Lingua Project

(3) Yokes, objects, classes and states

(Sec. 4.4, 4.5, 5.1, 5.2, 5.3)

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

Andrzej Jacek Blikle

February 8th, 2025

Recapitulation of former lecture

Data domains

Data

ide : Identifier — a finite subset of $Character^+$

dat : Data = SimpleData | List | Array | Record
dat : SimpleDat = Boolean | Integer | Real | Text
int : Integer = $[-2^{30}, 2^{30}-1]$ an example
rea : Real = $[-1{,}8 \times 10^{308}, 1{,}8 \times 10^{308}]$ an example
boo : Boolean = {tt, ff}
tex : Text = {‘}Character*{‘} with len.tex ≤ $2^{24} - 5$ an example
lis : List = $Data^{c*}$
arr : Array = Integer ⟹ Data
rec : Record = Identifier ⟹ Data

domain recursion

Data domains are <u>supersets</u> of future reachable domains,
e.g. non-homogeneous lists of arbitrary length or arrays with indexes -4, 0, 3, 5

A

Recapitulation of former lecture

Datatypes

Datatypes are finitistic elements that describe structures of (corresponding) data

typ : DatTyp =
{'integer', 'real', 'boolean', 'text'} | simple types
{'L'} x DatTyp | list types
{'A'} x DatTyp | array types
{'R'} x (Identifier ⟹ DatTyp) record types

type record

record typów

An engineering decision is announced here:
Non-homogeneous list and arrays are not allowed.

Examples of types:

('L', ('R', [name/('word'), age/('integer')])) a type of lists of records
('A', ('L', ('R', [name/('word'), age/('integer')]))) a type of arrays of lists of records

A

Recapitulation of former lecture

Typed data

tyd : TypDat = {(dat, typ) | dat : CLAN-ty.typ}

Why shall we use typed data rather than just data?

1. Not all data have types, e.g., nonhomogeneous arrays have no types.
2. Typed data allow us to describe our typing discipline and eliminate typing errors.
3. We can describe type-covering relations.
4. Types will be used in checking the typing adequacy of:
   a. arguments of functions in the evaluation of expressions,
   b. actual parameters of procedures.

Another auxiliary function:
sort-td.(dat, typ) = sort-t.typ

Yokes

domain and first examples

jarzma

- Yokes describe these properties of typed data that can't be described by types.
- Yokes will be assigned to variables in states (by declarations).
- In SQL yokes are known as integrity constraints.

yok : Yoke = TypDatE ⟼ TypDatE
yok : BooYok = TypDatE ⟼ { (tt, 'boolean'), (ff, 'boolean') } a special case

Examples of boolean yoke expressions:

1,93 ≤ **value** ≤ 2,47
record.salary + **record.**commission < 7000
TT - always satisfied
small integer
sorted list

an example of a non-boolean yoke expression:

record.salary + **record.**commission

The clans of yokes:
CLAN-Yo : Yoke ⟼ Sub.TypDat
CLAN-Yo.yok = {com | yok.com = (tt, ('boolean'))}

A

Yokes
the signatures of constructors

Specific constructors not derived from constructors of typed data

yo-pass	:	$\longmapsto$ Yoke	
yo-sum-li-in	:	$\longmapsto$ Yoke	
yo-give-td	: TypDat	$\longmapsto$ Yoke	a constant-td (typed data) yoke

Constructors derived from simple typed-data constructors (except boolean)

yo-add-in	: Yoke x Yoke	$\longmapsto$ Yoke	
yo-subtract-in	: Yoke x Yoke	$\longmapsto$ Yoke	
yo-multiply-in	: Yoke x Yoke	$\longmapsto$ Yoke	
yo-divide-in	: Yoke x Yoke	$\longmapsto$ Yoke	

(analogously for reals)

Constructors derived from selection constructors for structured typed data

yo-top	:	$\longmapsto$ Yoke	
yo-get-from-ar	: TypDat	$\longmapsto$ Yoke	ar- stands for “array”
yo-get-from-rc	: Identifier	$\longmapsto$ Yoke	rc- stands for “record

Yokes
the signatures of constructors (cont.)

