# A Denotational Engineering of Programming Languages

…

## Part 12: Lingua-OO Object-oriented programming

(Not yet in the book)

Andrzej Jacek Blikle

April 27th, 2020

Work in progres with Piotr Chrząstowski-Wachtel and Janusz Jabłonowski

# Classes and objects intuitively

A class is a tuple of six components:

1. directories – catalogs of identifiers assigned to abstract members of a class; play an auxiliary role, (katalogi)
2. type environment (T) where identifiers are bound to:
    a. *abstract types* (Θ)
    b. *concrete types* as in Lingua
3. method environment (M) where identifiers are bound to:
    a. *abstract methods* or *signatures* which "announce" future procedures by their lists of formal parameters,
    b. *concrete methods,* which are just procedures as in Lingua,
4. class environment (C) where identifiers are bound to:
    a. *abstract classes* (NN) — some of their members are abstract,
    b. *concrete classes* (NN) — all their members are concrete,
5. valuation where identifiers (called attributes) are bound to:
    a. *abstract values* (pseudovalues (Ω, (bod, yok)))
    b. *concrete values* (proper values (dat, (bod, yok)); dat ≠ Ω)
6. error register as in Lingua-2

Types, methods, classes and values are called the members of a class.
A concrete class is called an object.

# Classes formally

| | | | |
|---|---|---|---|
| cla | : Class | = Env x Store | |
| env | : Env | = Dir x TypEnv x MetEnv x ClaEnv | environment |
| dir | : Dir | = {[ ]} \| Identifier ⟹ Dir | directory (katalog) |
| tye | : TypEnv | = Identifier ⟹ Type \| {Θ} | type environment |
| mee | : MetEnv | = Identifier ⟹ Method | method environment |
| cle | : ClaEnv | = Identifier ⟹ Class | class environment |
| sto | : Store | = Valuation x (Error \| {'OK'}) | store |
| vat | : Valuation | = Identifier ⟹ Value | valuation |

where:

identifiers in valuations are called attributes

| | | | |
|---|---|---|---|
| met | : Method | = Procedure \| ProSig | method |
| pro | : Procedure | = ImpPro \| FunPro \| TypPro | procedure |
| prs | : ProSig | = FunProSig \| ImpProSig \| TypProSig | procedure signature |
| fps | : FunProSig | = FoPaDe x TypExpDen | functional-procedure signature |
| ips | : ImpProSig | = FoPaDe x FoPaDe | imperative-procedure signature |
| tps | : TypProSig | = Identifier$^{c*}$ | typological-procedure signature |
| fpa | : FoPaDe | = (Identifier x TypExpDen)$^{c*}$ | formal parameters |

class = ((dir, tye, mee, cle), (vat, err))

In our model classes play the role of states!

# Classes and their directories

**directory**
weight → [ ]
volume → [ ]
Shapes → "dir. of Shapes"
p → [ ]
**type environment**
weight → Θ
sizes → (('L', ('A', number)), TT)
**method environment**
volume → (**val** a, b, c, **ref** z)
area → (**val** a, b, **ref** y) {y:=a*b}
**class environment**
Shapes → "abstract class" }
Triangles → "concrete class"
**valuation**
a → (Ω, (('number'), TT))
b → (12, (('number'), value≤100))

Surface level (level 0) of a class

abstract members in red
concrete members in black

a well-formed class

Directories are redundancies

Directory lists all and only abstract members.

inner classes (has-a)
of surface level
classes of level 1

def: concrete class ≡ all elements concrete
def: object = concrete class

**System Assumption** The well formedness of classes is an invariant of every transformation (mutation) of classes reachable in our algebra.

# Carriers of AlgDen-OO

## Applicative layer

| | | | |
|---|---|---|---|
| ide | : Identifier | = … somehow defined | identifiers |
| ded | : DatExpDen | = Class → Value \| Error | data-expression denotations |
| ted | : TypExpDen | = Class ↦ Type \| Error | type-expression denotations |
| tra | : TraExpDen | = Transfer | transfer-expression denotations |
| yok | : YokExpDen | = Yoke | yoke-expression denotations |

