebra of denotations is one of its models. The existence of this model guarantees that our theory is consistent.

2.5 Formalized Peano's arithmetic

Let's illustrate our method on the example of second-order Peano's arithmetic, this time seen from an algebraic perspective. As **AlgDen-V**, we chose a standard zero-order model of this theory (only ground terms and ground formulas). Let:

```
nat  : Natural  = {0, 1, 2,…}
boo  : Bool     = {tt, ff}
```

with the following functions:

```
zer  :                    ⟼ Natural;  constant zero
suc  : Natural            ⟼ Natural;  suc.nat = nat + 1
num  : Natural            ⟼ Bool;     num.nat = tt  iff      nat is a natural number
equ  : Natural x Natural  ⟼ Bool;     equ.(nat-1, nat-2)  iff  nat-1= nat-2
```

The signature of this algebra is the following:

```
Sig-V = ({nat, boo}, {zer, suc, num, equ}, arity, sort}

arity.zer   = ()
sort.zer    = nat

arity.suc   = (nat)
sort.suc    = nat

arity.num   = (nat)
sort.num    = boo

arity.equ   = (nat, nat)
sort.equ    = boo
```

The following grammar describes the abstract-syntax algebra of the corresponding Language-V:

```
ter-V : Term-V =                                  ground terms
          zer() | suc(Term-V)

for-V : Form-V =                                  ground formulas
          num(Term-V) | equ(Term-V, Term-V)
```

To build a grammar of a second-order **Language-D**, we first introduce three sets of variables.

```
IndVar.nat   = {x, y, z}     individual decimal variables
IndVar.boo   = {a, b, c}     individual boolean variables
PreVar       = {P}           one predicational variable
```

with

```
sort.x    = nat
…
sort.a    = boo
…
sort.P    = boo
arity.P   = (nat)
```

Note that we introduce only one predicational variable and no functional variables. A practical rule in this case is such that we introduce only as many second-order variables as we shall need to formulate our axioms.

With individual variables, we introduce corresponding constructors:

```
civ-nat-x : ⟼ {x, y, z}
civ-nat-x.() = x
…
```

Note that we do not introduce a constructor for P since we do not introduce a syntactic domain of the denotations of second-order variables. In our example, P never appears alone — always as a part of a formula, e.g., P(x). The following grammar then describes the abstract syntax of Language-D:

```
inv-D : IndVar.nat =
            civ-nat-x.() | civ-nat-y.() | civ-nat-z.()

inv-D : IndVar.boo =
            civ-boo-a.() | civ-boo-b.() | civ-boo-c.()

ter-D : Term-D =
            mk-term(IndVar.nat) | zer() | suc(Term-D)

for-D : Form-D =
            mk-formula(IndVar.boo) | num(Term-D) | P(Term-D) | equ(Term-D, Term-D) |
            and(Formula-D , Formula-D) | …
```

The domain of valuations and the corresponding domains of denotations are the following:

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

The domains of denotations:

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

carDen-D.nat      = TerDen  = Valuation ⟼ Natural     the name of the domain of terms is nat
carDen-D.boo        = ForDen  = Valuation ⟼ Bool      the name of the domain of formulas is boo
```

Examples of definitions of function in **AlgDenFT** are the following:

```
funDen-D.civ-nat-x() : ⟼ TerDen                       i.e.
funDen-D.civ-nat-x.() = x

funDen-D.mk-term : IndVar.nat ⟼ TerDen
funDen-D.mk-term : IndVar.nat ⟼ Valuation ⟼ Natural
funDen-D.mk-term.inv.vlu = vlu.inv

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

The signature of the lifted algebra is the following:

Sig-D = ({nat, boo}, {civ-nat-x() ,…, civ-boo-a() ,…, mk-term,
zer, suc, num, equ, and, or, not, implies, ∀i, ∃i, ∀p, ∃p}, arity, sort}

3 Formalizing Lingua-V

3.1 The grammar of Lingua-V

Since in [6] conditions and metaconditions were defined by examples only, we have to complete the grammar of **Lingua-V** by corresponding equations. Similarly to the case of **Lingua**, we shall not attempt to define a fully developed "practical language", but we restrict our attention to a few typical clauses. We shall omit the prefixes Abs (for "abstract) or Con (for "concrete") since we shall now consider only one version of our grammars. However, we continue to use postfixes -V and -D to distinguish between the two languages. The grammar of **Lingua-V** is built by adding to the grammar of **Lingua** equations corresponding to the following syntactic categories (we slightly modify the metanames compared to those used in [6]):

```
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
```

Below, we focus on conditions and metaconditions, as the remaining categories are defined in [6].

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

- they include several predicational symbols that are not available for value expressions,
- logical connectives used in compound conditions represent Kleene's rather than McCarthy's operators.

A scheme of an equation defining the category of conditions may be, therefore, the following (cf. Sec. 7.2.4 of [6]):

```
con : Con-V =

    duplicates of atomic boolean expressions of Lingua
      con-equal-int(ValExp , ValExp)          |   equal integers
      con-less-int(ValExp , ValExp)           |   less than, integers
      …
    conditions with predicates that are not available for value expressions
      con-is-typ(Identifier, TypExp)          |   identifier declared as a type constant
      con-var-is-typ(Identifier, TypExp)      |   declared variable of a given type
      con-proc-opened(Identifier, Identifier) |   opened procedure
      …
    algorithmic conditions
      con-left-algorithmic(SpePro-V , Con-V)  |   SpePro-V — specprograms
      con-right-algorithmic(Con-V , SpePro)   |

    compound conditions with Kleene's operators
      con-or-k(Con-V, Con-V)                  |
      con-and-k(Con-V, Con-V)                 |
      con-not-k(Con-V)
```

In the case of algorithmic conditions, we have ad hoc transformed the corresponding grammatical equation from Sec. 9.2.6 of [6] into a prefix form. A scheme of an equation defining the category of metaconditions may be the following:

```
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)	|	
…		

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)		

3.2 The denotations of Lingua-V

The algebra of denotations of **Lingua-V** is a direct extension of **AlgDen** described in [6] by the carriers corresponding to new syntactic categories and the corresponding constructors. The new carriers are:

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

The signatures of corresponding constructors can be derived from grammatical clauses of Sec. 3.1, e.g.:

```
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
```

and similarly for the denotations of metaconditions:

```
mcd-stronger           : ConDen-V x ConDen-V        ⟼ MetConDen-V     i.e.,
mcd-stronger           : ConDen-V x ConDen-V        ⟼ {tt, ff}
```

mcd-weak-equivalent	: ConDen-V x ConDen-V	⟼ {tt, ff}
mcd-less-defined	: ConDen-V x ConDen-V	⟼ {tt, ff}
mcd-strong-equivalent	: ConDen-V x ConDen-V	⟼ {tt, ff}
mcd-insures-LR	: ConDen-V x InsDen-V	⟼ {tt, ff}
mcd-resilient	: ConDen-V x SpeProDen-V	⟼ {tt, ff}
mcd-primary	: ConDen-V x MetPro-V	⟼ {tt, ff}
mcd-induced	: ConDen-V x MetPro-V	⟼ {tt, ff}

Formalized definitions of these constructors can be easily inferred from their definitions in Sec. 9.2 and Sec. 9.3 of [6].

4 Defining Lingua-D

4.1 Individual variables in Lingua-D

Proceeding to **Lingua-D**, we define, first, the domains of variables. Let's start with individual variables, which for every sort of **Lingua-V** we define as a separate carrier. For simplicity, we write the corresponding grammatical equations in a concrete-syntax style:

ide	: IdeVar-D	= {ide,…}	variables corresponding to identifies,
vex	: ValExpVar-D	= {vex,…}	variables corresponding to value-expressions,
rex	: RefExpVar-D	= {rex,…}	variables corresponding to reference-expressions,
sin	: SpeInsVar-D	= {sin,…}	variables corresponding to specinstructions,
con	: ConVar-D	= {con, prc, poc,…}	variables corresponding to conditions,
…			
mec	: MetConVar-D	= {mec,…}	variables corresponding to metaconditions.

We also assume that all these individual variables may be "decorated" with arbitrary prefixes and postfixes. Note that individual variables of **Lingua-D** are printed in Arial underlined to distinguish them from non-underlined metavariables running over the syntactic domains of this language. For instance, vex is a metavariable running over the syntactic domain ValExp-D, whereas vex is an element of the syntactic domain ValExpVar-D. As we will see a little later, this distinction is essential.

4.2 Functional and predicational variables in Lingua-D

Whereas the introduction of individual variables is, in a sense, "unavoidable" — for every carrier name cn : Cn-VT, we introduce one carrier of variables — the situation with non-individual variables is different:

- they have not only sorts but also arities and, therefore, within every sort of such variables we may have a variety of variables with different arities,
- which second-order variables we shall need, will be seen only when we start writing second-order axioms.

We postpone, therefore, the introduction of second-order variables till the moment when they are needed. A case where we will undoubtedly need second-order variables is the second-order axioms for integers. We must ensure that all models of our formalized theory include standard models of integers, which are necessary to prove the termination properties of programs.

4.3 Terms in Lingua-D

According to the rule described in Sec. 2.4, grammatical equations of **Lingua-D** are created from corresponding equations of the source language by adding to each of them three categories of clauses:

1. one clause for the creation of single-variable terms with individual variables,
2. one clause for every functional variable that creates a term with this variable representing the root operation,

3. one clause for every predicational variable that creates a formula with this variable representing the root predicate.

Since we have not introduced second-order variables so far, the modification of grammatical equations is limited to point 1. We also slightly modify the (green) names of constructors to make them more intuitive[8], and we use individual variables associated with syntactic categories, e.g., vex, as metavariables running over these categories (cf. Sec. 2.1.1). Let's see a few examples (cf. Sec. 7.2 in [6]):

value expressions