Constructors of yokes based on predicates

yo-equal-in	: Yoke x Yoke	$\longmapsto$ Yoke	
yo-less-in	: Yoke x Yoke	$\longmapsto$ Yoke	
yo-no-repet-li	:	$\longmapsto$ Yoke	li- stands for “list”
yo-increasing-li-in	:	$\longmapsto$ Yoke	

Constructors of yokes based on Kleene’s propositional operators

yo-true	:	$\longmapsto$ Yoke
yo-and	: Yoke x Yoke	$\longmapsto$ Yoke
yo-or	: Yoke x Yoke	$\longmapsto$ Yoke
yo-not	: Yoke	$\longmapsto$ Yoke
yo-all-of-li	: Yoke	$\longmapsto$ Yoke
yo-exists-in-li	: Yoke	$\longmapsto$ Yoke
yo-all-of-ar	: Yoke	$\longmapsto$ Yoke
yo-exists-in-ar	: Yoke	$\longmapsto$ Yoke

Yokes

some definitions of constructors

yo-pass : ⟼ Yoke
yo-pass.() = pass
pass.tyd = tyd

An example of application: the denotation of yoke expression **value** + 2

yo-add-in.(pass, yo-in.2).tyd = td-add-in.(pass.tyd, yo-in.2.tyd)
= td-add-in.(tyd, (2, ‘integer’))

it takes two yokes as arguments

Getting a value from an array:

yo-get-from-ar : TypDat ⟼ Yoke
yo-get-from-ar : TypDat ⟼ TypDatE ⟼ TypDatE
yo-get-from-ar.ind-tyd.tyd =

ind- stands for “index”

tyd : Error ➔ tyd
sort-t.ind-tyd ≠ ‘integer’ ➔ ‘integer expected’
sort-t.tyd ≠ ‘A’ ➔ ‘array expected’
let
(dat, (‘A’, typ)) = tyd
dat.ind-tyd = ? ➔ ‘index out of scope’
true ➔ (dat.ind-tyd, typ)

Yokes

a boolean constructor of yokes

yo-and : Yoke x Yoke ⟼ Yoke

```
yo-and.(yok-1, yok-2).tyd =
  tyd : Error                ➔ tyd
  let
    tyd-i = yok-i.tyd                                  for i = 1, 2
  sort-td.tyd-i ≠ ‘boolean’  ➔ ‘boolean expected’     for i = 1, 2
  tyd-i = (ff, ‘boolean’)    ➔ (ff, ‘boolean’)         for i = 1, 2
  tyd-i : Error              ➔ tyd-i                   for i = 1, 2
  true                       ➔ (tt, ‘boolean’)
```

One false argument is enough to falsify conjunction (Kleene style).

Values, objectons, references and deposits

on a way to classes, object and memory states

val	: Value	= TypDat | Object	values
obj	: Object	= Objecton x ObjTyp	objects
obn	: Objecton	= Identifier ⟹ Reference	objectons
typ	: ObjTyp	= Identifier	object types
ref	: Reference	= Token x Profile	references
tok	: Token	= … (e.g. memory locations)	tokens
prf	: Profile	= Type x Yoke x OriTag	profiles
typ	: Type	= DatTyp | ObjTyp	types
yok	: Yoke	= TypDatE ⟼ TypDatE	yokes
ota	: OriTag	= Identifier | {$}	origin tags
dep	: Deposit	= Reference ⟹ Value	deposits

Profile of a reference ref = (tok, (typ, yok, ota)) indicates:

typ – the type of values assignable to ref,

yok – other properties of values assignable to ref,

ota – the visibility status of ref; $ for public visibility, ide – private for a class named ide.

