Lingua Project

(7 bis) Methods

(Sec. 6.6)

The book "**Denotational Engineering**" may be downloaded from:
https://moznainaczej.com.pl/what-has-been-done/the-book

Andrzej Jacek Blikle

March 22th, 2025

An overview of methods

Three categories of methods:

- imperative methods
- object-constructors
- functional methods

concrete methods

abstract methods

met : Method = Procedure | ProSigDen

procedures

pro : Procedure = ImpPro | FunPro | ObjCon

ipr : ImpPro = ActParDen x ActParDen ⟼ Store → Store
fpr : FunPro = ActParDen x TypExpDen ⟼ Store → ValueE
oco : ObjCon = ActParDen x Identifier ⟼ Store → Store

not in AlgDen

procedure signatures

prs : ProSigDen = ImpProSigDen | FunProSigDen | ObjConSigDen

ips : ImpProSigDen = ForParDen x ForParDen
fps : FunProSigDen = ForParDen x TypExpDen
ocs: ObjConSigDen = ForParDen x Identifier

Why procedures modify stores?

(rather than states?)

if procedures were modifying states

(an illegal recursion)

Procedure = State → State

State = Env x Sto

Env = ClaEnv x ProEnv x CovRel

ProEnv = Identifier ⟹ Procedure

Preprocedures

ppr	: PrePro	= ImpPrePro | FunPrePro | ObjPreCon	pre-procedures
ipp	: ImpPrePro	= Env ⟼ ImpPro	imperative pre-procedures
fpp	: FunPrePro	= Env ⟼ FunPro	functional pre-procedures
opc	: ObjPreCon	= Env ⟼ ObjCon	object pre-constructors

Preprocedures are necessary to describe
mutually recursive procedures
declared in different classes.

When a procedure pro is called, we execute the corresponding pre-pro in a declaration time environment dt-env, i.e., we execute the function

pre-pro.dt-env : Store → Store

Signatures and parameters

loi	: ListOfIde	= Identifier^{c*}	lists of identifiers
dse	: DecSec	= ListOfIde x TypExpDen	declaration sections
fpd	: ForParDen	= DecSec^{c*}	formal-parameter-denotations
apd	: ActParDen	= ListOfIde	actual-parameter-denotations