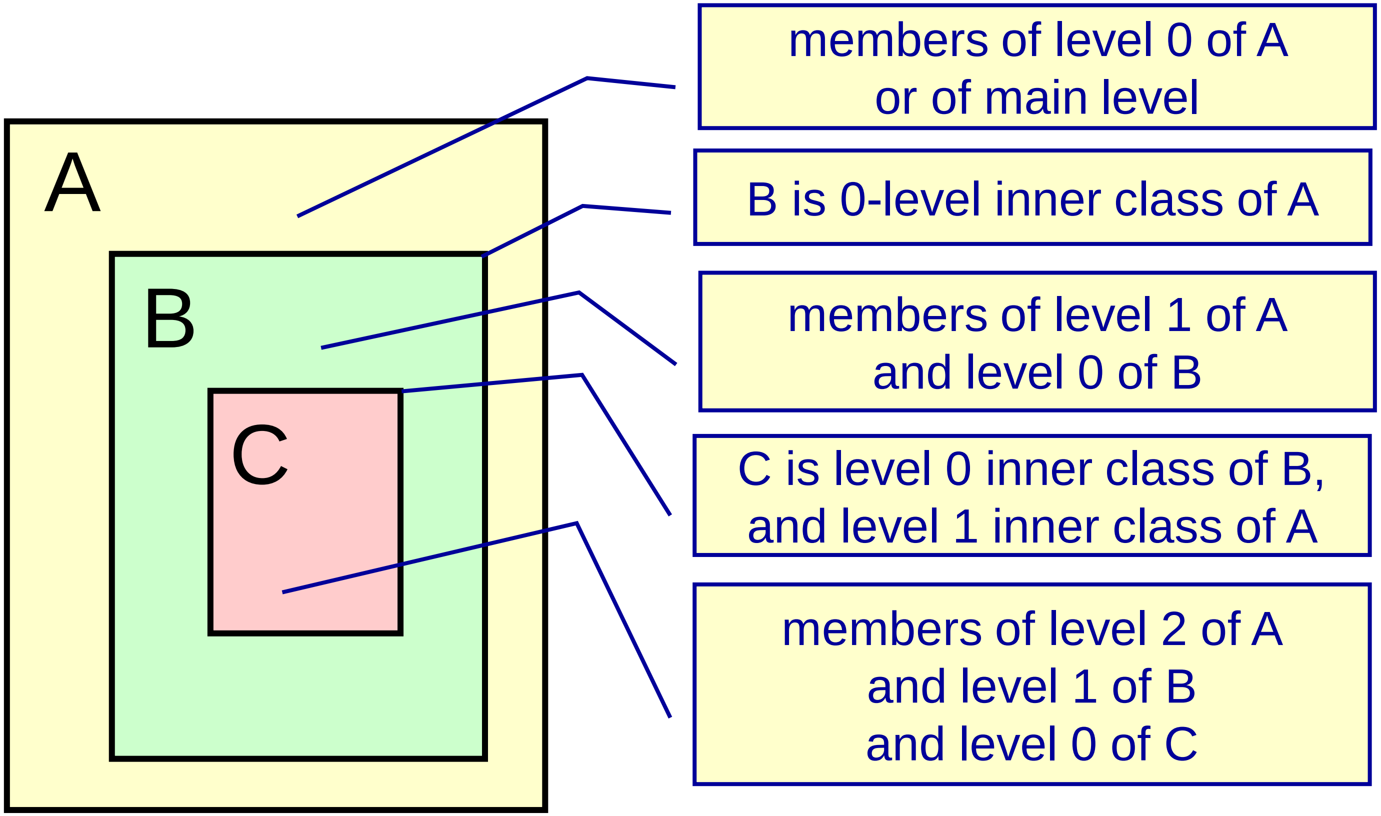
## Imperative layer

| | | | |
|---|---|---|---|
| ded | : DecDen | = Class ↦ Class | declarations (add new elements) |
| ind | : InsDen | = Class → Class | instruction (change values of attributes) |
| prd | : ProDen | = Class → Class | program denotations |
| fpa | : FoPaDe | = (Identifier x TypExpDen)$^{c*}$ | formal parameter denotations |
| apd | : AcPaDe | = Identifier$^{c*}$ | actual parameter denotations |
| tpd | : TypParDen | = Identifier$^{c*}$ | type parameters (formal and actual) |
| prs | : ProSig | = FunProSig \| ImpProSig \| TypProSig | procedure signatures |
| fps | : FunProSig | = FoPaDe x TypExpDen | |
| ips | : ImpProSig | = FoPaDe x FoPaDe | |
| tps | : TypProSig | = TypParDen | |
| pth | : Path | = {('empty')} \| Identifier$^{c*}$ | paths |

# Levels of a class

A = ((dir, tye, pre, cle), (vat, err))
0-level of A = members bound to identifiers in tye, pre, cle, vat
1-level of A = members bound to identifiers in inner classes of 0-level,
etc. for n > 1

**Note**. Members of level n do not belong to level n-1!



members of level 0 of A
or of main level

B is 0-level inner class of A

members of level 1 of A
and level 0 of B

C is level 0 inner class of B,
and level 1 inner class of A

members of level 2 of A
and level 1 of B
and level 0 of C

surface
level

deep
levels

# Four categories of denotations

| **Lingua-OO** | **Applicative** | **Imperative** |
|---|---|---|
| **Surface** | DatExpDen<br>TypExpDen<br>TraExpDen<br>YokExpDen | DecDen<br>InsDen<br>ProDen |
| **Deep** | select + apply | select + apply + replace |

Lingua-2

Lingua-2 + abstract methods + classes

<u>Surface constructors</u> = constructors of surface denotations

<u>Deep constructors</u> = constructors of deep denotations

Here "constructors" are understood in an algebraic sense.

# Surface Denotations and Their Constructors in AlgDen-OO

# Categories of surface constructors

Basic assumptions:

A. all attributes are private in their (directly) hosting classes,
B. all other members are public, although with restricted visibility,
C. instructions modify only the values of attributes (of an arbitrary level),
D. a program consists of a declaration followed by an instruction.

} recommended by
J.Jabłonowski.
J.Sroka

Three categories of surface constructors:

1. old constructors in a new shape; the are said to be inherited from **Lingua-2**,
2. old constructors (slightly) modified: assignment and variable declaration,
3. (genuinely) new constructors.

Old constructors in a new shape create old denotations in a new shape:

((dir, tye, mee, cle), (vat, err)) → ((dir, tye-1, mee-1, cle), (vat-1, err-1))

old denotation in a new shape

Old surface constructors in a new shape:

Constructors of data-expression denotations.
Constructors of type-expression denotations.
Constructors of declaration denotations except variable declaration.
Constructors of instruction denotations except assignment.
Constructors of program denotations.