A

Notational conventions

ide → ref we say that ide points to ref, if obn.ide = ref,
ref → val we say that ref points to val, if dep.ref = val,

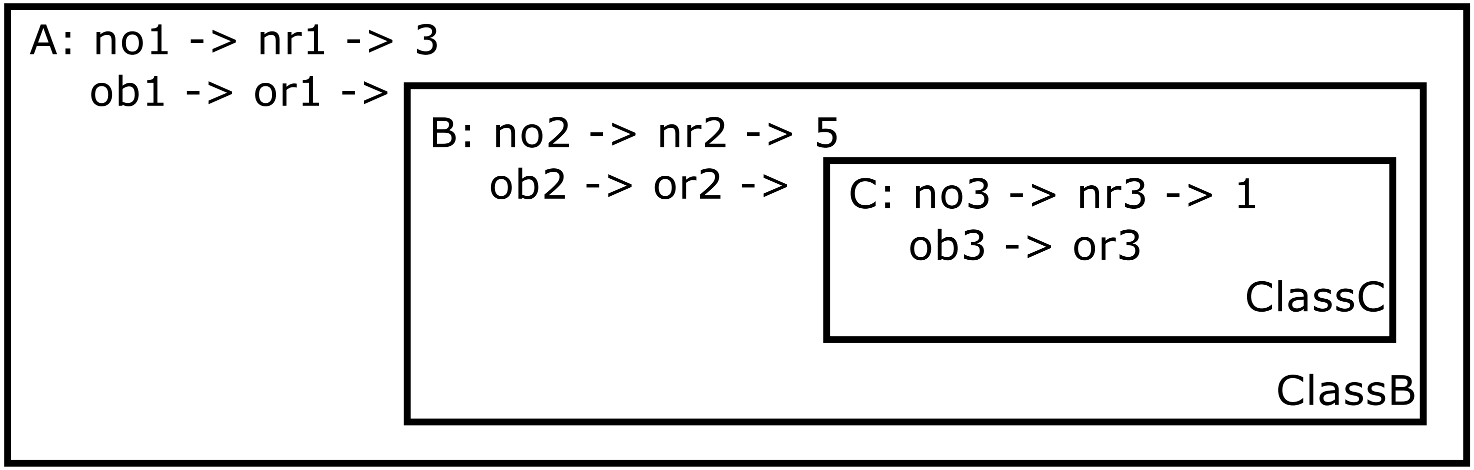
Three situations are possible:
ide → ref → val a standard situation; ide has been declared and initialized,
ide → ref ide has been declared but not initialized; a dangling reference
ref → val ref is an orphan reference; may happen in procedure calls

The sorts of values:

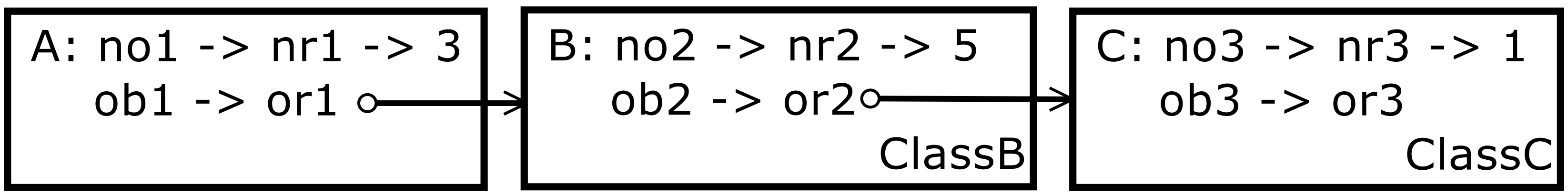
sort-va : Value ⟼ {'boolean', 'integer', 'real', 'text', 'L', 'A', 'R'} | {'object'}
sort-va.val =
val : TypDat ➔ sort-td.val
true ➔ 'object'

Two views of an objecton

Structure view



Graph view



Note the difference:

object's attribute – an attribute assigned to an object (in an objecton)

attribute of an object – an attribute assigned to a reference in an objecton

Classes

definition

cla	: Class	= Identifier x TypEnv x MetEnv x Objecton	classes
tye	: TypEnv	= Identifier ⟹ Type | {Θ}	type environments
mee	: MetEnv	= Identifier ⟹ Method	method environments
met	: Methods	= ProSig | PrePro	methods