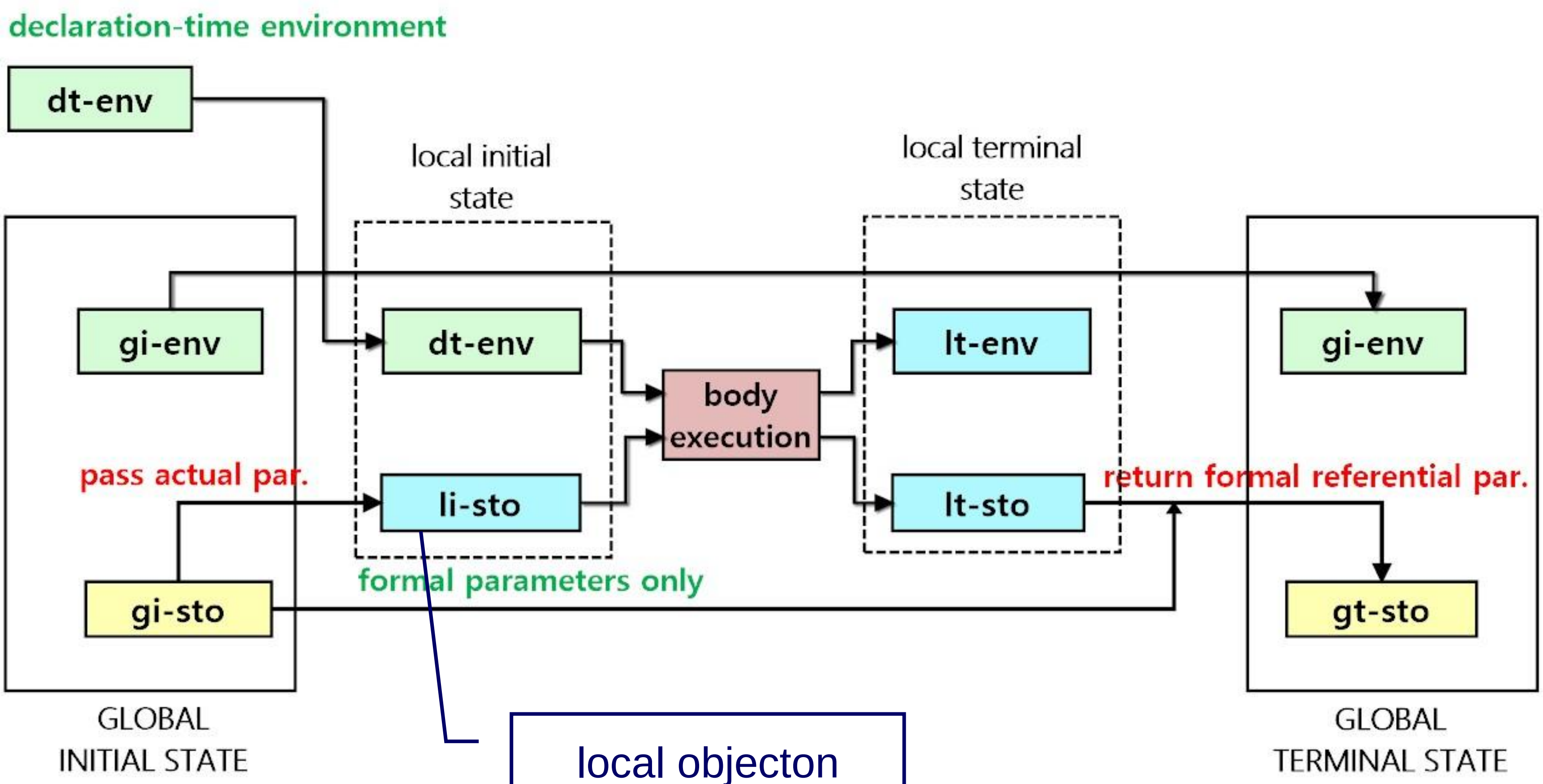
constructors

build-loi	: Identifier	$\longmapsto$ ListOfIde	
add-to-loi	: Identifier x ListOfIde	$\longmapsto$ ListOfIde	
build-dse	: ListOfIde x TypExpDen	$\longmapsto$ DecSec	
build-fpd	: DecSec	$\longmapsto$ ForParDen	
add-to-fpd	: DecSec x ForParDen	$\longmapsto$ ForParDen	
build-apd	: ListOfIde	$\longmapsto$ ActParDen	actual-par. denot.
build-ipsd	: ForParDen x ForParDen	$\longmapsto$ ImpProSigDen	signatures of IP
build-fpsd	: ForParDen x TypExpDen	$\longmapsto$ FunProSigDen	signatures of FP
build-ocsd	: ForParDen x Identifier	$\longmapsto$ ObjConSigDen	signatures of OC

value parameters

reference parameters

The execution of an imperative-procedure call



local objecton
global deposit

The creation of an imperative pre-procedure

create-imp-pre-pro : ImpProSigDen x ProDen x Identifier ⟼ ImpPrePro
create-imp-pre-pro : ForParDen x ForParDen x ProDen x Identifier ⟼
                    ⟼ Env ⟼ ActParDen x ActParDen ⟼ Store → Store

```
create-imp-pre-pro.(fpd-v, fpd-r, prd, cl-ide).dt-env.(apd-v, apd-r).ct-sto =
  is-error.ct-sto      ➔ ct-sto                              dt-  declaration time
  let                                                        ct-  call time
    li-sto = pass-actual.(fpd-v, fpd-r, apd-v, apd-r, cl-ide).dt-env.ct-sto
  is-error.li-sto    ➔ ct-sto ◄ error.li-sto
  let
    li-sta = (dt-env, li-sto)                                local initial state
  prd.li-sta = ?       ➔ ?
  let
    lt-sta = prd.li-sta                                      local terminal state
  is-error.lt-sta      ➔ ct-sto ◄ error.lt-sta
  let
    (dt-cle, dt-pre, dt-cov)  = dt-env
    (lt-env, lt-sto)          = lt-sta
    gt-sto                    = return-formal.fpd-r.ct-sto.lt-sto.dt-cov
  is-error.gt-sto      ➔ ct-sto ◄ error.gt-sto
  true                 ➔ gt-sto                              global terminal store
```

Siblings and twins of objects

in a context of a common deposit

INFORMAL DEFINITIONS

A sibling of an object:

- the same structure (tree),
- the same names of attributes,
- references differ only with tokens,
- possibly different „leaves” (typed data).

