A Denotational Engineering of Programming Languages

...

Part 5: Lingua-1 – Instructions and declarations
(Section 5 of the book)

Andrzej Jacek Blikle

March 23rd, 2021

The denotational domains of Lingua-1

Lingua-1: inherited domains (applicative denotations)

ide	: Identifier	= …	
ded	: DatExpDen	= State → ValueE	data-expression denotations
tra	: TraExpDen	= Transfer	transfer-expression denotations
bed	: BodExpDen	= State ↦ BodyE	body-expression denotations
yok	: YokExpDen	= Yoke	yoke-expression denotations
ted	: TypExpDen	= State ↦ TypeE	type-expression denotations

Lingua-1: new domains (imperative denotations)

dde	: DecDen	= State ↦ State	declaration denotations
ind	: InsDen	= State → State	instruction denotations
prd	: ProDen	= State → State	program denotations

Lingua-1 emerges from Lingua-A by adding the following components to the algebras of denotations and syntax:

- new carriers (instructions, declarations, programs),
- new constructors for new carriers.

Everything else remains unchanged!

A

Conservative denotations

DEF an imperative denotation dim is called conservative if

1. if error.sta ≠ 'OK' then dim.sta = sta,
2. if dim.sta = ! then bodies in dim.sta are identical with bodies in sta

error-state transparency

In Lingua all reachable imperative denotations, which do not involve error handling, will be conservative.

sumienny

DEF a constructor of imperative denotations is called decent if it preserves conservativeness.

A reminder:

rzetelny

DEF A constructor of data-expression denotations is called diligent if it transforms transparent denotations into transparent denotations.

A

Programs

create-program : DecDen x InsDen $\mapsto$ ProDen
create-program.(dde, ins) = dde ● ins

Declarations may be:

1. Atomic
    a. data-variable declaration,
    b. type-constant declaration,
    c. trivial (do nothing)
2. Structured, i.e. composed by means:
    a. sequential composition

Trivial declarations and instructions are used in procedure declarations.

Instructions may be:

1. Atomic:
    a. assignment,
    b. yoke replacement,
    c. trivial (do nothing)
2. Structured, i.e. composed by means of:
    a. sequential composition,
    b. if-then-else-fi
    c. while-do-od
    d. error handling

An auxiliary metapredicate

declared : Identifier ⟼ State ⟼ BooleanE

identifier declared

declared.ide.((tye, pre), (vat, err)) =
  tye.ide = ! **or** pre.ide = ! **or** vat.ide = ! ➔ tt
  true ➔ ff

An engineering decision protecting identifiers against multiple declarations.

Declarations of data variables

declare-dat-var : Identifier x TypExpDen $\mapsto$ DecDen

declare-dat-var.(ide, ted).sta =
  is-error.sta ➔ sta
  declared.ide.sta ➔ sta ◄ 'variable-declared'
  let
    (env, (vat, 'OK')) = sta
    typ = ted.sta
  typ : Error ➔ sta ◄ typ
  true ➔ (env, (vat[ide/(Ω, typ)], 'OK'))

Only valuation is modified

Declarations of body constants

declare-bod-con : Identifier x BodExpDen ⟼ DecDen

declare-bod-con.(ide, bed).sta =
  is-error.sta ➔ sta
  declared.ide.sta ➔ sta ◄ ‘identifier-not-free’
  let
    bod = bed.sta
    ((tye, pre), sto) = sta
  bod : Error ➔ sta ◄ bod
  true ➔ ((tye[ide/bod], pre), sto)

Protects against a redeclaration of a body constant

Only type environment is modified

Declarations of type constants

declare-typ-con : Identifier x TypExpDen ⟼ DecDen

declare-typ-con.(ide, ted).sta =
  is-error.sta ➔ sta
  declared.ide.sta ➔ sta ◄ ‘identifier-not-free’
  let
    typ = ted.sta
    ((tye, pre), sto) = sta
  typ : Error ➔ sta ◄ typ
  true ➔ ((tye[ide/typ], pre), sto)

Structured and trivial declarations

create-trivial-dec : $\mapsto$ DecDen
create-trivial-dec.().sta = sta

sequence-dec : DecDen x DecDen $\mapsto$ DecDen
sequence-dec.(dde-1, dde-2) = dde-1 ● dde-2

Assignment instructions

assign : Identifier x DatExpDen ⟼ InsDen

assign.(ide, ded).sta =
  is-error.sta ➔ sta
  let
    ((tye, pre), (vat, 'OK')) = sta
  vat.ide = ? ➔ sta ◄ 'identifier-not-declared'
  ded.sta = ? ➔ ?
  ded.sta : Error ➔ sta ◄ ded.sta
  let
    (dat-f, (bod-f, yok-f)) = vat.ide
    (dat-n, (bod-n, yok-n)) = ded.sta
    com = yok-f.(dat-n, bod-n)
  com : Error ➔ sta ◄ com
  bod-n ≠ bod-f ➔ sta ◄ 'inconsistent-bodies'
  com ≠ (tt, ('Boolean')) ➔ sta ◄ 'yoke-not-satisfied'
  let
    val-n = (dat-n, (bod-f, yok-f))
  true ➔ ((tye, pre), (vat[ide/val-n], 'OK'))