Classes

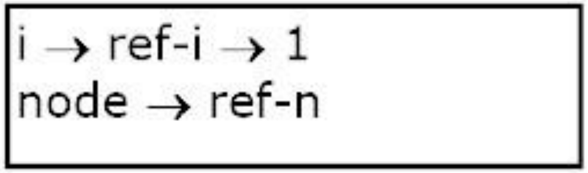
example 1/3

```
class ListNode
  let no = 23 be integer and public tel;
  let next  be ListNode and public tel                    objecton
  cons ConstructObject(val number as integer, node as ListNode return ListNode)
    no      := number + 1;
    next    := node                                 constructor declaration
  snoc
  proc BuildCircularList()
    let i be integer  tel;                          procedure declaration
    let node be ListNode tel;
    i := 1;
    while i <= 3
      do
        node  := ListNode.ConstructObject(i, node);
        i        := i+1
      od;
    node.next.no       := 11
    node.next.next     := node;
  corp
ssalc;
```

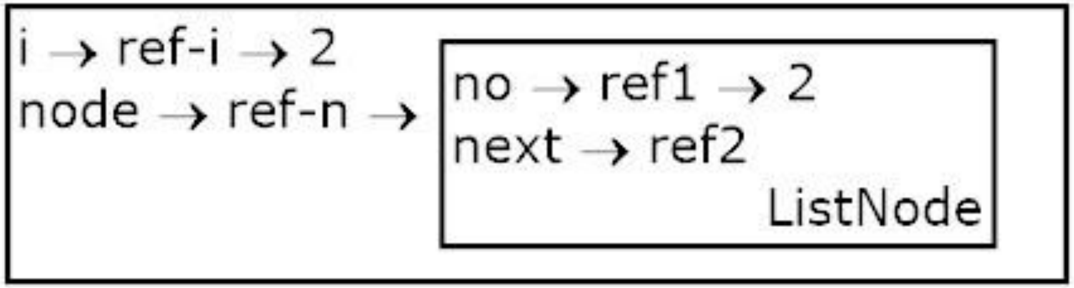
Classes

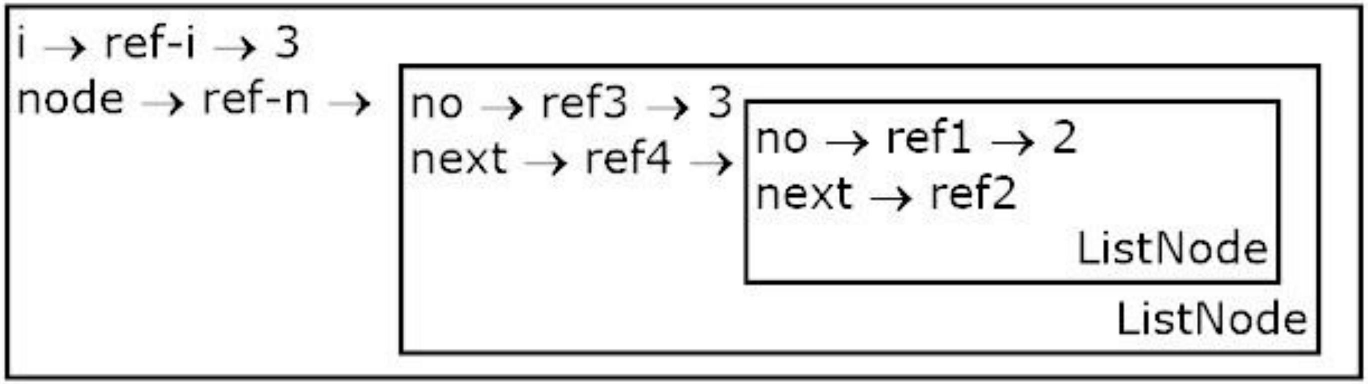
example 2/3

execute procedure call: ListNode.BuildCircularList()