# Surface variable declaration

```
declare-dat-var: Identifier x TypExpDen  ⟼ DecDen     i.e.
declare-dat-var: Identifier x TypExpDen  ⟼ Class ⟼ Class
declare-dat-var.(ide, ted).cla =
  is-error.cla          ➔ cla
  declared.ide.cla      ➔ ‘identifier-declared’
  let
    typ                              = ted.cla
    ((dir, tye, mee, cle), (vat, ‘OK’))  = cla
  typ : Error           ➔ cla ◄ typ
  true                  ➔ ((dir[ide/[ ]], tye, mee, cle), (vat[ide/(Ω, typ)], ‘OK’))
```

A variable declaration always yields an abstract class.

# Surface assignment

```
assign : Identifier x DatExpDen ↦ InsDen      i.e.
assign : Identifier x DatExpDen ↦ Class → Class
assign.(ide, ded).cla =
  is-error.cla                ➔ cla
  let
    ((dir, tye, pre, cle), (vat, ‘OK’)) = cla
  vat.ide    = ?              ➔ cla ◄ ‘identifier-not-declared’
  ded.cla    = ?              ➔ ?                    an infinite execution
  ded.cla : Error             ➔ cla ◄ ded.cla
  let
    (dat-f, (bod-f, yok-f))    = vat.ide                            f – former
    (dat-n, (bod-n, yok-n))    = ded.cla                            n – new
    com                        = yok-f.(dat-n, bod-n)
  com : Error                 ➔ cla ◄ com
  bod-n ≠ bod-f               ➔ cla ◄ ‘inconsistent-bodies’
  com ≠ (tt, (‘Boolean’))     ➔ cla ◄ ‘yoke-not-satisfied’
  let
    val-n = ((dat-n, bod-n), yok-f)
  dir.ide = [ ]               ➔ ((dir[ide/?], tye, pre, cle), (vat[ide/val-n], ‘OK’))
  true                        ➔ ((dir, tye, pre, cle), (vat[ide/val-n], ‘OK’))
```

ide is abstract

# Surface constructors for type- and method declarations

1. for declarations of abstract types and methods,
2. for concretizations of abstract types and methods,
3. for declarations of concrete types and methods (inherited from **Lingua-2**).

# Types – decl. and concretizations of abstract t.

```
declare-abs-typ      : Identifier                ⟼ DecDen
concretize-abs-typ   : Identifier x TypExpDen    ⟼ DecDen
```

```
declare-abs-typ.ide.cla =
  is-error.cla           ➔ cla
  ide : declared.cla     ➔ 'identifier-declared'
  let
    ((dir, tye, mee, cle), (vat, err)) = cla
  true                   ➔ ((dir[ide/[ ]], tye[ide/Θ], mee, cle), (vat, err))
```

a pseudotype

```
concretize-abs-typ.(ide, ted)).cla =
  is-error.cla    ➔ cla
  let
    ((dir, tye, mee, cle), (vat, err)) = cla
  tye.ide = ?     ➔ 'abstract-type-unknown'
  tye.ide ≠ Θ     ➔ 'an-abstract-type-expected'
  let
    typ = ted.cla
  typ : Error     ➔ cla ◄ error.cla
  true            ➔ ((dir[ide/?], tye[ide/typ], mee, cle), (vat, err))
```

only abstract types may be concretized

# Methods – declarations of abstract m.

```
declare-imp-sig : Identifier x ImpProSig ⟼ DecDen      i.e.
declare-imp-sig : Identifier x ImpProSig ⟼ Class ⟼ Class
declare-imp-sig.(ide, ips).cla =
  is-error.cla              ➔ cla
  declared.ide.cla          ➔ 'identifier-declared'
  let
    ((dir, tye, mee, cle), (vat, 'OK')) = cla
  not no-repetition.ips     ➔ cla ◄ 'repetitions-of-formal-parameters'
  true                      ➔ ((dir[ide/[ ]], tye, mee[ide/ips], cle), (vat, 'OK'))
```

```
declare-fun-sig : Identifier x FunProSig ⟼ Class ⟼ Class
declare-fun-sig.(ide, fps).cla =
  is-error.cla              ➔ cla
  declared.ide.cla          ➔ 'identifier-declared'
  let
    ((dir, tye, mee, cle), (vat, 'OK')) = cla
  not no-repetition.fps     ➔ cla ◄ 'repetitions-of-formal-parameters'
  true                      ➔ ((dir[ide/[ ]], tye, mee[ide/fps], cle), (vat, 'OK'))
```

analogous to imperative

# Concretizations of abstract imperative methods

concretize-imp-sig : Identifier x FoPaDe x FoPaDe x ProDen ⟼ DecDen i.e.
concretize-imp-sig : Identifier x FoPaDe x FoPaDe x ProDen ⟼ Class ⟼ Class

The case of functional methods is analogous

```
concretize-imp-sig.(ide, fpa-v, fpa-r, prd).cla =
  is-error.cla                    ➔ cla
  not ide : declared.cla          ➔ 'identifier-unknown'
  mee.ide = ?                     ➔ 'no-such-method'
  dir.ide = ?                     ➔ 'method-already-concrete'
  let
    message                            = no-repetitions.(fpd ₵ fpd-r)
    ((dir, tye, mee, cle), (vat, 'OK')) = cla
    (fpa-vs, fpa-rs)                   = mee.ide                  - a signature of ide
  message ≠ 'OK'                  ➔ cla ◄ message
  fpa-vs ≠ fpa-v                  ➔ cla ◄ 'value-parameters-not-compatible'
  fpa-rs ≠ fpa-r                  ➔ cla ◄ 'reference-parameters-not-compatible'
  let
    P = create-imp-pro.((fpa-v, fpa-r, prd), (dir, tye, mee[ide/P], cle))   fixed-point def.
  true                            ➔ ((dir[ide/?], tye, mee[ide/P], cle), (vat, 'OK'))
```

**Note**: Procedures may create their private temporary classes.

# Declarations of classes

by a path

1. selecting an inner class of an input class, which in particular may be also the input class itself or an empty class,
2. modifying that class by a given OO-program,
3. binding the new class to a given identifier in the input class.