```
vex : ValExp-D =
        vex-make-vex(ValExpVar-D)                  |   single-variable term
        vex-bo(BooleanSyn-D)                       |
        vex-in(IntegerSyn-D)                       |
        vex-re(RealSyn-D)                          |
        vex-te(TextSyn-D)                          |
        vex-variable(Ide-D)                        |   single-identifier value-expression
        vex-attribute(ValExp-D , Ide-D)            |
        vex-call-fun-pro(Ide-D, Ide-D, ActPar-D)   |
        vex-add-int(ValExp-D , ValExp-D)           |
        vex-less-int(ValExp-D , ValExp-D)          |
        vex-or-m(ValExp-D , ValExp-D)              |   McCarthy's alternative
        vex-create-li(ValExp-D)                    |
        vex-get-from-rc(ValExp-D , Ide-D)          |
        …
```

Note that we now have to suffix all the names of syntactic domains with -D, since they differ from those in **Lingua-V**.

specified instructions

```
sin : SpeIns-D =
        sin-make-sin(SpeInsVar-D)                          |   single-variable term
        sin-make-asr(Con-D)                                |   assertions[9]
        sin-skip-ins()                                     |
        sin-assign(RefExp-D , ValExp-D)                    |
        sin-call-imp-pro(Ide-D , Ide-D , ActPar-D , ActPar-D)  |
        sin-call-obj-con(Ide-D , Ide-D , ActPar-D)         |
        sin-if(ValExp-D , SpeIns-D , SpeIns-D)             |
        sin-if-error(ValExp-D , SpeIns-D)                  |
        sin-while(ValExp-D , SpeIns-D)                     |
        sin-compose-ins(Ins-D , Ins-D)
```

identifiers

```
ide : Ide-D =
        IdeVar-D      |
        Identifier
```

We recall that, according to our earlier convention, the elements of IdeVar-D are printed in black Arial, and the elements of Identifier are printed in green Arial Narrow. We also bring to the attention of our readers that identifiers in **Lingua-D** belong to the category of terms.

conditions

```
con : Con-D =
        con-make-con(ConVar-D  )          |   single-variable term
```

[8] For instance, we replace the name prefix ved- (value-expression denotation) by vex- (value expression).

[9] Our reader may guess why we abbreviate "assertion" as "asr" rather than "ass".

```
    con-or-k(Con-D , Con-D)                 |  Kleene's alternative
    con-less-int(ValExp-D , ValExp-D)       |
    con-is-value(ValExp-D)                  |
    con-is-free(Ide-D)                      |
    con-left-algorithmic(SpeIns-D , Con-D)  |
    con-right-algorithmic(Con-D , SpePro-D) |
    …
```

In the end, the domain of terms is defined as a union of all sort-oriented term domains, i.e.:

```
ter : Term-D = ValExp-D | SpeIns-D | Ide-D | Con-D | …
```

Examples of ground D-terms are the following (for simplicity, we omit the name of identifier-creating constructors):

```
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
```

and examples of free terms are the following

```
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)))
```

4.4 Formulas in Lingua-D

Formulas in **Lingua-D** are metaconditions and their patterns, e.g.:

```
mec : MetCon-D =
    mec-make-mec(MetConVar-D)
    mec-stronger(Con-D ,  Con-D)                |
    mec-weakly-equivalent(Con-D , Con-D)        |
    mec-less-defined(Con-D , Con-D)             |
    mec-strongly-equivalent(Con-D , Con-D)      |
    mec-insures LR(Con-D , SpeIns-D)            |
    mec-hereditary(Con-D , MetPro-D)            |
    mec-immunizing(Con-D)                       |
    mec-metaprogram(Con-D , SpePro , Con-D)     |
    …
    mec-and(MetCon-D , MetCon-D)                |  classical conjunction
    mec-or(MetCon-D , MetCon-D)                 |
    mec-implies(MetCon-D , MetCon-D)            |
    mec-not(MetCon-D)
```

We recall that the logical operators in the above equations are 2-valued classical connectives and, therefore, we write them without suffixes -m or -k. Examples of ground formulas in our theory are the following (for convenience, we write them in concrete syntax):

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

or

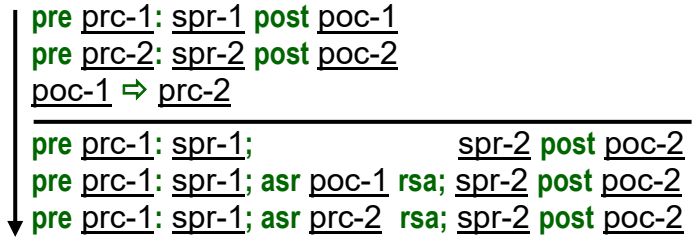
```
pre nni(x, k) and-k n+1 ≤ M:
    x := 0;
    while x+1 ≤ n
        do
            x := x+1
        od
```

post x = n

In turn, examples of free formulas, written in concrete syntax, are metaprogram construction rules such as

pre sin @ con
  sin
post con

or



We recall that above the line and below the line, we have classical conjunctions of formulas, and the vertical arrow represents classical implication. Another example of a free formula in **Lingua-D** may be

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

Of course, all these formulas are valid.

4.5 The denotations of Lingua-D

The algebra of denotations of **Lingua-D** can be "algorithmically" derived from the algebra of **Lingua-V** in a way described in Sec. 2.4. In this section, we only sketch a way of doing this. Let's start with valuations:

uni : Universe = IdeDen-D | 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-D,
then vlu.vex : ValExpDen-V.

At this moment, we do not introduce 2nd-order variables. The carriers of denotations in **Lingua-D** are the following:

ved : ValExpDen-D = Valuation ⟼ ValExpDen-V
red : RefExpDen-D = Valuation ⟼ RefExpDen-V
ind : InsDen-D = Valuation ⟼ InsDen-V
spd : SpeProDen-D = Valuation ⟼ SpeProDen-V
ide : IdeDen-D = Ide-D
cod : ConDen-D = Valuation ⟼ ConDen-V
…
mcd : MetConDen-D = Valuation ⟼ MetConDen-V

An example of a constructor of D-denotations may be the following:

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

where ind-assign-v is a denotational constructor of instructions from **Lingua-V**. Another exemplary constructor builds the denotation of a free metaprogram formula:

mcd-metaprogram-D : ConDen-D x SpeProDen-D x ConDen-D ⟼ MetConDen-D i.e.

```
mcd-metaprogram-D : ConDen-D x SpeProDen-D x ConDen-D ⟼ Valuation ⟼ {tt, ff}
mcd-metaprogram-D.(cod-1, spd, cod-2).val =
    mcd-stronger.(cod-1.val, con-left-algorithmic.(spd.val, con-2.val))
```

This definition, written in the metanotation used in Sec. 9.2.7 and Sec. 9.3.2 of [6], is as follows:

```
mcd-metaprogram-D.(cod-1, spd, cod-2).val = cod-1.val ⇨ (spd.val)@(con-2.val)
```

Note that we are using non-underlined metavariables in this context.

5 A formalized theory of the denotations of Lingua-V

5.1 The structure of the theory

We assume that the denotational model of our source language, i.e., of **Lingua-V**, is represented by two algebras, **AlgSyn-V** and **AlgDen-V**, with a homomorphism (semantics) between them (cf. Sec. 2.4). Our goal is to construct such a **D-theory** whose language **Lingua-D** includes **AlgSyn-V** — this has already been done in Sec. 4 — and whose set of models includes (an extension of)[10] **AlgDen-V**.

As we will see, in our target **D-theory**, the categories of axioms and inference rules will be split into several subcategories. The overall structure of this theory will be the following:

1. Language: **Lingua-D**.
2. Axioms:
   a. axioms describing abstract mathematical entities such as numbers, sets, functions, etc.,
   b. axioms describing values in **Lingua-V,** such as integer, real, and boolean values, list values, array values, objects, etc.,
   c. axioms describing **Lingua-V** denotations. i.e., the denotations of expressions, instructions, conditions, metaconditions, etc.
3. Inference rules:
   a. universal rules — substitutions, detachment, generalization, etc.,
   b. standard rules — rules expressible by axioms,
   c. non-standard rules — rules not expressible by axioms.

Universal rules are significant for proving theorems of our theory, by which we mean that the removal of any of them would make some valid formulas not provable. Standard rules are not significant, but, as we will see, they may substantially speed up the process of program development. Non-standard rules play a similar role to standard rules, but further research may be necessary to determine if they are significant or not.

Regarding axioms, we shall concentrate on their third group, as the first group is pretty well-known in mathematical logic, and the second should be easily derivable from them. In turn, the third group is highly dependent on **Lingua-V**.

5.2 Denotation-oriented axioms

5.2.1 Program-independent axioms for metaconditions

Dependencies between metapredicates

(con1 ≡ con2) **iff** ((con1 ⊑ con2) **and** (con2 ⊑ con1))
(con1 ⇔ con2) **iff** ((con1 ⇨ con2) **and** (con2 ⇨ con1))
(con1 ≡ con2) **implies** (con1 ⇔ con2)

Definitions of ternary metapredicates

(con1 ≡ con2 **whenever** con) **iff** ((con **and-k** con1) ≡ (con **and-k** con2))
(con1 ⇔ con2 **whenever** con) **iff** ((con **and-k** con1) ⇔ (con **and-k** con2))
(con1 ⇨ con2 **whenever** con) **iff** ((con **and-k** con1) ⇨ (con **and-k** con2))

Relations ≡ and ⇔ are equivalences in the set of conditions

con ≡ con
(con1 ≡ con2) **implies** (con2 ≡ con1)

[10] A target model of **D-theory** may include, as an algebra, more carriers and more constructors that may be needed to formulate some axioms.

…

Relation ≡ is a congruence for and-k, or-k, and not-k

(con1 ≡ con2) **implies** ((con **and-k** con1) ≡ (con **and-k** con2))

…

Relation ⇔ is a congruence for and-k and **or-k**

(con1 ⇔ con2) **implies** ((con **and-k** con1) ⇔ (con **and-k** con2))

…

Operators and-k and or-k are strongly and weakly commutative

(con1 **and-k** con2) ≡ (con2 **and-k** con1)
(con1 **or-k** con2) ≡ (con2 **or-k** con1)
(con1 **and-k** con2) ⇔ (con2 **and-k** con1)
(con1 **or-k** con2) ⇔ (con2 **or-k** con1)

The operator and-k is both strongly and weakly left-hand-side and right-hand-side distributive with respect to or-k and vice versa.

(con1 **and-k** (con2 **or-k** con3)) ≡ ((con1 **and-k** con2) **or-k** (con1 **and-k** con3))
(con1 **or-k** (con2 **and-k** con3)) ≡ ((con1 **or-k** con2) **and-k** (con1 **or-k** con3))
((con2 **or-k** con3) **and-k** con1) ≡ ((con2 **and-k** con1) **or-k** (con3 **and-k** con1))

…

De Morgan's laws for and-k and or-k and for the negation of quantifiers are satisfied with strong equivalence and weak equivalence

not (con1 **and-k** con2) ≡ (**not**(con2) **or-k not**(con1))
not (con1 **and-k** con2) ⇔ (**not**(con2) **or-k not**(con1))

…

The nearly-true condition

error-transparent(con) **implies** (con ⇨ **NT**)

The relationship between the three implications:

error-sensitive(con1) **and** ((con1 **implies-k** con2) ≡ **NT**) **implies** (con1 ⇨ con2))

5.2.2 Axioms corresponding to behavioral metaconditions

In this group, we show three examples of axioms:

different(ide1, ide2)
———————————————
(ide1 **is free**) **irrelevant for** (**let** ide2 **be** tex)

different(ide1, ide2)
———————————————
(ide1 **is** tex1) **irrelevant for** (**let** ide2 **be** tex2)

pre prc : sin **post** poc
con **irrelevant for** sin
———————————————
pre prc **and-k** con : sin **post** poc **and-k** con

In all three cases, we have used a diagrammatic notation that improves the readability of formulas; however, formally, they are implicative formulas in **Lingua-D**. We assume that the ad hoc introduced formula **different**(ide1, ide2) is satisfied for a valuation vlu iff vlu.ide1 ≠ vlu.ide2, where ≠ compares two strings of characters.

Of course, to use this formula in our formalized theory, we have to characterize it axiomatically. One such axiom will be

not different(ide, ide**).**

However, as we will see in Sec. 6.4.4.2, the axiomatization of **different** may be replaced by an "implemented" procedure that compares two texts. For the denotation of **irrelevant,** see Sec. 9.3.4 of [6].

5.2.3 Axioms corresponding to temporal metaconditions

See Sec. 9.3.5 of [6].

not(ide **is free) hereditary in** mpr
(ide **is free) co hereditary in** mpr
(ide **is** tex**) hereditary in** mpr

5.2.4 Axioms corresponding to declarations

These axioms are formulated as lemmas in Sec. 9.4.4 of [6]. Correct metaprograms are constructed using these rules and the substitution inference rule.

Axiom for variable declaration

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

Axiom for an abstract attribute declaration

pre (ide-at **is free) and-k (**ide-cl **is class) and-k (**tex **is type) :**
  let ide-at **be** tex **with** yex **as** pst **tel in** ide-cl
post att ide-at **is** tex **in** ide-cl **as** pst

Axiom for a type constant declaration

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

Axiom for an imperative pre-procedure declaration

pre (ide-pr **is free) and-k (**ide-cl **is class)**
  proc ide-pr **(val** my-fpc-v **ref** my-fpc-r**)** my-body **in** ide-cl**;**
post pre-proc ide-pr **(val** my-fpc-v **ref** my-fpc-r**)** my-body **imperative in** ide-cl

Axiom for a declaration of a funding class

pre : (ide-cl **is free) and-k (**cli **is class)**
  class ide-cl **parent** cli **with skip-ctr ssalc**
post ide-cl **child of** cli

Axiom for class declaration

pre prc : **class** ide **parent** cli **with skip-ctr ssalc post** pa-poc
pre pa-poc : ctr-1 **in** ide **post (**pa-poc **and-k** cr-poc-1**)**
pre (pa-poc **and-k** cr-poc-1**)** : ctr-2 **in** ide **post (**pa-poc **and-k** cr-poc-1 **and-k** cr-poc-2**)**
...

---

pre prc**:**
  class ide **parent** cli **with** ctr-1; … ; ctr-k **ssalc**
post pa-poc **and-k** cr-poc-1 **and-k** cr-poc-2 **and-k** …

Here we have a scheme of an axiom whose parameter is the sequence of class transformation variables

ctr-1; … ;ctr-k

in the class declaration.

Axiom for the opening of procedures

```
pre
   pre-proc ide-pr-11 (val fpc-v-11 ref fpc-r-11) body-11 imperative in ide-cl-1 and-k
   pre-proc ide-pr-12 (val fpc-v-12 ref fpc-r-12) body-12 imperative in ide-cl-1 and-k
   …
   pre-proc ide-pr-21 (val fpc-v-21 ref fpc-r-21) body-21 imperative in ide-cl-2 and-k
   pre-proc ide-pr-22 (val fpc-v-22 ref fpc-r-22) body-22 imperative in ide-cl-2 and-k
   …
   open procedures
post
   ide-cl-1.ide-pr-11 opened and-k
   ide-cl-1.ide-pr-12 opened and-k
   …
   ide-cl-2.ide-pr-21  opened and-k
   ide-cl-2.ide-pr-22  opened and-k
   …
```

Here, as well, we have a scheme of an axiom, and this time the parameter is the number of pre-procedure declarations.

5.2.5 @-axiom

There is only one axiom in this group (Sec. 9.4.6.2 of [6]):

```
pre sin @ con :
   sin
post con
```

5.2.6 Some standard implicative axioms

See Sec. 9.4.4 of [6].

Axiom for a final composition

```
pre prc                              : spp                 post (de-con and-k sp-con)
pre (de-con and-k sp-con)            : open procedures     post (de-con and-k op-con and-k sp-con)
pre (de-con and-k op-con and-k sp-con) : sin               post (de-con and-k op-con and-k si-con)
-------------------------------------------------------------------------------------------------
pre prc:
   spp ; open procedures ; sin
post (de-con and-k op-con and-k si-con)
```

Axiom for strengthening preconditions

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

Axiom for weakening postconditions

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

Axiom for conjunction and disjunction of conditions

```
pre prc-1 : spr  post poc-1
pre prc-2 : spr  post poc-2
-------------------------------------------------
pre (prc-1 and-k prc-2) : spr  post (poc-1 and-k poc-2)
pre (prc-1 or-k prc-2)  : spr  post (poc-1 or-k poc-2)
```

Axiom for the propagation of an irrelevant condition

```
pre prc: spr post poc
con irrelevant for spr
-------------------------------------------------
pre (prc and-k con) : spr post (poc and-k con)
```

5.2.7 Implicative axioms for structural instructions

Axiom for sequential composition

```
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
```

Axiom for conditional branching if-then-else-fi

```
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
```

Axiom for loop while-do-od

