Lingua Project

(5) The denotations of expressions

(Sec. 6.1 – 6.4)

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

Andrzej Jacek Blikle

February 22nd, 2025

A recapitulation of former lecture

Classes

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

A recapitulation of former lecture

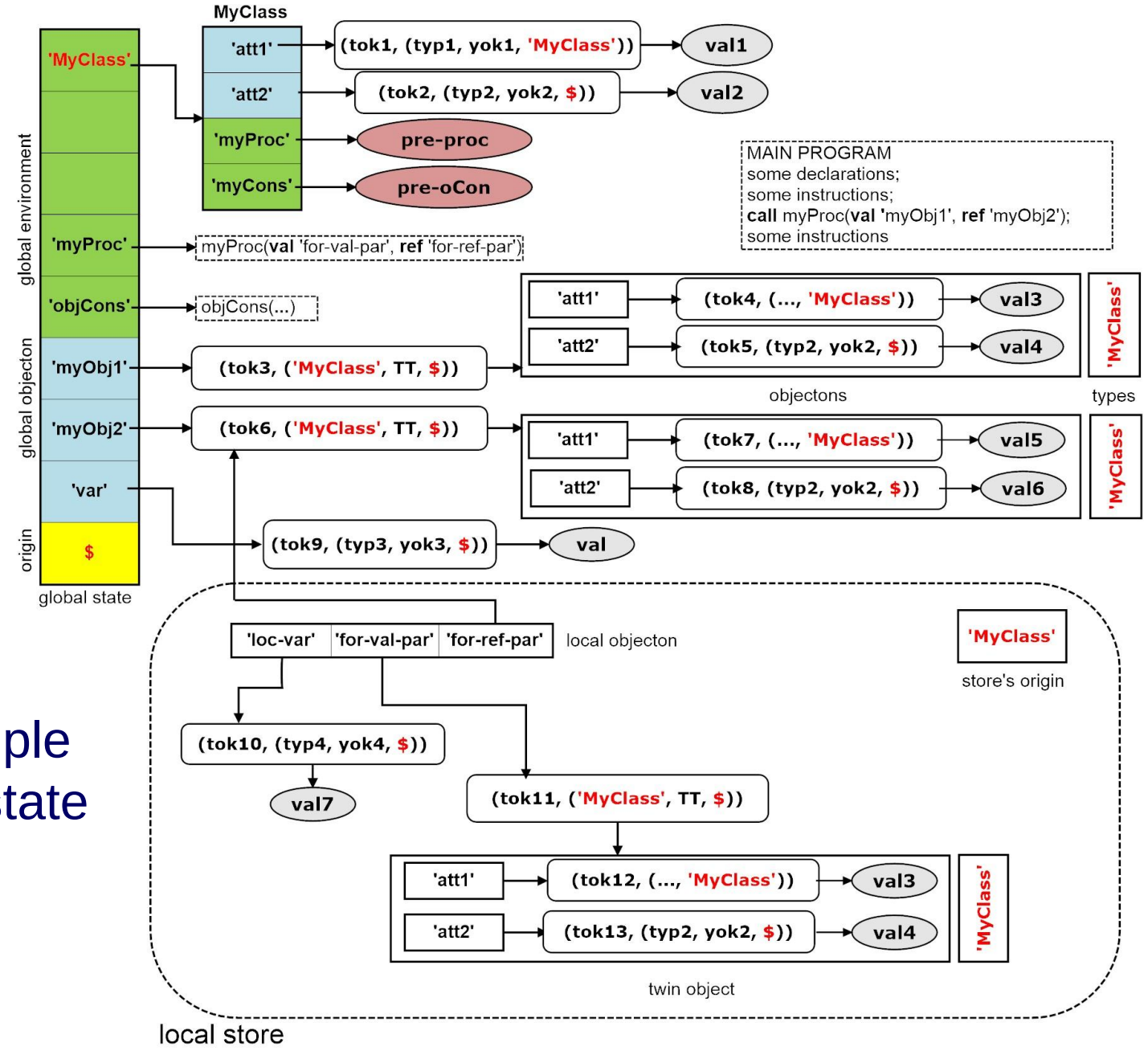
Stores and states

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
obn	: Objecton	= Identifier ⟹ Reference	
dep	: Deposit	= Reference ⟹ Value	