# Surface declarations of classes

dec-class : Identifier x Path x ProDen ⟼ DecDen      i.e.

dec-class : Identifier x Path x ProDen ⟼ Class ⟼ Class

```
dec-class.(ide, pat, prd).ho-cla =
  is-error.ho-cla            ➔ ho-cla
  ide : declared.ho-cla      ➔ ho-cla ◄ 'class-name-declared'
  let
    in-cla   = select-cla.pat.ho-cla          the parent inner class
  is-error.in-cla            ➔ in-cla         in that case in-cla = ho-cla
  let
    dec-cla  = prd.in-cla                     the declared class
  is-error.dec-cla           ➔ ho-cla ◄ error.dec-cla
  let
    ((dec-dir, dec-tye, dec-mee, dec-cle), (dec-vat, ‘OK’)) = dec-cla
    ((ho-dir, ho-tye, ho-mee, ho-cle), (ho-vat, ‘OK’))      = ho-cla
  dec-dir = [ ]              ➔ ((ho-dir, ho-tye, ho-mee, ho-cle[ide/dec-cla]), (ho-vat, ‘OK’))
  true         ➔ ((ho-dir[ide/dec-dir], ho-tye, ho-mee, ho-cle[ide/dec-cla]), (ho-vat, ‘OK’))
```

hosting class
k. goszcząca

If pat = ('empty') we construct the declared class "from scratch".

declared class is abstract

Two ways of introducing inner classes into a declared class:
- by inheritance,
- by declaration.

# Auxilary Deep Constructors not in AlgDen-OO

# Categories of deep constructors

all these constructors have an auxiliary character, i.e.
they will not become constructors in AlgDen-OO

no syntactic representation
available to programmers

s-select-cl : Identifier ⟼ Class ⟼ Class    s- surface
select-cl : Path ⟼ Class ⟼ Class

s-remove-cl : Identifier ⟼ Class ⟼ Class
remove-cl : Path ⟼ Class ⟼ Class

s-insert-cl : Identifier x Class ⟼ Class ⟼ Class
insert-cl : Path x Identifier x Class ⟼ Class ⟼ Class

replace-cl : Path x Class ⟼ Class ⟼ Class

deep constructors
are defined
inductively starting
from from surface
constructors

# Selecting a surface inner-class

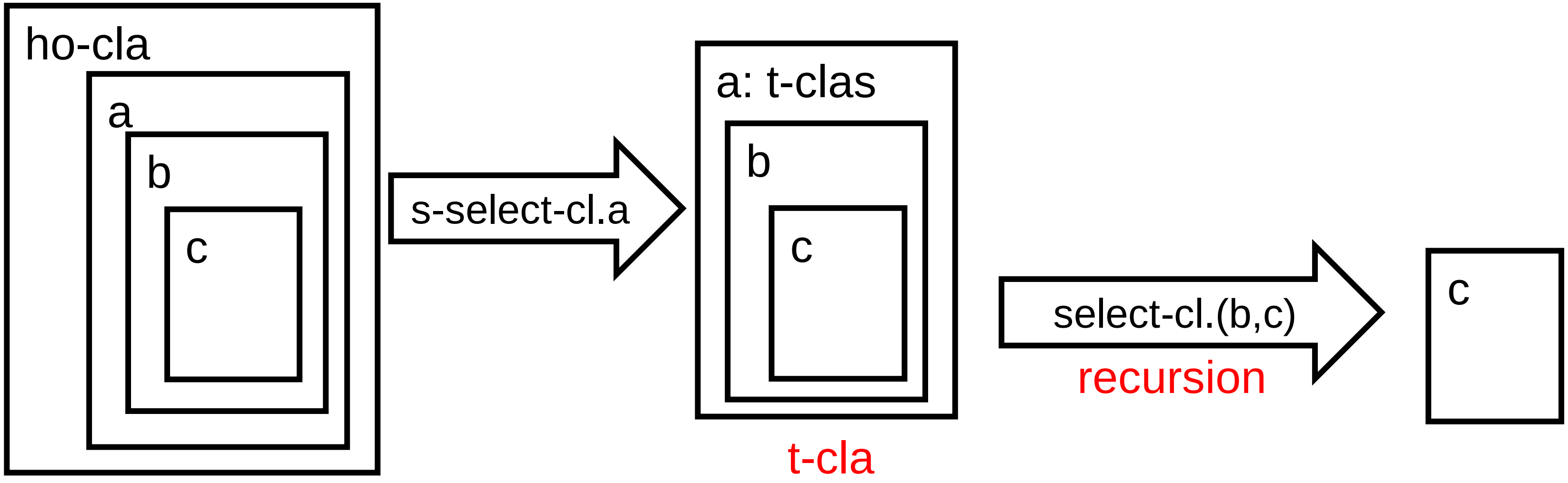
Selecting a class which is a level 1 class of a given class

```
s-select-cl : Identifier ⟼ Class ⟼ Class
s-select-cl.ide.ho-cla =
  is-error.ho-cla   ➔ ho-cla
  let
    ((dir, tye, mee, cle), (vat, err))  = cla
  cle.ide = ?       ➔ ho-cla ◄ 'no-such-class'
  true              ➔ cle.ide
```