```
pre (inv and-k vex) : sin post inv
inv insures LR of asr vex rsa ; sin
prc ⇨ inv
inv ⇨ (vex or-k (not-k vex))
inv and-k (not-k vex) ⇨ poc
-------------------------------------------------
pre prc : asr inv rsa ; while vex do sin od post poc
```

5.3 Inference rules

5.3.1 Program-building rules versus inference rules

It is essential for our further investigations to understand a subtle difference between program-building construction rules and lemma-proving inference rules.

The former are formulated in a non-formalized **M-theory**, whose language is **MetaSoft** and where we freely use arbitrary mathematical tools, such as set theory, classical logic, the theory of relations, and CPOs, among others. It is formal, but not formalized. Construction rules of metaprograms are written in **MetaSoft** using a diagrammatic notation like:

pre prc : spr **post** poc
prc1 ⇨ prc
---
pre prc1 : spr **post** poc (5.3.1-1)

Formally, however, it is just a special presentation of a "usual" implicative formula:

((pre prc : spr **post** poc**) and (**prc1 ⇨ prc**)) implies (pre** prc1 : spr **post** poc**)**

where logical connectives are classical.

Since this rule has been proved sound in [6], we may assume it to be an axiom of **D-theory**. However, since an axiom must be a formula in **Lingua-D,** we rewrite our **MetaSoft** formula to the **Lingua-D** form:

(1) **((pre** prc : spr **post** poc**) and (**prc1 ⇨ prc**)) implies (pre** prc1 : spr **post** poc**)**

Assume further that in the set of axioms we have the following tautology:

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

Let now for some concrete prc, spr, poc, and poc1 (note that these metavariables are not underlined), the following metaconditions are valid:

(3) **pre** prc **:** spr **post** poc
(4) prc1 ⇨ prc

By substituting in (1) prc → prc, spr → spr, etc., we may include the following formula in the set of lemmas:

(5) **((pre** prc : spr **post** poc**) and (**prc1 ⇨ prc**)) implies (pre** prc1 : spr **post** poc**)**

Note that (5) does not represent itself, like (1), but a concrete formula (free or ground) in **Lingua-D**. By substituting in (2)

pre prc **:** spr **post** poc → mec1
prc1 ⇨ prc → mec2

we may include

((pre prc : spr **post** poc**) and (**prc1 ⇨ prc**))**

in the set of lemmas, hence, from (5) by detachment, we include in the set of lemmas the metacondition

pre prc1 : spr **post** poc.

Assume now that at the level of **M-theory,** the metaformula

|- for

expresses the fact that the formula for is a theorem or axiom in the **D-theory**. On the grounds of **M-theory,** our reasoning described above may be expressed by the following metaformula:

|- **(pre** prc : spr **post** poc**)** and |- (prc-1 ⇨ prc) implies |- **(pre** prc-1 : spr **post** poc)

where red-typed operators belong to the **M-theory**. This metaimplication, written in diagrammatic notation, looks as follows:

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

|- **pre** prc-1 : spr **post** poc (5.3.1-2)

Note that this metaformula is true only under the condition that (1) is an axiom. Note also that we have not underlined variables in the rule, because our rule is a formula in **M-theory**. This observations is, of course, true for all inference rules of **D-theory**.

Observe the difference between (5.3.1-1) and (5.3.1-2). The former represents an implication expressible by an axiom in **D-theory**. The latter says that for arbitrary concrete prc, spr, poc, and poc1, if

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

are lemmas (in a repository), then the formula

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

may be added to the set of lemmas (to the repository). From the perspective of **D-theory**, the former is an axiom, whereas the latter may be regarded as an inference rule. Although our two rules look similar, they belong to two different worlds.

From a formal viewpoint, rule (5.3.1-2) does not add anything to the "proving power" of **D-theory**. In every concrete situation, instead of using this rule, we could have repeated the construction (the reasoning) that leads to that rule. However, from a practical perspective, it enables a significant shortening of the process of developing correct metaprograms within our ecosystem.

In our example, we proceeded from a program-building rule in **M-theory** to an axiom in **D-theory** and from that axiom to a lemma in **M-theory**, which may be regarded as an inference rule in **D-theory**. Our "construction trace" was, therefore, the following:

informal construction rule → *formal axiom* → *formal inference rule.*

Such meta-construction may be repeated for all these program-building rules that are expressible as implications in the **D-theory**. Note that these rules, being formulas in **D-theory**, may be proved sound on the grounds of this theory, hence, in particular, using our future theorem prover.

It turns out, however, that not all program-building rules of **M-theory** may be transformed in that way into inference rules of **D-theory**. That is the case for rules that can't be expressed as formulas in **D-theory.** For example, a rule which says (cf. Lemma 9.4.3-3 in Sec. 9.4.3 of [6]) that

in every correct metaprogram we can replace any assertion **asr** con **rsa** *by* **asr** con1 **rsa** *under the conditions that* con ⟺ con1,

can't be expressed as an axiom (see Sec. 5.3.5 for more examples). In such a case, if we want to have that rule "available" in **D-theory**, we have to introduce it as a non-standard inference rule (see again Sec. 5.3.5). In other words, we skip the formal-axiom stage and proceed directly to an inference rule:

informal construction rule → *formal inference rule.*

In that case, we can't use our theorem prover to prove the soundness of such a rule. We have to prove it "by hand" on the grounds of **M-theory**.

5.3.2 Universal inference rules

For the case of abstract formalized theories, universal inference rules were sketched in Sec. 2.3. The rule of detachment may be included in our **D-theory** (cf. Sec. 5.3.1) without any modifications, but others have to be slightly "tuned" to take into account the fact that our theory is many-sorted. For instance, the rule of substitution now has the following form:

```
|- mec
inv is free in mec
ter is sort of inv
-------------------
|- mec[inv/ter]
```

The predicates is free and is sort of belong to the meta level, i.e., to **M-theory**. They are not expressible in **Lingua-D** since they concern facts about syntactic elements of this language, i.e., facts about this language.

Note that the assumption inv is free in mec was present also in an abstract formulation of the rule (see Sec. 2.3), but the second assumption — ter is sort of inv — has been added to comply with the many-sortedness of **D-theory**. The fact that we write both these requirements as parts of our rule, rather than as an "external" commentary as in Sec. 2.3, is a matter of notational convention. The critical fact is that we do not write

|- inv is free in mec.

but

inv is free in mec

and the same for the other prerequisite.

At the level of our future ecosystem, both metapredicates will be implemented as procedures operating on (the syntaxes of) mec, inv, and ter. Note that the implementation of ter is sort of inv is possible since the domain of our individual variables has been split into sort-dependent categories (cf. Sec. 4.1).

Note also that in the substitution rule we use variables from the meta level, i.e., mec, inv, and ter, that represent variables running over arbitrary metaconditions, individual variables, and terms of **Lingua-D**, respectively. For instance, from the axiom:

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

we can generate, by substitution, a lemma:

pre ide := vex @ con : ide := vex **post** con (5.3.1-1)

and from this lemma, we can further generate a ground formula (another lemma):

pre x := y+1 @ x > 0 : x := y+1 **post** x > 0

but in this lemma, we can't substitute anything for x or y since in the **D-theory,** they are ground terms rather than free variables. If we want to replace y by z, we have to generate a new lemma from (5.3.1-1).

5.3.3 Not all construction rules are expressible as axioms

Theorems and lemmas formulated in the intuitive theory of program denotations, known as **M-theory**, outlined in Sec. 9 of [6], were expressed in **MetaSoft**. In particular, program-construction rules like, e.g.,

pre ide := vex @ con :
  ide := vex (5.3.3-1)
post con

were expressed in this language, where ide, vex, and con denote arbitrary identifiers, value expressions, and conditions, respectively. Based on this rule, we can prove the soundness of a new one:

pre type-compatible(ide, vex**) and-k** con[ide/vex]:
  ide := vex (5.3.3-2)
post con

where con[ide/vex] denotes con with all free occurrences of ide replaced by vex and **type-compatible(**ide, vex**)** is a condition satisfied if ide and vex are of the same type (see Sec. 5.3.5.1). Both these rules are lemmas proved on the ground of **M-theory**.

Let's assume now that we want to include this lemma in the set of axioms of **D-theory**. A valid formula corresponding to (5.3.3-1) would be the following:

pre ide := vex @ con :
  ide := vex (5.3.3-3)
post con

Using the inference rule of substitution, vex may be replaced by a value-expression-term ide + 1, thus getting a new lemma:

pre ide := ide + 1 @ con :
  ide := ide + 1
post con

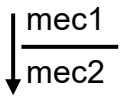
which is, again, a valid formula in **D-theory**.

Consider now (5.3.3-2). This **MetaSoft** lemma can't be "transliterated" into a valid formula in **Lingua-D**, since there is no term in this language that would correspond to con[ide/vex]. In this case, our **MetaSoft** lemma must be introduced into **M-theory** as a non-standard inference rule. Such rules will be discussed in Sec. 5.3.5.

5.3.4 Standard inference rules

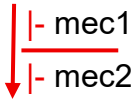
As we have seen in Sec. 5.3.1, every implicative axiom of **D-theory** "induces" an inference rule. This fact may be formally described by the following (meta) theorem:

Theorem 5.3.4-1 *If* mec1 and mec2 are metaconditions in **Lingua-D** and



(i.e., mec1 **implies** mec2)

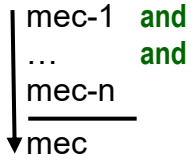
then the following inference rule is sound:



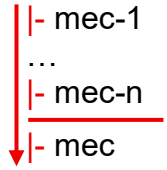
■

Since our program-building rules usually have a conjunction of formulas above the line, the following theorem may be useful as well:

Theorem 5.3.4-2 *If*



is in the repository, then the following inference rule is sound:



■

5.3.5 Nonstandard inference rules

5.3.5.1 Assignment-instruction inference rule

As we have already noted in Sec. 5.2.2, not all program construction rules can be expressed as formulas in **Lingua-D**. In such a case, we shorten the way

program-construction rule → *metacondition* → *inference rule*,

to

program-construction rule → *inference rule.*

Our first example is the derivation of a rule expressed by a metatheorem that ensures the correctness of all metaprograms of the following form:

```
pre type-compatible(ide, vex) and-k con[ide/vex] :
    ide := vex                                          (5.3.5.1-1)
post con
```

To prove that, we start from a lemma that we should have in our repository,

```
pre sin @ con :
    sin
post con
```

from which, by substitution, we derive the next lemma:

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

Now, we want to transform the algorithmic precondition into a non-algorithmic one. To do this, we must move out of the level of **Lingua-D** and continue our reasoning on a **MetaSoft** level. Initially, based on the rule of substitution, we may derive the following **M-theory** rule, which is, in fact, a scheme of a rule.

First Assignment Rule: *For each identifier* ide*, value expression* vex, *and condition* con, *the following met-aprogram is correct:*

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

Here, we have replaced underlined D-variables by not-underlined metavariables that run over concrete ground identifiers, value expressions, and conditions. Note — they are concrete, grounded, but arbitrary.

Now, to eliminate @ from the precondition, we apply the following nonstandard rule:

Second Assignment Rule: *For each identifier* ide*, value expression* vex, *and condition* con, *the following met-acondition is satisfied:*

type-compatible(ide, vex**) and-k** con[ide/vex] ⇨ **(**ide := vex @ con**)**

where con[ide/vex] *denotes condition* con *where all free occurrences of* ide *were replaced by* vex.

We skip a formal definition of con[ide/vex], which must refer to the recursive definition of the syntax of conditions. Note that in this case, ide is a value variable in con.

The denotation of the ad-hoc introduced predicate **type-compatible** is defined as follows:

```
type-compatible(ide, ved).sta =
    is-error.sta                    ➔ error.sta
    ved.sta = ?                     ➔ ?
    ved.sta : Error                 ➔ sta ◄ ved.sta
    let
        ((cle, pre, cov), (obn, dep, st-ota, sft, 'OK'))   = sta
        val                                                = ved.sta
        (tok, (typ, re-ota))                               = obn.ide
    re-ota ≠ $ and re-ota ≠ st-ota ➔ sta ◄ 'reference not visible'
    not ref VRA.cov val             ➔ sta ◄ 'incompatibility of types'
    true                            ➔ (tt, 'boolean')
```

The proof of the ***Second Assignment Rule*** requires a rather laborious argument carried out by induction on the syntactic definition of conditions in **Lingua-V**. Our excuse for skipping this proof is that we have not (yet) fully defined the syntax of conditions. A practical lesson derived from this exercise is that our conditions should be defined in a way that makes our rule sound.

It may be worth noticing in this place that "usually" the satisfaction of con[ide/vex] implies that ide is type-compatible with vex, since otherwise con[ide/vex] would evaluate to an error. However, in some cases, this claim may be unjustified. One such case is where ide does not appear in con and, therefore, con[ide/vex] = con. In that case, con may be satisfied, although ide := vex may generate a typing error. Another case is where ide appears in con, but will never be evaluated, like in:

```
if x > x+1 then ide+1 else x+1 fi > 0
```

These arguments justify the use of metapredicate **type-compatible(**ide, vex**)** in the Second Rule. Using that rule and the rule of strengthening preconditions, we may finally derive the third rule.

Third Assignment Rule: *For each identifier* ide*, value expression* vex, *and condition* con, *the following met-aprogram is correct:*

```
pre type-compatible(ide, vex) and-k con[ide/vex]:
    ide := vex
post con
```

This rule is nonstandard since it is not a D-formula:

- the script "con[ide/vex]" is not a condition,
- the rule is, in fact, a scheme of a rule where ide, con, and vex are metavariables quantified by a general quantifier.

From this metatheorem, we derive the following nonstandard inference rule:

```
|- true
-------------------------------------------
|- type-compatible(ide, vex) and con[ide/vex]:
       ide := vex
   post con
```

In this case, the use of the sign |- below the line does not denote the validity of a formula — note that the script under |- is not a formula — but the fact that once ide, vex, con and con[ide/vex] are replaced by concrete terms (although not necessarily ground), then the formula generated in this way is valid and, therefore, may be stored in the repository.

The purpose of writing |- true above the line is only to make our rule an implicative one.

5.3.5.2 The removal of an assertion

The following nonstandard rule expresses the fact that the removal of an assertion from a correct metaprogram with an error-sensitive postcondition does not violate the correctness of this program (for error sensitivity see Sec. 9.2.1 of [6]):

```
|- pre prc : head ; asr con rsa ; tail post poc
|- error-sensitive(poc)
-----------------------------------------------
|- pre prc : head ; tail post poc
```

Here, we assume that the phrase above the line represents a metaprogram; hence, the phrase below the line also represents a metaprogram.

5.3.5.3 The replacement of a condition in an assertion by a weakly equivalent one

The following diagram may describe this rule:

```
|- pre prc : head ; asr con1 rsa ; tail post poc
|- con1 ⇔ con2
|- error-sensitive(poc)
-----------------------------------------------
|- pre prc : head ; asr con2 rsa ; tail post poc
```

This rule is to be understood similarly to the former.

5.3.5.4 The call of an imperative procedure

```
|- prc-call      ⇨ myProc (val fpa-v ref fpa-r) my-body imperative in MyClass
|- prc-call      ⇨ (pass actual val apa-v ref apa-r to formal val fpa-v ref fpa-r with MyClass) @ prc-body
|- prc-call      ⇨ procedure MyClass.myProc is opened
|- prc-call      ⇨ coe is current
|- prc-body      ⇨ my-body @ poc-body
|- poc-body      ⇨ fpa-r well-valued in coe
|- poc-body[fpa-r/apa-r] ⇨ poc-call
-----------------------------------------------
|- pre prc-call :
      call MyClass.myProc (val apa-v ref apa-r)
   post poc-call
```

Other examples of nonstandard rules may be the following:

1. the replacement of a boolean value-expression in a program by a strongly equivalent expression,
2. the introduction of an assertion block into a program (see Sec. 9.5.1 of [6]),
3. adding a register identifier to a program (see Sec. 9.5.3 of [6]).

This list is certainly not complete.

6 Denotational models of ecosystems

6.1 Primary and secondary ecosystems

In this section, we outline our vision of an ecosystem that can assist programmers developing programs in **Lingua-V** (or a similar language). We do not attempt to provide a comprehensive description of such a system; instead, we aim to outline the primary directions of its development. We illustrate our general investigations with a few examples.

The category of ecosystems that we shall describe below, we shall refer to as *primary ecosystems*. In such ecosystems, programmers are dynamically building metaprograms and their components using a fixed set of construction rules. Precisely speaking, they are given an initial repository of valid metaconditions written in **Lingua-D**, and they build and store new valid metaconditions in the repository. All elements of these repositories will be referred to as *lemmas* and will be given individual names. Although some of them will be initially assumed as *axioms*, from the perspective of programmers, assumed axioms and proved lemmas will be used in the same way, and therefore, we shall call them all *lemmas*.

The only tools used by programmers in deriving new lemmas from the existing ones will be *actions.* Each metacondition-creating action will represent an individual inference rule. Formally, the denotations of actions will be operations that, given the names of some lemmas in the repository, derive a new lemma using a chosen rule. E.g., an action assigned to the rule of detachment will be given three names and will:

1. identify lemmas assigned to the first two names,
2. check if they are of appropriate syntactic structures,
3. detach and store in the repository the target lemma under the third name.

In primary ecosystems, programmers can't create new actions — they can only use the existing ones. Since all actions represent, by definition, sound construction rules, programmers who use primary ecosystems can't develop incorrect programs. Of course, it may also happen that they are unable to create the metaprograms they want.

The impossibility of developing an intended program, or the difficulty in doing so, may also be due to the lack of appropriate rules, i.e., actions. In that case, we temporarily assume that a new action can be added by a system superuser, who takes responsibility for its soundness. Formally, such a superuser modifies the current **Lingua-E** and must prove that the new rule is sound. This approach may be suitable at an early stage of developing our experimental ecosystem.

Once a primary ecosystem is built and verified by a sufficient number of examples, we can begin to consider secondary ecosystems where actions are user-definable and storable, such as procedures. Such ecosystems should be based on a higher-order formalized theory, **D^2-theory**, which is grounded in a language, **Lingua-D^2**, allowing us to formulate and prove the soundness of the inference rules of **D-theory**.

In this paper, we focus on primitive ecosystems. We believe that some "substantial" experiments with primitive systems should precede any research on secondary systems.

6.2 Repositories and actions

In a primary ecosystem viewed as a programming language **Lingua-E**, where repositories play the roles of states, the denotations of instructions, referred to as *actions*, are functions that modify repositories. In Sec. 6, we shall describe only the algebra of denotations **AlgDen-E**, since the derivation of syntax is, in this case, straightforward. We start by formalizing the concept of a repository, and to do that, we introduce three domains:

car : Character = {a, b, c, …, A, B, C,…, 0, 1,…,9, (,), …}
nam : Name = Character^{c+} the names of elements stored in repository
rep : Repository = Name ⟹ (ValMetCon-D | Con-D).

We assume that all metaconditions stored in repositories are valid and may be grounded or free. In repositories, we shall also store conditions to allow for using their acronyms (names) in the developed metaprograms in place of long conditions (examples in Sec. 6.5). An example of a free metacondition to be stored in a repository may be the following:

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

It represents an atomic standard program-construction rule as described in Sec. 9.4.4 of [6]. An example of a corresponding ground metacondition may be

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

6.3 Carriers of the algebra of denotations

We assume that the algebra of denotations **AlgDen-E** will have the following carriers:

```
tex  : Text          = Character^c+
nam  : Name          = Text
svd  : SubVecDen     = IndVal-D ⟹ Text                the denotations of substitution vectors
acd  : ActDen        = Repository ↦ Repository | Error  the denotations of actions
inv  : IndVar-D   =                                    variables of Lingua-D
```

Following our categorization of inference rules (Sec. 5.3.1), we split actions into three categories:

- *basic actions* — actions derivable from basic inference rules,
- *standard actions* — actions derivable from standard inference rules,
- *nonstandard actions* — actions derivable from nonstandard inference rules.

Since we have assumed that the formulas stored in repositories are valid, all reachable actions must ensure this requirement is met.

6.4 Constructors of the algebra of denotations

6.4.1 Auxiliary functions

We shall need two functions:

```
free      : IndVar-D x MetCon-D  ↦ {tt, ff}        variable is free in metacondition
matching  : IndVar-D x Text      ↦ {'OK'} | Error  the sort of a variable matches the sort of text
```

We assume that the first function returns ff also in the case where a variable does not appear in the metacondition.

The second function is more sophisticated. It recognizes the sort of a variable and then performs a parsing procedure of the textual argument to identify its sort. The existence of this function in our model implies that the future implementation of the ecosystem must be equipped with an intelligent editor built on the grammar of **Lingua-D**, i.e., it must include its parser.

6.4.2 Constructors of substitution vectors

Substitution vectors are defined as arbitrary mappings from individual variables to texts, but reachable substitution vectors should comply with the compatibility of the sorts of variables with the sorts of associated texts. First function builds a simple substitution vector:

```
create-sub : IndVar-D x Text ↦ SubVecDen
create-sub(inv, ter) =
```

not matching(inv, tex) ➔ 'matching not satisfied'
true ➔ [inv/tex]

The next function expands a given substitution vector by a new component:

expand-sub : SubVecDen x IndVar-D x Text ⟼ SubVecDen
expand-sub.(sub, inv, tex) =
sub.inv = ! ➔ 'variable already assigned'
not matching(inv, tex) ➔ 'matching not satisfied'
true ➔ sub[inv/ter]

6.4.3 Constructors of basic actions

6.4.3.1 Substitution actions

Let's recall the rule of substitution:

|- mec
inv is free in mec
ter is sort of inv
|- mec[inv/ter]

To define the corresponding action, we shall need some auxiliary concepts. For any metacondition mec, any text tex, and any individual variable inv (inv runs over ide, rex, vex, etc.) by

mec[inv/tex]

we denote the result of the substitution of tex for all free occurrences of inv in mec. If inv is not free in mec, or does not appear in mec, then mec[inv/tex] = mec. The next function does the same, but checks if the substitution is sort-wise legal:

swap : IndVar-D x Text ⟼ MetCon-D ⟼ MetCon-D | Error
swap.(inv, tex).mec =
not free.(inv, mec) ➔ 'variable not free'
not matching(inv, tex) ➔ 'matching not satisfied'
true ➔ mec[inv/tex]

Note that this function returns an error if the result of the swapping does not belong to MetCon-D. This situation will occur if the type of inv is different from the type of ter. The following function swaps several free variables for texts one after another.

replace : SubVecDen ⟼ MetCon-D ⟼ MetCon-D | Error
replace.svd.mec =
let
[inv-1/tex-1,…,inv-n/tex-n] = svd
mec-1 = swap.(inv-1,tex-1).mec
mec-i = for i = 2;n
mec-(i-1) : Error ➔ mec-(i-1)
true ➔ swap.(inv-i/tex-i).mec-(i-1)
true ➔ mec-n

Now we are prepared to define the constructor of the actions of substitution. It takes three arguments:

- nam-s — source-formula name
- svd — substitution-vector denotation,
- nam-t — target-formula name,

and returns a function that modifies repositories:

substitute : Name x SubVecDen x Name ⟼ ActDen i.e.
substitute : Name x SubVecDen x Name ⟼ Repository ⟼ Repository | Error

```
sub-gro.(nam-s, svd, nam-t).rep =                          -s – source,  -t – target
   rep.nam-s = ?   ➔ 'no source metacondition'
   rep.nam-t = !   ➔ 'target name already assigned'
   let
      so-mec = rep.nam-s
      ta-mec  = replace.svd.so-mec
   ta-mec : Error          ➔ ta-mec
   true                    ➔ (rep[nam-t/ta-mec], rdi)
```

This function:

1. checks if the source identifier points to a metacondition,
2. checks if the target identifier is not already assigned,
3. gets the source metacondition,
4. performs the indicated replacement and checks if the result is not an error,
5. modifies the current repository by storing in it the target metacondition under the target identifier.

6.4.3.2 Detachment actions

The detachment rule is the following:

```
|- mec1
|- (mec1 implies mec2)
|- mec2
```

To define a corresponding action, let's start by introducing an auxiliary function:

```
root : MetCon-D ⟼ {'and', 'or', 'implies', 'not', 'nil'}
root.mec =
   mec = and(mec-1, mec-2) ➔ 'and'
   …
```

This function returns the name of the root operator of a compound metacondition (i.e., the top operator of the parsing tree of the metacondition) and 'nil' for an atomic metacondition. The following constructor creates an action that generates a valid formula and stores it in the current repository.

```
detach : Name x Name x Name ⟼ ActDen
detach : Name x Name x Name ⟼ Repository ⟼ Repository | Error
detach.(nam-p, nam-i, nam-t).rep =                  -p – prerequisite, -i – implication, -c – conclusion
   rep.nam-p = ?          ➔ 'no prerequisite metacondition'
   rep.nam-i  = ?         ➔ 'no implication metacondition'
   rep.nam-c = !          ➔ 'conclusion name already assigned'
   let
      mec-p   = rep.nam-p
      mec-i    = rep.nam-i
   root.mec-i ≠ implies   ➔ 'implication expected'
   let
      implies(mec-ps, mec-co) = mec-i                  ps- "premise", co- "conclusion"
   mec-p ≠ mec-ps         ➔ 'prerequisite inadequate'
   true                   ➔ (rep[nam-c/mec-co], rdi)
```

Note that the validity of mec-co follows from the rule of detachment and the facts that mec-p and mec-i are in the repository, i.e., are valid. The sign ≠ denotes the inequality of texts.

6.4.4 Constructors of standard actions

6.4.4.1 Strengthening-precondition action

Based on the rule derived in Sec. 5.3.4, we define the following constructor of actions:

```
strengthen-pre : Name x Name x Name ⟼ Repository ⟼ Repository | Error
strengthen-pre.(nam-m, nam-s, nam-t).rep =          -m – metaprogram, -s – stronger, -t – target
   rep.nam-m    = ?                    ➔ 'no prerequisite metaprogram'
   rep.nam-s    = ?                    ➔ 'no stronger-than metacondition'
   rep.nam-t    = !                    ➔ 'target name already assigned'
   not is-metaprogram.(rep.nam-m)      ➔ 'metaprogram expected'
   root.(rep.nam-s) ≠ ⇨                ➔ 'a stronger-than metacondition expected'
   let
      pre prc : spr post poc    = rep.nam-m
      prc1⇨ con                 = rep.nam-s
   con ≠ prc                      ➔ 'conclusion not adequate'
   let
      new-mec = pre prc1 : spr post poc
   true                           ➔ rep[nam-t/new-mec]
```

In this definition, we have introduced two new parsing-oriented techniques. One is a textual predicate is-metaprogram which checks if a given piece of text is a metaprogram. It belongs to the same category as the function root. The second technique is more sophisticated and is included in the **let**-declaration

```
let
   pre prc : spr post poc = rep.nam-m
   prc1⇨ con              = rep.nam-s.
```

It is understood as a description of the following parsing-based algorithm that creates the following local substitution vector:

```
pre-con          ← prc
spec-prg         ← spr
post-con         ← poc
stronger-pre-con ← prc1
condition        ← con
```

and then uses it in synthesizing the target metaprogram.

Of course, to apply the strengthening-precondition action, we must store earlier in the repository appropriate valid metaconditions of the form con1 ⇨ con2 (note that the D-formula con1 ⇨ con2 is, of course, not valid). Examples of valid stronger-than metaconditions may be the following:

```
(ide is integer)                                          ⇨ (ide < ide+1)
(ide1 is integer) and (ide2 is integer) and (ide1 < ide2) ⇨ (ide1+1 < ide2+1)
(x is integer)                                            ⇨ (x < x+1)
y := x+1 @ (var x is integer) and-k (var y is integer) and-k (y = 4)  ⇨
                                  (var x is integer) and-k (var y is integer) and-k (x=3)
```

6.4.4.2 Adding irrelevant conditions

If we need to add an irrelevant condition to a pre- and post-condition of a program — this may happen when we are adding an underivable condition — we have to use an action derived from the following construction rule:

```
pre prc: spr post poc
con irrelevant-for (pre prc: spr post poc)
-------------------------------------------
pre (prc and-k con) : spr post (poc and-k con)
```

To do that, analogously as in the case described in Sec. 6.4.4.1, we must previously store in the repository appropriate lemmas about **irrelevant-for** metacondition, such as, e.g., (cf. Sec. 2.2):

```
different(ide1, ide2) implies ((ide1 is free) irrelevant-for (let ide2 be tex tel)).   (6.4.4.2-1)
```

The denotation of the ad-hock introduced metacondition **different**(ide1, ide2) is the following:

```
different.(ide1, ide2).vlu =
```

vlu.ide1 ≠ vlu.ide2 ➔ true
true ➔ false

where vlu is a valuation (see Sec. 2.3) and relation ≠ compares two strings of characters.

Let's see how to apply this rule by executing a corresponding action to add a condition **(var** x **is** integer**)** to the metaprogram

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

To do that, we identify lemma (6.4.4.2-1) in the repository, and we apply to it a substitution action with the following vector:

ide1 ← x
ide2 ← y
tex ← integer

thus getting

different(x, y) **implies** ((x **is free**) **irrelevant for** (**let** y **be** integer))

At this moment, we, of course, would like to use detachment, but can we expect that **different**(x, y) is in the repository? To achieve that, we should have derived it from some axioms about the inequality of character strings. In this case, we might use our theorem prover; however, a more straightforward solution may be to employ a procedure that compares two strings of characters. Note that we may proceed analogously when we want to prove the validity of, say, 8 < 10 (cf. Sec. 2.3). Such shortenings of a formalized route would not be acceptable if we were building a "self-standing" theorem prover. Still, in our situation, we may accept a hybrid solution, as all we need is a vehicle for justifying the formulas.

The expected constructor of actions is the following:

add-irrelevant : Name x Name x Name ⟼ Repository ⟼ Repository | Error
add-irrelevant.(nam-m, nam-i, nam-t).rep = -m – metaprogram, -i – irrelevant, -t – target
  rep.nam-m = ? ➔ 'no prerequisite metaprogram'
  rep.nam-i = ? ➔ 'no irrelevant-for metacondition'
  rep.nam-t = ! ➔ 'target name already assigned'
  not is-metaprogram.(rep.nam-m) ➔ 'metaprogram expected'
  root.(rep.nam-s) ≠ **irrelevant-for** ➔ 'an irrelevant-for metacondition expected'
  let
    pre prc **:** spr **post** poc = rep.nam-m
    con **irrelevant-for** (**pre** prc**:** spr **post** poc) = rep.nam-i
    new-mec = **pre** (prc **and-k** con**) :** spr **post** (poc **and-k** con)
  true ➔ rep[nam-t/new-mec]

This definition is similar to the one of strengthen-pre in Sec. 6.4.4.1. It involves parsing and pattern-matching techniques.

6.4.5 Constructors of nonstandard actions

6.4.5.1 Assignment-creation action

In this case, we implement the program-construction rule developed in Sec. 5.3.5.1. The corresponding constructor is the following:

assign : Identifier x Text x Name x Name ⟼ Repository ⟼ Repository | Error
assign.(ide, tex, nam-c, nam-t).rep = -c – condition, -t – target
  rep.nam-t = ? ➔ 'target name already assigned'
  not is-value-expression.tex ➔ 'value expression expected'
  let