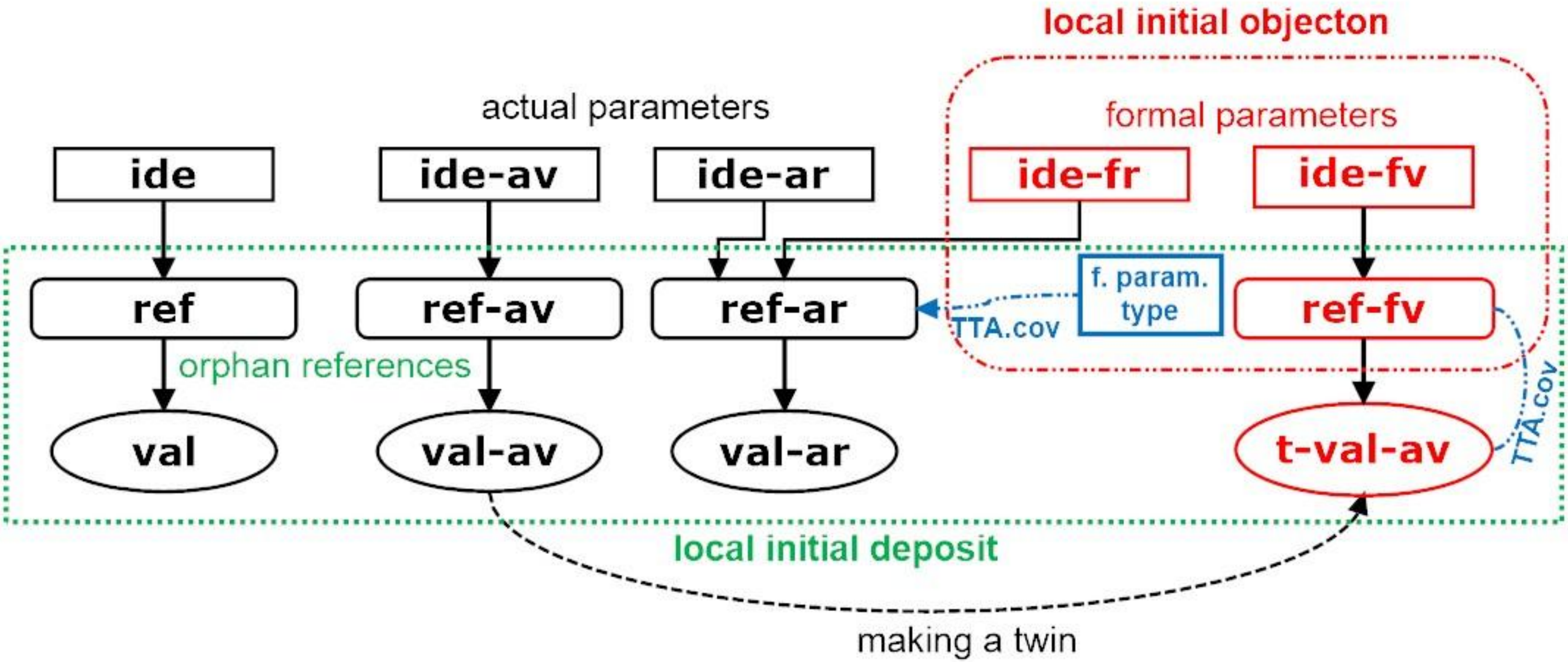
A twin of an object:

- a sibling with identical leaves (differ only with references)