# Deep class selection

## Selecting a class at the end of a path

select.(a,b,c).ho-cla



# Deep class selection

## Selecting a class at the end of a path

```
select-cl : Path ⟼ Class ⟼ Class

select-cl.pat.ho-cla =
  is-error.ho-cla    ➔ ho-cla
  pat = ('empty')    ➔ (([ ], [ ], [ ]), ([ ], 'OK'))       an empty class
  pat = ( )          ➔ ho-cla
  let
    ide      = top.pat
    p-pat    = pop.pat
    t-cla    = s-select-cl.ide.cla
  is-error.t-cla     ➔ ho-cla ◄ error.t-cla
  true               ➔ select-cl.p-pat.t-cla                recursion
```

# Removing a surface inner-class

## Removing a class which is level 1 class of a given class

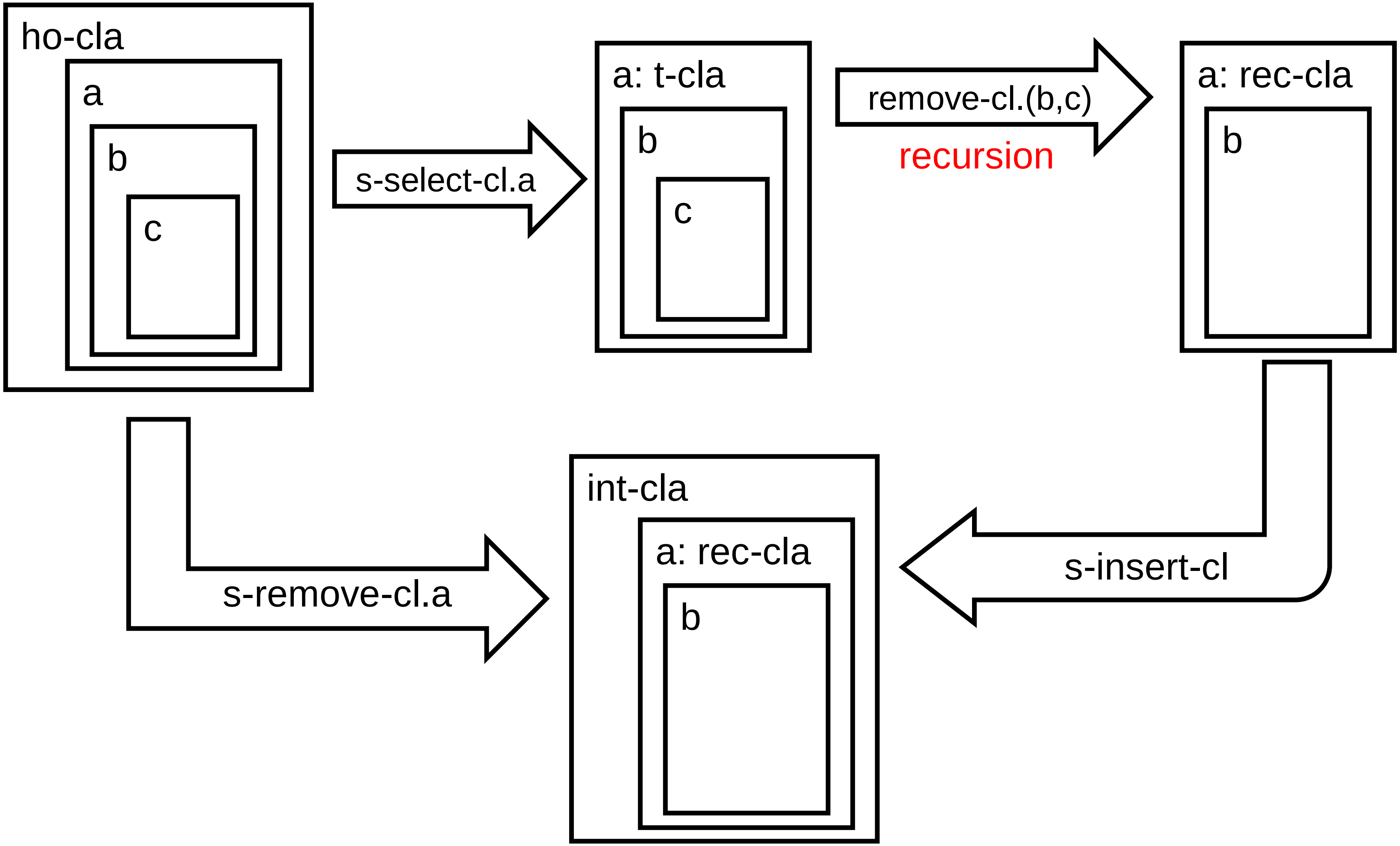
```
s-remove-cl : Identifier ⟼ Class ⟼ Class

s-remove-cl.ide.ho-cla =
  is-error.ho-cla ➔ ho-cla
  let
    ((dir, tye, mee, cle), (vat, err))  = ho-cla
  cle.ide = ?    ➔ ho-cla ◄ 'no-such-class'
  dir.ide  = ?   ➔ ((dir, tye, mee, cle[ide/?]), (vat, err))          concrete-class removal
  true           ➔ ((dir[ide/?], tye, mee, cle[ide/?]), (vat, err))   abstract-class removal
```