{$} | Identifier

get-tok : SetFeeTok → {tok} | SetFeeTok

An example of a state



Selected carriers of the algebra of denotations

Primitive carriers			
ide : Identifier	= …		identifiers
prs : PriSta	= {‘private’, ‘public’}		privacy statuses indicators
loi : ListOfIde	= Identifier^{c*}		lists of identifiers
cli : ClaInd	= {‘empty-class’} | Identifier		class indicators
Applicative carriers			
ved : ValExpDen	= WfState	→ ValueE	value-expression denotations
yok : YokExpDen	= WfState	→ YokeE	yoke-expression denotations
ted : TypExpDen	= WfState	↦ TypeE	type-expression denotations
red : RefExpDen	= WfState	↦ ReferenceE	reference-expression den.
Imperative carriers			
dcd : DecDen	= WfState	→ WfState	declaration denotations
ind : InsDen	= WfState	→ WfState	instruction denotations
ppd : ProPreDen	= WfState	→ WfState	program-preamble den.
prd : ProDen	= WfState	→ WfState	program denotations

Selected constructors of value-expression denotations

Constructors of fixed-value-expression denotations

ved-bo.boo	:	⟼ ValExpDen	for boo : {tt, ff}
ved-in.int	:	⟼ ValExpDen	for int : Integer
ved-re.rea	:	⟼ ValExpDen	for rea : Real
ved-tx.tex	:	⟼ ValExpDen	for tex : Text

Constructors of selection-expression denotations

ved-variable	: Identifier	⟼ ValExpDen
ved-attribute	: Identifier x Identifier	⟼ ValExpDen

Constructor of functional procedure calls

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

…

Selected constructors of value-expression denotations

Constructors based on typed-data constructors (examples)

ved-divide-re	: ValExpDen x ValExpDen	⟼ ValExpDen
ved-create-li	: ValExpDen	⟼ ValExpDen
ved-get-from-rc	: ValExpDen x Identifier	⟼ ValExpDen
ved-get-from-ar	: ValExpDen x ValExpDen	⟼ ValExpDen

Constructors of boolean-expression denotations

equal	: ValExpDen x ValExpDen	⟼ ValExpDen
less	: ValExpDen x ValExpDen	⟼ ValExpDen
ved-and	: ValExpDen x ValExpDen	⟼ ValExpDen
ved-or	: ValExpDen x ValExpDen	⟼ ValExpDen
ved-not	: ValExpDen	⟼ ValExpDen

Conditional-expression constructor

ved-if	: ValExpDen x ValExpDen x ValExpDen	⟼ ValExpDen

A

Selected definitions of constructors of value-expression denotations

A constant-value expression

ved-int.3 : ⟼ ValExpDen i.e.
ved-int.3 : ⟼ WfState → Value | Error
ved-int.3.().sta =
  is-error.sta ➔ error.sta
  true ➔ (3, 'integer')

A single-variable expression

ved-variable : Identifier ⟼ ValExpDen i.e.
ved-variable : Identifier ⟼ WfState ⟶ ValueE
ved-variable.ide.sta
  is-error.sta ➔ error.sta
  let
    (env, (obn, dep, st-ota, sft, 'OK')) = sta
  obn.ide = ? ➔ 'variable not declared'
  dep.(obn.ide) = ? ➔ 'variable not initialized'
  true ➔ dep.(obn.ide)

A

Selected definitions of constructors of value-expression denotations

An object-attribute expression

ved-attribute : ValExpDen x Identifier ⟼ ValExpDen i.e.
ved-attribute : ValExpDen x Identifier ⟼ WfState ⟶ ValueE
ved-attribute.(ved, at-ide).sta =
  is-error.sta ➔ error.sta
  ved.sta = ? ➔ ?
  ved.sta : Error ➔ ved.sta
  ved.sta /: Object ➔ 'object expected'
  let
    (ob-obn, cl-ide) = ved.sta
  obn.at-ide = ? ➔ 'attribute unknown'
  let
    (tok, (typ, yok, ota)) = obn.ide the reference of ide in obn
    (env, (obn, dep, st-ota, sft, 'OK')) = sta
    dep.(obn.at-ide) = ? ➔ 'attribute not initialized'
    ota ≠ $ **and** ota ≠ st-ota ➔ 'attribute not visible'
  true ➔ dep.(ob-obn.at-ide)

A

Selected definitions of constructors of value-expression denotations

Division of real numbers

ved-divide-re: ValExpDen x ValExpDen ⟼ ValExpDen i.e.
ved-divide-re: ValExpDen x ValExpDen ⟼ WfState ⟶ ValueE
ved-divide-re.(ved-1, ved-2).sta =
  is-error.sta ➔ error.sta
  ved-i.sta = ? ➔ ? for i = 1,2
  ved-i.sta : Error ➔ ved-i.sta for i = 1,2
  let
    val-i = ved-i.sta for i = 1,2
    val = td-divide-re.(val-1, val-2)
  true ➔ val