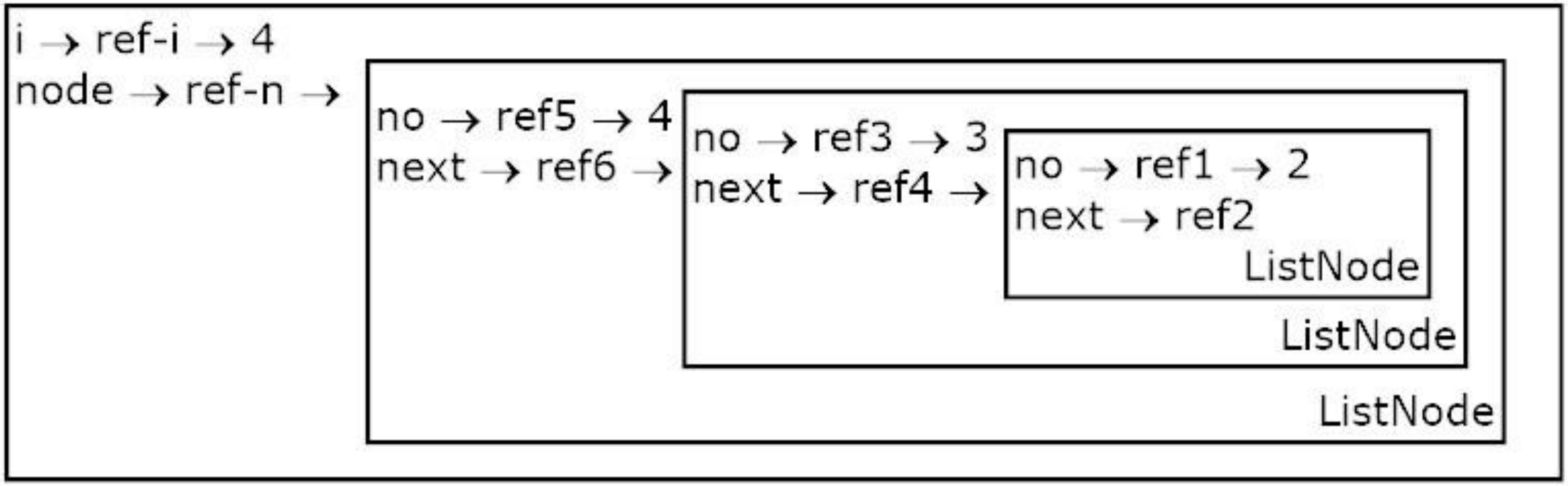
This objecton results from the declaration of local attributes of the procedure.