# Deep class removal

Removing a class at the end of a path

remove-cl.(a,b,c).ho-cla



# Deep class removal

## Removing a class at the end of a path

```
remove-cl : Path ⟼ Class ⟼ Class

remove-cl.pat.ho-cla =
  is-err.ho-cla      ➔ ho-cla ◄ 'empty-class-cannot-be-removed'
  pat = ('empty')    ➔ ho-cla
  pat = ( )          ➔ ho-cla ◄ 'empty-path'          a class can't be removed
  let                                                 from itself
    ide     = top.pat
    p-pat   = pop.pat
    t-cla   = s-select-cl.ide.ho-cla                  a top class
  is-error.t-cla     ➔ ho-cla ◄ error.t-cla
  let
    int-cla = s-remove-cl.ide.ho-cla                  intermediate class
  is-error.int-cla   ➔ ho-cla ◄ error.int-cla
  p-pat = ( )        ➔ int-cla
  let
    rec-cla = remove-cl.p-pat.t-cla                   recursion
  is-error.rec-cla   ➔ ho-cla ◄ error.rec-cla
  true               ➔ s-insert-cl.(ide, rec-cla).int-cla
```

# Inserting a surface class

## Inserting a class which becomes level 1 class of a given class

s-insert-cl : Identifier x Class ⟼ Class ⟼ Class

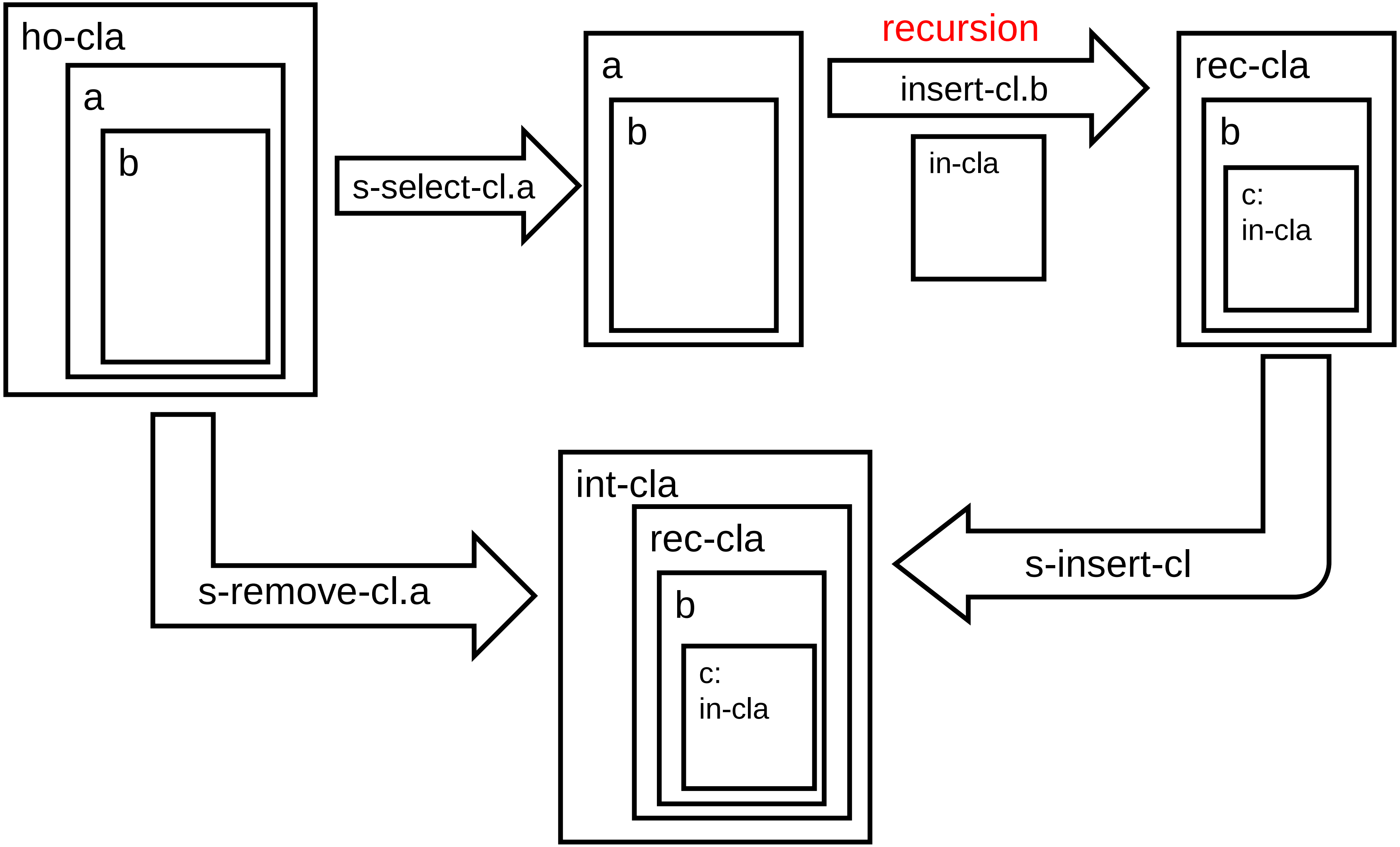
```
s-insert-cl.(ide, in-cla).ho-cla =
  is-error.ho-cla          ➔ ho-cla
  is-error.in-cla          ➔ ho-cla ◄ error.in-cla
  ide : declared.ho-cla    ➔ ho-cla ◄ 'identifier-already-declared'
  let
    ((dir-in, tye-in, mee-in, cle-in), (vat-in, 'OK'))       = in-cla
    ((dir-ho, tye-ho, mee-ho, cle-ho), (vat-ho, 'OK'))  = ho-cla
  dir-in = [ ]      ➔ ((dir-ho, tye-ho, mee-ho, cle-ho[ide/in-cla]), (vat-ho, 'OK'))
  true              ➔ ((dir-ho[ide/dir-in], tye-ho, mee-ho, cle-ho[ide/in-cla]), (vat-ho, 'OK'))
```