Formal definitions in Sec. 4.5

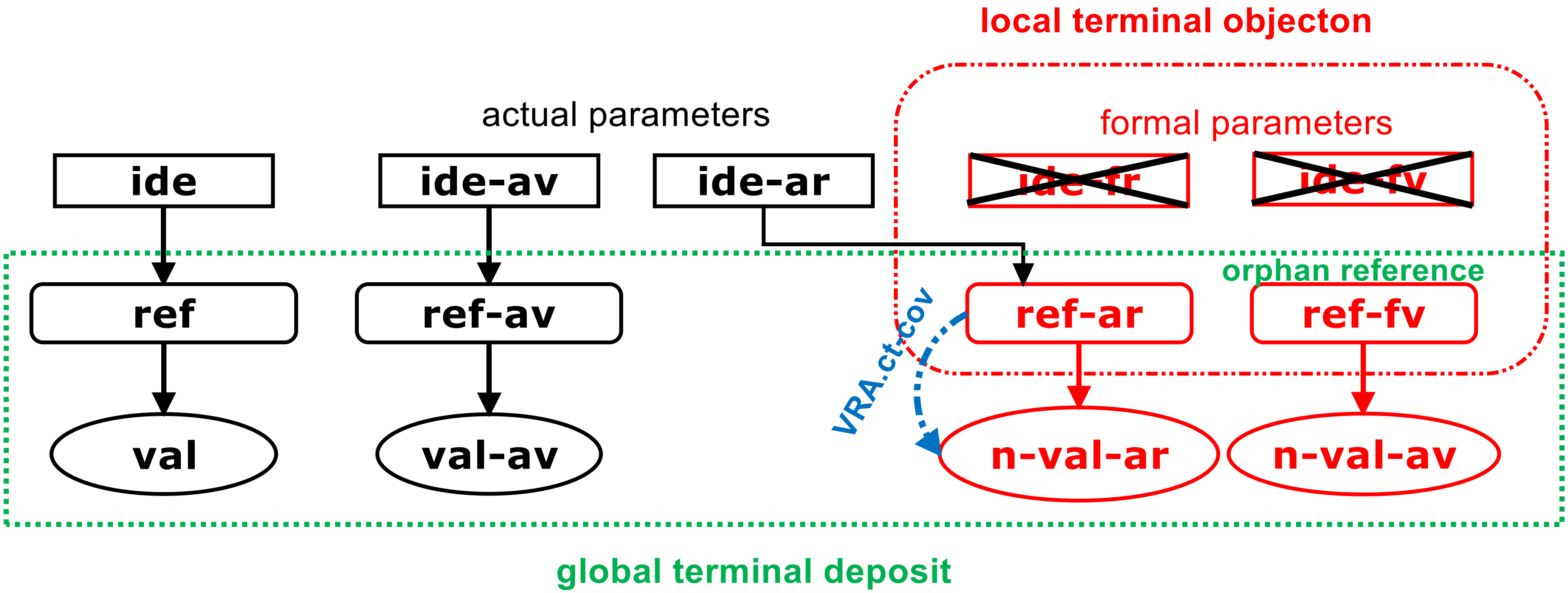
Passing actual parameters to a procedure

a simplified picture



Returning references to actual ref. parameters

a simplified picture



Global terminal state:

- global call-time environment (unchanged)
- global terminal deposit:
  - global call-time objecton; all global variables regain visibility,
  - local terminal deposit; some references become orphans,
  - call-time origin of the store,
  - call-time covering relations (a type checking necessary)

Calling an imperative procedure

```
call-imp-pro : Identifier x Identifier x ActParDen x ActParDen ⟼ InsDen
call-imp-pro : Identifier x Identifier x ActParDen x ActParDen ⟼
                                                              WfState ⟶ WfState
call-imp-pro.(cl-ide, pr-ide, apd-v, apd-r).ct-sta =
  is-error.ct-sta                    ➔ ct-sta
  let
    (ct-env, ct-sto)  = ct-sta                                  call-time state
    (ct-cle, ct-pre)  = ct-env
  ct-pre.(cl-ide, pr-ide) = ?        ➔ ct-sta ◄ ‘procedure-unknown’
  ct-pre.(cl-ide, pr-ide) /: ImpPro  ➔ ct-sta ◄ ‘imperative-procedure-expected’
  let
    ipr = ct-pre.(cl-ide, pr-ide)
  ipr.(apd-v, apd-r).ct-sto = ?      ➔ ?
  let
    gt-sto = ipr.(apd-v, apd-r).ct-sto                     global terminal store
  true                               ➔ (ct-env, gt-sto)
```