```
con          = rep.nam-c
new-con      = con[ide/tex]                                          see Sec. 5.3.3.1
asi-program  = pre con[ide/vex] and-k type-compatible(ide, vex) : ide := vex post con
true                          ➔ rep[nam-t/asi-program]
```

In this case, one of the arguments is a text that is supposed to be a value expression. Since this argument is written by a programmer "from the keyboard", the action is equipped with a parsing engine is-value-expression that checks the syntactic correctness of this argument.

6.4.5.2 Proving action

Implementationally, this "singular" action activates a theorem prover which attempts to prove the validity of a given metacondition. We include it in the category of nonstandard actions, although it is not associated with any single inference rule, but, in a sense, with all of them. It will be used to prove the validity of these formulas, which can't be derived — or which a programmer is unable to derive — from the formulas stored in the repository.

To define the corresponding constructor, we assume that we have in our model a partial function that represents a theorem prover:

valid : MetCon-D → {YES, NO]

This function is partial, as the validity of formulas in our theory is undecidable; that is, no algorithm can determine whether a given formula is valid or not[11]. The constructor of the corresponding action is the following:

```
prove : Text x Name ⟼ Repository → Repository | Error
prove.(tex, nam).rep =
   not is-metacondition.tex   ➔ "the argument is not a metacondition'
   valid.mec = ?              ➔ ?
   valid.mec = 'NO'           ➔ 'metacondition not valid'
   true                       ➔ rep[nam/mec]
```

This action first checks if the argument text is a metacondition and, if so, attempts to prove its validity.

6.5 An example of a program's derivation — bubble sort

Bubble sort is a well-known program that sorts an array "in situ", i.e., without using additional memory resources. It uses two pointers that are moving along the array being sorted. Initially, both pointers are in their starting positions, where i = j = 0.

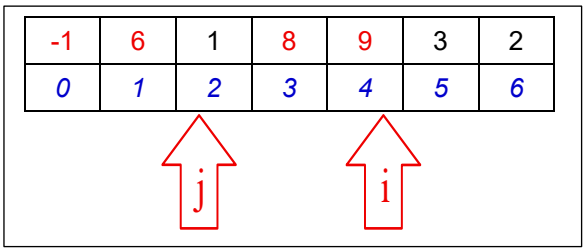


Fig. 6.5-1 Bubble sort

Next, in every iteration of an outer loop, pointer i is incremented by 1, and pointer j is moved to the position of i. At this moment, the array is sorted from 0 to i, possibly except the j's element (the bubble). We say that the array is *sorted from* 0 *to* i, *but* j, and this property is an invariant of an inner loop where pointer j, together with the assigned element, is moved step-by-step to the left. This happens as long as j's element is smaller than its (j-1)'s neighbor. Once that is not the case, our program stops moving j since the array is already sorted from 0 to i. In Fig. 6.5-1, our array is sorted from 0 to 4, *but 2*.

[11] Kurt Gödel proved in 1931 that all theories that "include" arithmetic are undecidable.

The program that we intend to develop is the following:

```
pre (constant source is array of integer) and-k (len(source) > 0) :
    let n, i, j be integer tel ;
    let arr be array of integer tel ;
    read source into arr daer ; n := len(arr) ; i := 0 ; j := 0;
    while i < n                                                    # sorting from 0 to i+1
        do
            asr (arr sorted from 0 to i) and-k permutation(arr, source) and-k (0 ≤ j ≤ i ≤ n) ;
            i := i+1;
            j := i ;
            while arr[j-1] > arr[j]
                do
                    asr (arr sorted from 0 to i but j) and-k permutation(arr, source) and-k (0 ≤ j ≤ i ≤ n) rsa ;
                    swap(arr, j - 1, j) ;
                    j := j - 1 ;
                    asr (arr sorted from 0 to i but j) and-k permutation(arr, source) and-k (0 ≤ j ≤ i ≤ n) rsa ;
                od ;
                asr sorted(arr, 0, i) and-k permutation(arr, source) and-k (0 ≤ j ≤ i ≤ n) rsa ;
        od ;
        asr (arr sorted from 0 to i) and-k permutation(arr, source) and-k (i=n)
post sorted(arr, 0, n) and-k permutation(arr, source)
```

To construct this program, we assume that **Lingua-V** has been enriched by the concept of a *constant* and associated with it a *constant's predicate* of the form:

```
constant ide is tex
```

where tex is a type expression. This predicate is satisfied in a state if ide has been (somehow) marked as a constant of type indicated by tex in this state. We assume further that the only way a constant may be used in the program component of a metaprogram is in a *reading instruction* of the form:

```
read ide1 into ide2 daer
```

where ide1 is a constant and ide2 is a declared variable[12]. For this instruction to be executed cleanly in a state, ide1 must be a constant of the type of ide2. We assume further to have an array-oriented instruction:

```
swap(arr, j)
```

that swaps two adjacent elements arr[j-1] with arr[j] in array, and expression

```
len(ide)
```

that returns the length of the value of ide, provided that it is an array. We also assume to be given three array-oriented predicates (for simplicity, we define them using variables appearing in our program):