in-cla is concrete

# Deep class insertion

## Inserting a class at the end of a path

insert-cl.((a,b), c, in-cla).ho-cla



# Deep class insertion

## Inserting a class at the end of a path

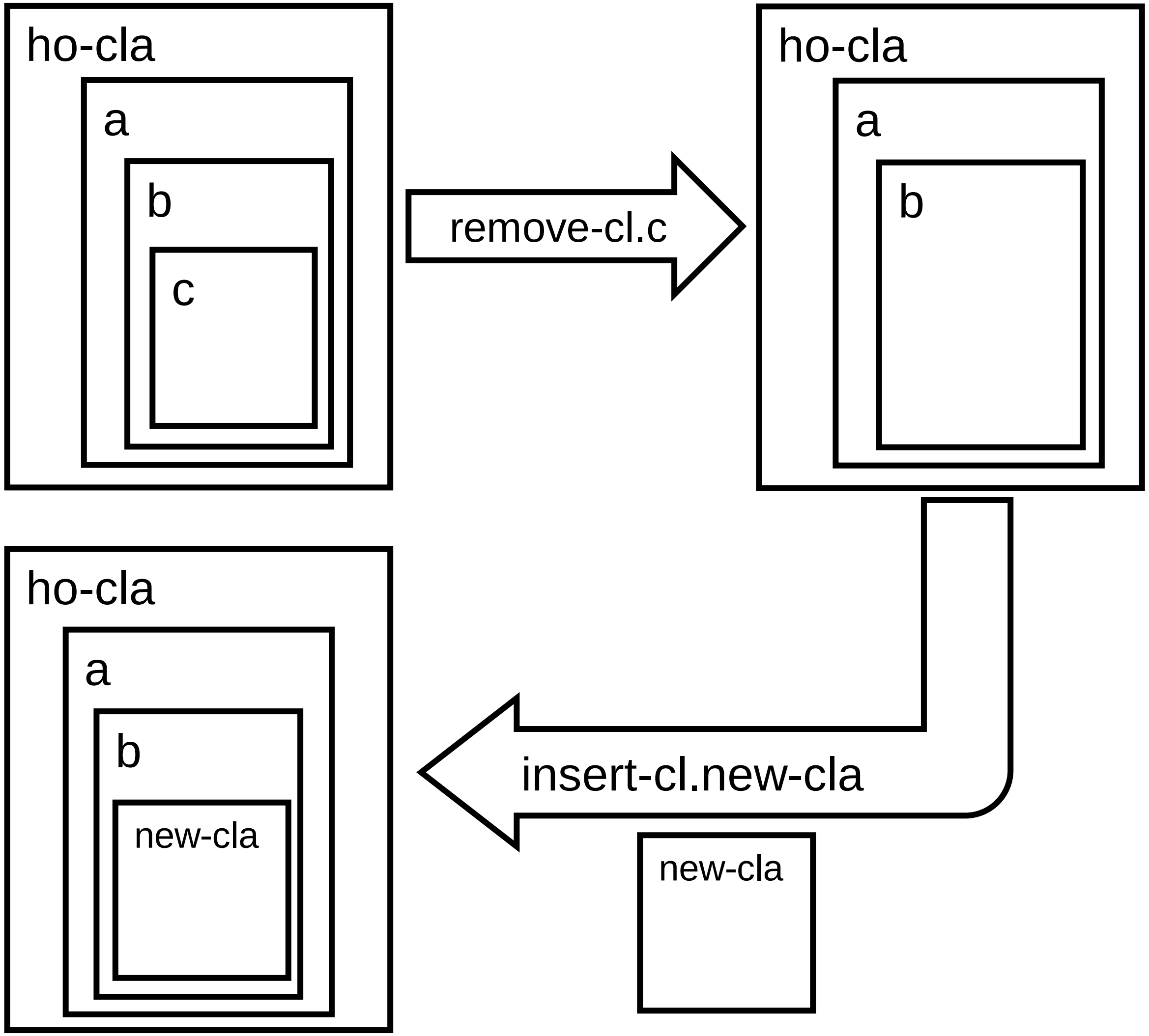
```
insert-cl : Path x Identifier x Class ⟼ Class ⟼ Class

insert-cl.(pat, ide, in-cla).ho-cla =
  is-error.ho-cla     ➔ ho-cla
  is-error.in-cla     ➔ ho-cla ◄ error.in-cla
  pat = ('empty')     ➔ ho-cla ◄ '('empty')-path-not-allowed'
  pat = ( )           ➔ s-insert-cl.(ide, in-cla).ho-cla
  let
    top-ide  = top.pat
    p-pat    = pop.pat
    t-cla    = s-select-cl.top-ide.ho-cla            top class
    int-cla  = s-remove-cl.top-ide.ho-cla            intermediate class
  is-error.t-cla      ➔ ho-cla ◄ error.t-cla
  is-error.int-cla    ➔ ho-cla ◄ error.int-cla
  let
    rec-cla  = insert-cl.(p-pat, ide, int-cla).t-cla   recursion
  is-error.rec-cla    ➔ ho-cla ◄ error.rec-cla
  true                ➔ s-insert-cl.(ide, rec-cla).int-cla
```

# Deep class replacement

## Replacing a class at the end of a path

replace-cl.((a,b), c, new-cla).ho-cla



# Deep class replacement

A general scheme: remove + insert.