The creation of a functional pre-procedure

```
create-fun-pre-pro : FunProSigDen x ProDen x ValExpDen x Identifier ⟼
                                        ⟼ Env ⟼ ActParDen ⟼ Store → ValueE

create-fun-pre-pro.(fps,  prd, ved, cl-ide).dt-env.apd.ct-sto =
  is-error.ct-sto                  ➔ error.ct-sto      dt- creation time, ct- call time
  let
    (ct-obn, ct-dep, ct-sft, ct-ota, 'OK')  = ct-sto
    (fpd, ted)                               = fps        functional-procedure signature
    li-sto = pass-actual.(fpd, (), apd, (), cl-ide ).dt-env.ct-sto        local initial store
  is-error.li-sto                  ➔ error.li-sto
  let
    li-sta   = (dt-env, li-sto)                                            local initial state
    ex-typ  = ted.li-sta                          the expected type of the returned value
  ex-typ : Error                   ➔ ex-typ
  (prd • ved).li-sta= ?            ➔ ?
  (prd • ved).li-sta : Error       ➔ (prd • ved).li-sta
  let
    (cor, typ) = (prd • ved).sta-li                                        lt- local terminal
  not ex-typ TTA.ct-cov typ ➔ ‘types-incompatible’
  true                             ➔ (cor, ex-typ)
```

If f : A → B and g : B → C then (f•g).a = g.(f.a)

Calling a functional procedure

call-fun-pro : Identifier x Identifier x ActParDen ⟼ValExpDen

```
call-fun-pro : Identifier x Identifier x ActParDen ⟼ WfState ⟶ ValueE
call-fun-pro.(cl-ide, pr-ide, apd).ct-sta =                              ct- call time
  is-error.ct-sta                   ➔ ct-sta
  let
    ((cle, pre, cov), ct-sto) = ct-sta
  pre.(cl-ide, pr-ide) = ?          ➔ ‘procedure-unknown’
  pre.(cl-ide, pr-ide) /: FunPro    ➔ ‘functional-procedure-expected’
  let
    fpr = pre.(cl-ide, pr-ide)
  fpr.apd.ct-sto = ?                ➔ ?
  true                              ➔ fpr.apd.ct-sto
```

Object constructors informally

oco : ObjCon = ActParDen x Identifier ⟼ Store → Store

opc : ObjPreCon = Env ⟼ ObjCon object preconstructors

create-obj-pre-con : ObjConSigDen x ProDen ⟼ ObjPreCon

create-obj-pre-con : ForParDen x Identifier x ProDen ⟼
⟼ Env ⟼ ActParDen x Identifier ⟼ Store → Store

The store-to-store action of an abject constructor in a class ’MyClass’:

1. it creates (obn, ’MyClass’) where obn is a twin of the objecton of MyClass
2. it (optionally) modifies current deposit by executing a program; the twin may become a sibling,
3. it creates a reference, assigns it to ide and assigns the sibling to this reference.

Calling an object constructor

```
call-obj-con : Identifier x Identifier x Identifier x ActParDen ⟼ InsDen
call-obj-con : Identifier x Identifier x Identifier x ActParDen
                                                      ⟼ WfState ⟶ WfState
call-obj-con.(ob-ide, cl-ide, co-ide, apd-v).sta =
  is-error.ct-sta                        ➔ ct-ct-sta
  let
    ((ct-cle, ct-pre), ct-sto) = ct-sta                              call-time state
  ct-pre.(cl-ide, co-ide) = ?            ➔ ct-sta ◄ ‘constructor unknown’
  ct-pre.(cl-ide, co-ide) /: ObjCon      ➔ ct-sta ◄ ‘object constructor expected’
  let
    oco = ct-pre.(cl-ide, co-ide)
  oco.(apd-v, ob-ide).ct-sto = ?         ➔ ?
  let
    new-sto = oco.(apd-v, ob-ide).ct-sto
  true                                   ➔ (ct-env, new-sto)
```

Thank you for your attention