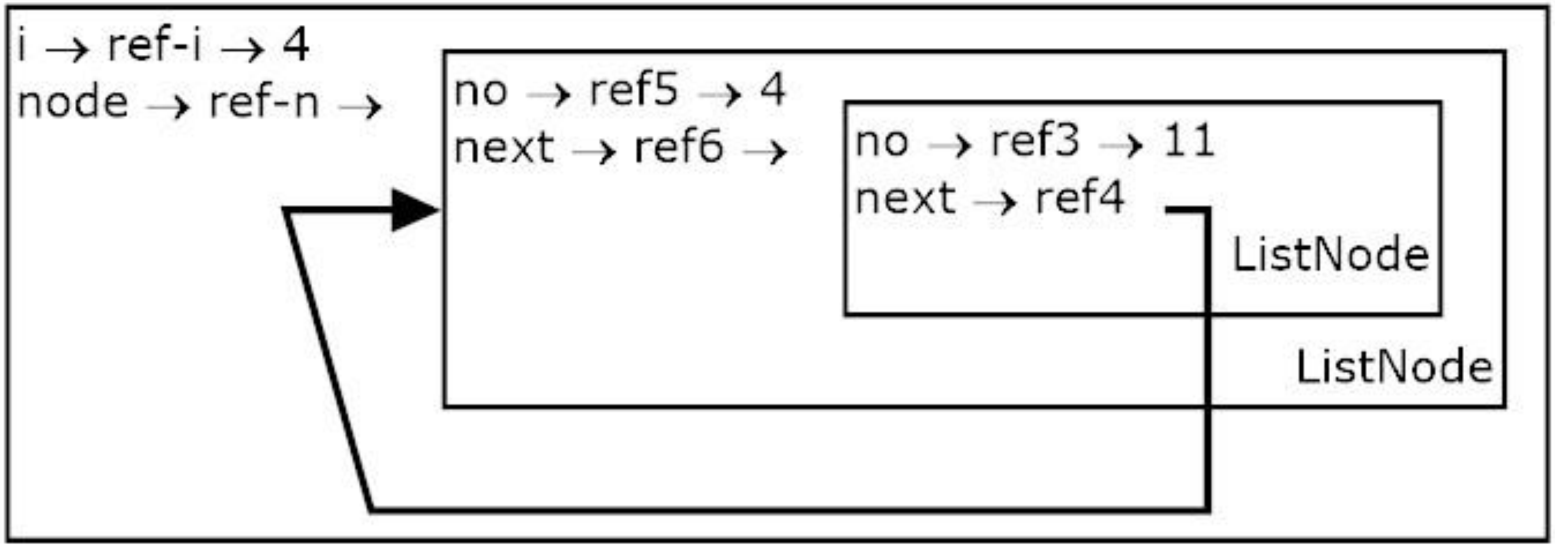




Classes

example 3/3





Stores and states

sta	: State	= Env x Store	states
env	: Env	= ClaEnv x ProEnv x CovRel	environments
cle	: ClaEnv	= Identifier ⟹ Class	class environments
pre	: ProEnv	= ProInd ⟹ Procedure	procedure environments
pri	: ProInd	= Identifier x Identifier	procedure indicators
sto	: Store	= Objecton x Deposit x OriTag x SetFreTok x (Error | {‘OK’})	stores
cov	: CovRel	= Sub.((DatTyp x DatTyp) | (ObjTyp x ObjTyp))	covering relations
sft	: SetFreTok	= Set.Token	sets of (free) tokens

Auxiliary function

get-tok : SetFreTok ⟼ Token x SetFreTok
get-tok.sft = (tok, sft − {tok})) such that tok : sft

An objecton my-obn is said to be well-formed in a state
sta = ((cle, pre, cov), (obn, dep, ota, sft, err)), if:
(1) for any attribute ide, if obn.ide = !, and dep.(obn.ide) = !, then:
obn.ide **VRA.cov** dep.(obn.ide) — value by reference acceptability (see later),
(2) all inner objectons of obn are well-formed in sta.

A class (ide, tye, mee, obn) is said to be well-formed in a state, if
(1) obn is well-formed in this state,
(2) for every reference (tok, (typ, yok, ota)) in obn, its origin tag ota is either $ or ide

Well-formed states

A state sta = ((cle, pre, cov), (obn, dep, ota, sft, err)) said to be well-formed, if:

1. obn is well formed in sta,
2. external names of all classes declared in cle coincide with their internal names,
3. all surface and inner objects in obn are of types that are the names of classes declared in cle,
4. all classes declared in cle are well-formed,
5. sft includes only such tokens that do not appear in references bound in dep,
6. every identifier appearing in a state, appears in it only once; e.g., if an identifier is a variable, it can't be at the same time a type constant or a class name.

WfState − the set of all well-formed states

Auxiliary functions:

error	: Store ⟼ Error | {'OK'}	error	: State ⟼ Error | {'OK'}
is-error	: Store ⟼ Boolean	is-error	: State ⟼ Boolean

(env, (obn, dep, ota, sft, err)) ◄ new-err = (env, (obn, dep, sft, ota, new-err))
(env, (obn, dep, ota, sft, err)) ◄ new-sft = (env, (obn, dep, new-sft, ota, err))

declared : Identifier x State ⟼ {tt, ff}


Thank you for
your attention