Selected definitions of constructors of value-expression denotations

The equality of values (a boolean expression)

equal : ValExpDen x ValExpDen ⟼ ValExpDen i.e.
equal : ValExpDen x ValExpDen ⟼ WfState ⟶ Value | Error
equal.(ved-1, ved-2).sta =
  is-error.sta ➔ error.sta
  ved-i.sta = ? ➔ ? for i = 1,2
  ved-i.sta : Error ➔ ved-i.sta for i = 1,2
  let
    (cor-i, typ-i) = ved-i.sta for i = 1,2
  typ-1 ≠ typ-2 ➔ 'compared values must be of the same type'
  not comparable.typ-1 ➔ 'values not comparable'
  cor-1 = cor-2 ➔ (tt, 'boolean')
  true ➔ (ff, 'boolean')

Selected constructors of type-expression denotations

ted-create-bo	:	⟼ TypExpDen
ted-create-in	:	⟼ TypExpDen
ted-create-re	:	⟼ TypExpDen
ted-create-tx	:	⟼ TypExpDen
ted-create-ot	: Identifier	⟼ TypExpDen
ted-constant	: Identifier x Identifier	⟼ TypExpDen
ted-create-li	: TypExpDen	⟼ TypExpDen
ted-create-ar	: TypExpDen	⟼ TypExpDen
ted-create-re	: Identifier x TypExpDen	⟼ TypExpDen
ted-put-to-re	: Identifier x TypExpDen x TypExpDen	⟼ TypExpDen

Selected definitions of constructors of type-expression denotations

Boolean-type expression denotation

ted-create-bo.().sta =
  is-error.sta ➔ error.sta
  true ➔ ‘boolean’

Object-type expression denotation

ted-create-ot.ide.sta =
  is-error.sta ➔ error.sta
  true ➔ ide

One-attribute-record-type expression denotation

ted-create-re.(ide, ted).sta =
  is-error.sta ➔ error.sta
  ted.sta : Error ➔ ted.sta
  let
    typ = ted.sta
  true ➔ ty-create-re.(ide, typ): (R’, [ide/typ])

A

Selected definitions of constructors of type-expression denotations

Type-constant expression denotation

ted-constant.(ide-cl, ide-ty).sta =
  is-error.sta ➔ error.sta
  let
    ((cle, mee, cov), sto) = sta
  cle. ide-cl = ? ➔ ‘class unknown’
  let
    (ide-cl, tye, mee, obn) = cle.ide-cl
  tye.ide-ty = ? ➔ ‘type unknown’
  tye.ide-ty = Θ ➔ ‘type not concretized’
  true ➔ tye.ide-ty

well-formedness of sta

Two constructors of reference-expression denotations

A variable reference

ref-variable : Identifier ⟼ RefExpDen

An attribute reference

ref-attribute : ValExpDen x Identifier ⟼ RefExpDen

The definition of a constructor of reference-expression denotations

ref-variable : Identifier ⟼ RefExpDen i.e.
ref-variable : Identifier ⟼ WfState ⟼ ReferenceE
ref-variable.ide.sta =
  is-error.sta ➔ error.sta
  let
    (env, (obn, dep, st-ota, sft, 'OK')) = sta
  obn.ide = ? ➔ 'variable not declared'
  true ➔ obn.ide

The definition of a constructor of reference-expression denotations

ref-attribute : ValExpDen x Identifier ⟼ RefExpDen i.e.
ref-attribute : ValExpDen x Identifier ⟼ WfState ⟼ ReferenceE
ref-attribute.(ved, at-ide).sta =
  is-error.sta ➔ error.sta
  ved.sta = ? ➔ ?
  ved.sta : Error ➔ ved.sta
  ved.sta /: Object ➔ 'object expected'
  let
    (va-obn, va-ide) = ved.sta
    (env, (obn, dep, st-ota, sft, 'OK')) = sta
  va-obn.at-ide = ? ➔ 'attribute not declared'
  let
    (tok, (typ, yok, at-ota)) = va-obn.at-ide
  at-ota ≠ $ **and** at-ota ≠ st-ota ➔ 'attribute not visible'
  true ➔ va-obn.at-ide


Thank you for
your attention