Replacement of a class at the end of a path by a new class.

```
replace-cl : Path x Class ⟼ Class ⟼ Class

replace-cl.(pat, new-cla).ho-cla =
  is-error.ho-cla    ➔ ho-cla
  is-error.new-cla   ➔ ho-cla ◄ error.new-cla
  pat = ('empty')    ➔ ho-cla ◄ '('empty')-path-not-allowed'
  pat = ( )          ➔ ho-cla ◄ 'empty-path'
  let
    ide     = last.pat
    sh-pat  = skip-last.pat                short path
    int-cla = remove-cl.pat.ho-cla         intermediate class
    is-error.int-cla ➔ ho-cla ◄ error.int-cla
  true               ➔ insert.(sh-pat, ide, new-cla).int-cla
```

# Surface constructors used at a depth

# A universal operator
# select + modify + replace

ope : Class → Class

deep.ope : Path ⟼ Class → Class

```
deep.ope.pat.ho-cla =
  is-error.ho-cla    ➔ ho-cla
  pat = ('empty')    ➔ ho-cla ◄ '('empty')-path-not-allowed'
  pat = ( )          ➔ ho-cla ◄ 'empty-path'
  let
    so-cla  = select.pat.ho-cla          source class
  is-error.so-cla    ➔ ho-cla ◄ error.so-cla
  let
    mod-cla = ope.so-cla                 modified class
  is-error.mod-cla   ➔ ho-cla ◄ error.mod-cla
  true               ➔ replace-cl.(pat, mod-cla).ho-cla
```

Ten operator pozwala zdefiniować konstruktory odpowiadające głębokiemu wykonaniu dowolnych denotacji powierzchniowych. W szczególności możemy na dowolnej głębokości przekształcić całą klasę dowolnym OO-programem. Czy taką możliwość chcemy dać do ręki programiście?

# Setter — a surface-to-deep instruction

Sets a value computed in a hosting class to an attribute of a deep class.

```
set : Path x Identifier x DatExpDen ⟼ InsDen        i.e.
set : Path x Identifier x DatExpDen ⟼ Class → Class
set.(pat, ide, ded).ho-cla =
  is-error.ho-cla       ➔ ho-cla
  let
    ta-cla = select.pat.ho-cla                                   target class
  is-error.ta-cla       ➔ ta-cla                       in this case ta-cla = ho-cla
  ded.ho-cla = ?        ➔ ?
  let
    val = ded.ho-cla
  val : Error           ➔ho-cla ◄ val
  let
    ((ta-dir, ta-tye, ta-pre, ta-cle), (ta-sto, ‘OK’)) = ta-cla
  ta-sto.ide = ?        ➔ ‘no-such-attribute’
  let
    ca-cla = ((ta-dir, ta-tye, ta-pre, ta-cle), (ta-sto[ide/val], ‘OK’))  concrete attribute
    aa-cla = ((ta-dir[ide/ ?], ta-tye, ta-pre, ta-cle), (ta-sto[ide/val], ‘OK’))    abst. attr.
  ta-dir.ide = ?        ➔ replace-cl.(pat, ca-cla).ho-cla
  true                  ➔ replace-cl.(pat, aa-cla).ho-cla
```

pat = ( ) - surface assignment
pat = ('empty') - yields a simple concrete class

# Getters – deep-to-surface expressions

## A general scheme: select + evaluate

data getter

d-get : Path x DatExpDen ⟼ Class → DatExpDen        i.e.

d-get : Path x DatExpDen ⟼ Class → Value | Error

d-get.(pat, ded).ho-cla = ded.(select-cl.pat.ho-cla)

type getter

t-get : Path x TypExpDen ⟼ Class → TypExpDen        i.e.

t-get : Path x TypExpDen ⟼ Class → Value | Error

t-get.(pat, ted).ho-cla = ted.(select-cl.pat.ho-cla)

Expression represented by ded/ted is evaluated in an inner class of cla indicated by the path pat.

Both constructions include deep calls of functional/typological procedures.

# Deep call of a procedure

call-imp-proc : Identifier x AcPaDe x AcPaDe ⟼ Class → Class

deep-call-imp-proc : Path x Identifier x AcPaDe x AcPaDe ⟼ Class → Class

deep-call-imp-proc.(pat, ide, apd-v, apd-r).cla =
  call-imp-proc.(ide, apd-v, apd-r).(select-cl.pat.cla)

This means that
  call-imp-proc.(ide, apd-v, apd-r)
is executed in the deep class
  deep-cla = select.pat.cla

Consequently:

- ide must be declared in deep-cla,
- actual parameters are computed in deep-cla.

If we want to pass actual parameters stored in the main-class store to a deep procedure call, we have to export them via setters to the deep class.

# I na tym na razie koniec
# c.d.n.