```
permutation(arr, source)      — array arr is a permutation of array source,
sorted(arr, j, i)             ≡ (def)  0 ≤ j < i ≤ len(arr)  and-k (∀k : j ≤ k < i) (arr[k] ≤ arr[k+1])
arr sorted from 0 to i but j  ≡ (def)  if j = 0 then sorted(arr, 0, i) else sorted(arr, 0, j-1) and-k sorted(arr, j+1, i) fi
```

Below, we present a two-column table that documents the development of our program. The elements of the domain Name are typed in *Times New Roman* italics. We use these names not only to name the repository's items, but also when "calling" them in other items.

In our example, we concentrate more on the "logistics" of program development than on its logic. Therefore, we justify some steps solely by intuitive arguments. To save space, we omit vertical arrows in the rules, as they are all unidirectional in our case.

---

[12] To formally build the mechanism of constants into **Lingua-V** we have to redefine the denotations of assignments, expressions, procedure calls and conditions. Since it is fairly clear, how to do it, we skip this issue.

COMMENTS	REPOSITORY
We introduce a named condition into the repository. In this case, we do not need to prove or derive anything; instead, we assume that our editor will verify the syntactic correctness of the introduced condition. We also assume that the condition **var** arr **is** array is implicit in the condition permutation(arr, source).	***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)
We identify the *while-axiom* in the repository.	***while-axiom ::*** **pre (**inv **and-k** vex**) :** sin **post** inv **and** inv **insures LR of asr** vex **rsa ;** sin **and** prc ⇨ inv **and** inv ⇨ (vex **or-k (not-k** vex)) **and** inv **and-k (not-k** vex)) ⇨ poc --- **pre** prc **: asr** inv **rsa ; while** vex **do** sin **od post** poc
By an action of substitution, we generate the *while-lemma1.* In the following steps, we have to derive all five metaconditions above the line of the *while-lemma.* We skip easy, but laborious, details, noticing only that *invariant* implies j ≥ 0 that, in turn, ensures the limited replicability (LR) of the body of the loop (see Sec. 9.3.4 of [6]).	***while-lemma1 ::*** **pre (***invariant* **and-k** arr[j-1] > arr[j] **) :** swap(arr, j - 1, j) **;** j:= j-1 **post** *invariant* **and** *invariant* **insures LR of asr** arr[j-1] > arr[j] **rsa ;** swap(arr, j - 1, j) **;** j:= j-1 **and** *invariant* ⇨ *invariant* **and** *invariant* ⇨ (arr[j-1] > arr[j] **or-k (not-k** arr[j-1] > arr[j])) **and** *invariant* **and-k (not-k** arr[j-1] > arr[j])) ⇨ *invariant* **and-k** sorted(arr, 0, i) --- **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)
To derive the *inner-loop* from the *while-lemma,* we apply two actions: • detachment, • the omission of an assertion in a correct metaprogram.	***inner-loop ::*** **pre** *invariant* **:** **while** arr[j-1] > arr[j] **do** swap(arr, j - 1, j) **;** j:= j-1 **od** **post** *invariant* **and-**k sorted(arr, 0, i)
Now, we proceed to the creation of the outer loop. The general lemma we need now is *while-lemma 2.0*. We have to generate it or find it in the repository. Here body **neutral for** ide1 means that the execution of the body does not alter the value of ide1.	***while-lemma2.0 ::*** **pre** inc **and-k (**ide1 < ide2**) :** ide1 **:=** ide1+1 **;** body **post** inc **and** inc ⇨ ide1 < ide2 **or-k** ide1 ≥ ide2 **and** body **neutral for** ide1 --- **pre** inc **end-k (**ide1 < ide2**) :** **while** ide1 < ide2 **do** ide1 := ide1 + 1 **;** body **od** **post** inc **end-k (**ide1 = ide2**)**

From this lemma, by appropriate substitutions (body is replaced by *inner-loop*), we get *while-lemma2.1*

while-lemma2.1 ::
pre *invariant* **and-k** (i < n) :
    i := i + 1 ; *inner-loop*
post *invariant* **and-k** sorted(arr, 0, i) **and**
invariant ⇨ i > n **or-k** i ≤ n **and**
inner-body **neutral for** i

---

pre *invariant* **end-k** (i < n) :
    while i < n **do** i:= i + 1 ; *inner-loop* **od**
post *invariant* **end-k** (i= n)

From *while-lemma2.1,* we obtain the result by detachment of the outer loop.

outer-loop ::
pre *invariant* **end-k** (i < n) :
    while i < n **do** i:= i + 1 ; *inner-loop* **od**
post *invariant* **end-k** (i = n)

In the last but one step, we generate the preamble program.

preamble ::
pre (constant source **is** array **of** integer**) and-k (**len(source) > 0**) :**
    let n, i, j **be** integer **tel ;**
    let arr **be array of** integer **tel ;**
    read source **into** arr **daer** ;
    n := len(arr) ;
    i := 0 ;
    j := 0
post *invariant* **end-k** (i < n)

In the last step, we sequentially combine the preamble with the outer loop, getting in this way our target program in a "compact form", i.e., with metanames. We can use it in this form in further work, or "unfold" the names by a substitution action if we want to run our program immediately.

6.6 A hybrid scenario of the development of prime repositories

As we already mentioned in Sec. 2.3, we shall not attempt to make our repository (the set of axioms) logically complete. On the other hand, we must ensure that it is consistent and, at the same time, "sufficiently complete" to make sufficiently many lemmas provable. We propose the following ad hoc scenario to achieve this goal:

1. We establish two folders in the repository of the ecosystem: one for lemmas (valid formulas) and conditions, and another one for inference rules. In the case of standard inference rules, both, the rule and the corresponding lemma, are stored in the corresponding folders.
2. We initialize the folder of lemmas with some commonly known mathematical axioms and lemmas that we can derive from them. This initial repository should include all (currently) known to us lemmas expressible in **Lingua-D**.
3. We initialize the folder of inference rules with basic inference rules plus these standard and nonstandard inference rules, the soundness of which we have proved.
4. While working with the ecosystem, we add to it new lemmas and new inference rules under the condition that we prove their validity or soundness, respectively, either within our formalized **D-theory** or within the metatheory **M-theory**.

7 A comparison of Lingua-V with Dafny

Contrary to **Lingua-V**, which is today only a sketch of a future language, the **Dafny** project is based on an implemented programming language. The syntax of this language is formally defined by a BNF-like grammar, but its semantic is described only informally and mainly by examples. Along with this language, **Dafny** offers an ecosystem within Visual Studio Code, as well as a system for proving program correctness. The latter is based on Hoare-like proof rules that are tacitly assumed to be adequate for the languages. In other words, Dafny is assumed to be implemented in a way that guarantees the soundness of these rules, rather than being proved to be so. The process of proving program correctness is partially automated by the theorem prover Z3 (see [1]).

In our opinion, the significant difference between the Lingua project and Dafny is such that our construction rules are proven sound on the ground of a denotational model of **Lingua-V**, rather than being assumed to be so. More technical differences are summarized in the table below.

CONCEPT	LINGUA	DAFNY
denotations	The development of a many-sorted algebra of denotations is the "founding step" in designing **Lingua**.	The concept of denotations is neither used nor even mentioned.
syntax	Syntax is derived from an earlier constructed algebra of denotations in three steps: the derivation of abstract syntax, of concrete syntax, and of colloquial syntax. The first two of them are many-sorted algebras. All syntaxes are described by equational grammars.	Syntax is defined by BNF equations and is "final", i.e., it corresponds to our colloquial syntax. The definition of syntax is essentially the "founding step" of the language.
semantics	Abstract and concrete syntaxes are defined in a way that guarantees the existence of (unique) homomorphisms into the algebra of denotations. These homomorphisms are the denotational semantics of **Lingua**. The semantics of the (final) colloquial syntax is a composition of a recovery function that turns colloquial syntax into concrete syntax and the semantics of concrete syntax.	In the source report [11], only the syntax is formally defined. Semantics is described informally and may be guessed to be implicit in Hoare-like proof rules. Although the authors never explicitly state this, they seem to regard these rules as evident, thereby tacitly assuming that the implementation (semantics) of **Dafny** ensures their soundness.
values	Typed data or objects.	No such concepts are explicitly defined.
abstract errors	All domains of values include abstract errors, and all constructors "react" to them. Besides, states may carry errors, and therefore, the denotations of program components also react to errors. The mechanism for handling errors is formalized in semantics.	We have not identified any comments about errors.
types	Finitistic structured elements that unambiguously identify sets of values called the *clans of types*.	Informally understood as sets of values plus corresponding constructors. There are several categories of built-in types, as well as mechanisms for creating user-defined types. The description of types mixes the syntax with an (informal) semantics of type- and value declarations.
value expressions	The denotations of value expressions are partial state-to-value functions, and may	Value expressions are called right-hand-side expressions and constitute a very rich syntactic category. They are used both in programs and in their

	return errors as values. Expressions do not generate side effects.	specifications. They may have side effects, e.g., if they include method calls.
reference expressions	Their denotations are total functions from states to references or errors.	They are called left-hand-side expressions and are defined in Sec. 9.14 of [11]. An example is a[i], where a is an array. It might be interesting to analyze them in the context of the de Bakker paradox (Sec. 9.4.6.6 of [10]).
boolean expressions (BE)	Their denotations are partial 3-valued predicates based on McCarthy's calculus.	No formal definition, but it is pointed out that the evaluation of BE may generate an error in a way that corresponds to McCarthy's calculus.
type expressions	Their denotations are total functions from states to types or errors.	Not explicitly defined.
conditions	Syntactically constitute a superset of boolean expressions, but semantically are based on Kleene's rather than McCarthy's calculus.	One may guess that conditions are just expressions with boolean values, although not all such conditions may be used in programs. This issue is technically complicated, and we did not have the patience to thoroughly study it.
constants versus ghost items	Constants have been introduced to describe the relationship between the initial values of variables and their current values. They may be used to initialize variables, but beside that, only in non-algorithmic conditions.	Ghost items are not visible to a compiler but are detected by a theorem prover. Ghost items are used entirely in specifications.
constants	Constants must be formally introduced into **Lingua**. In our example, we only announced this decision.	Introduced as a particular case of variables called "constant variables". Declarable.
metaprograms	**pre** pr-con specified instruction **post** po-con Specified instructions may include assertions.	Called just "programs", but including the same elements, although in a different arrangement: **requires** pr-con **assures** po-con specified instruction
program validation	Correct metaprograms are built in a stepwise way by sound construction rules. Usually, an application of a rule requires proving an implication concerning program items (variables, types, methods, etc.). During the development of programs, preconditions, postconditions and specinstructions are constructed and modified.	Developed specified programs are proved correct by a theorem prover. Programs may be built from other programs (lemmas), but — as far as we understood — when "a single" program is developed, it is first developed (written) as a whole, and then proved correct.

8 References

[1] Bjørner Nikolaj, Moura (de) Leonardo, Nachmanson Lev, Wintersteiger Christoph, *Programming Z3*, Microsoft Research

[2] Blikle Andrzej, *Toward Mathematical Structured Programming*, Formal Description of Programming Concepts (Proc. IFIP Working Conf. St. Andrews, N.B. Canada 1977, E.J. Neuhold ed. pp. 183-2012, North Holland, Amsterdam 1978

[3] Blikle Andrzej, *On Correct Program Development*, Proc. 4th Int. Conf. on Software Engineering, 1979 pp. 164-173

[4] Blikle Andrzej, *On the Development of Correct Specified Programs*, IEEE Transactions on Software Engineering, SE-7 1981, pp. 519-527

[5] Blikle Andrzej, *The Clean Termination of Iterative Programs*, Acta Informatica, 16, 1981, pp. 199-217.

[6] Blikle Andrzej, Chrząstowski-Wachtel Piotr, Jabłonowski Janusz, and Tarlecki Andrzej, *A Denotational Engineering of Programming Languages*, a book in progress, 2024, https://moznainaczej.com.pl/what-has-been-done/the-book

[7] Dijkstra Edsger, W., *A constructive approach to the problem of program correctness*, BIT 8 (1968)

[8] Dijkstra Edsger, W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976

[9] Leino K. Rustan M., *This is Boogie 2*, Manuscript KRML 178, working draft 24 June 2008, https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/publications/

[10] Leino K. Rustan M., *Program Proofs*, MIT Press 2023,

[11] Leino K. Rustan M., Cok David R., and the Dafny contributors, *Dafny Reference Manual*, August 29, 2024, http://dafny.org/dafny/DafnyRef/DafnyRef ,

[12] Getting Started with Dafny: A Guide, https://dafny.org/latest/OnlineTutorial/guide

[13] Mostowski, A., *Logika matematyczna*, Monografie Matematyczne 1948

[14] Rasiowa H., Sikorski R., *The mathematics of metamathematics,* Państwowe Wydawnictwo Naukowe, Warsaw 1963

[15] Sierpiński Wacław, *Arytmetyka teoretyczna*, Państwowe Wydawnictwo Naukowe, Warszawa 1955