Although yok-n is neglected, it may be used in checking the satisfaction of yok-f by proving

yok-n **implies** yok-f

an infinite execution

f – former
n – new

Yoke-replacement instructions

Replaces a yoke in a type assigned to a data variable

replace-yo : Identifier x YokExpDen ⟼ InsDen

The unique tool in Linguga to change the type (yoke) of a variable.

replace-yo.(ide, yok-n).sta = n - new
is-error.sta ➔ sta
let
((tye, pre), (vat, 'OK')) = sta
vat.ide = ? ➔ sta ◄ 'identifier-not-declared'
let
((com, yok-f) = vat.ide f - former
yok-n.com ≠ (tt, ('Boolean') ➔ sta ◄ 'yoke-not-satisfied'
let
val-n = (com, yok-n)
true ➔ ((tye, pre), vat[ide/val-n], 'OK')

Applications in Lingua-SQL

Trivial instruction

create-trivial-ins : $\mapsto$ InsDen

create-trivial-ins.().sta = sta

Trivial instruction will be used in in functional procedures.

Sequencing and branching instructions

sequence-ins : InsDen x InsDen ⟼ InsDen
sequence-ins.(ind-1,ind-2) = ind-1 ● ind-2

if : DatExpDen x InsDen x InsDen ⟼ InsDen

if.(ded, ind-1, ind-2).sta =
  is-error.sta ➔ sta
  ded.sta = ? ➔ ?
  ded.sta : Error ➔ sta ◄ ded.sta
  let
    (dat, (bod, yok)) = ded.sta
  bod ≠ (‘Boolean’) ➔ sta ◄ ‘Boolean-expected’
  dat = tt ➔ ind-1.sta
  true ➔ ind-2.sta

both ind-i.sta may be undefined or may generate errors

While instructions

while : DatExpDen x InsDen ⟼ InsDen

while.(ded, ind).sta =
  is-error.sta ➔ sta
  ded.sta = ? ➔ ?
  ded.sta : Error ➔ sta ◄ ded.sta
  let
    (dat, (bod, yok)) = ded.sta
  bod ≠ (‘Boolean’) ➔ sta ◄ ‘Boolean-expected’
  dat = ff ➔ sta
  true ➔ (ind • [while.(ded, ind)]).sta

This constructor has a recursive definition which means that for any ded and ind the denotation

while.(ded, ind) : State → State

is defined by a fixed-point equations

Error-handling instructions

Just an example showing the expressiveness of error handling in our model

if-error : DatExpDen x InsDen → InsDen
if-error.(ded, ind).sta =
  let is-error.sta ➔ sta
    (env, (vat, err)) = sta
  err = 'OK' ➔ sta
  let
    sta-1 = (env, (vat, 'OK'))
  ded.sta-1 = ? ➔ ?
  let
    val = ded.sta-1
  val : Error ➔ sta ◄ val © ‘error-handling-not-executed’
  let
    (dat, (bod, yok)) = val
  bod ≠ (‘word’) ➔ ‘sta ◄ ‘word-expected’ © ‘error-handling-not-executed’
  dat ≠ err ➔ sta ◄ dat © ‘error-handling-not-executed’
  ind.sta-1 = ? ➔ ?
  let
    sta-2 = ind.sta-1
  is-error.sta-2 ➔ sta ◄ error.sta-2 © ‘error-handling-not-executed’
  true ➔ sta-2

> this expression should evaluate to the handled error

> to execute the error-handling instruction we have to free the current state from the error

> error to be handled

> current error must be equal to the handled error

A

Concrete syntax of Lingua-1
(Imperative part)

prg : Program =
(Declaration **;** Instruction)

dec : Declaration =
let Identifier **be** TypExp **tel** | variable declaration
set-body Identifier **as** BodExp **tes** | body-constant declaration
set-type Identifier **as** TypExp **tes** | type-constant declarations
(Declaration **;** Declaration) |
skip-d

ins : Instruction =
Identifier **:=** DatExp |
yoke Identifier:= YokExp **ekoy** |
if DatExp **then** Instruction **else** Instruction **fi** |
if-error DatExp **then** Instruction **fi** |
while DatExp **do** Instruction **od** |
(Instruction **;** Instruction) |
skip-i

Colloquial syntax of Lingua-1
(Imperative part; examples)

The omission of parentheses in:

- `dec ; ins`
- `dec-1 ; dec-2`
- `ins-1 ; ins-2`

instead of

let x **be** number **tel;**

let y **be** number **tel;**

let z **be** number **tel**

we write

let x, y, z **be** number **tel**

Some more examples in the book.

The semantics of Lingua-1
(the imperative layer; implementor's version)

Three semantic functions:

Sde : Declaration $\mapsto$ DecDen
Sin : Instruction $\mapsto$ InsDen
Spr : Program $\mapsto$ ProDen

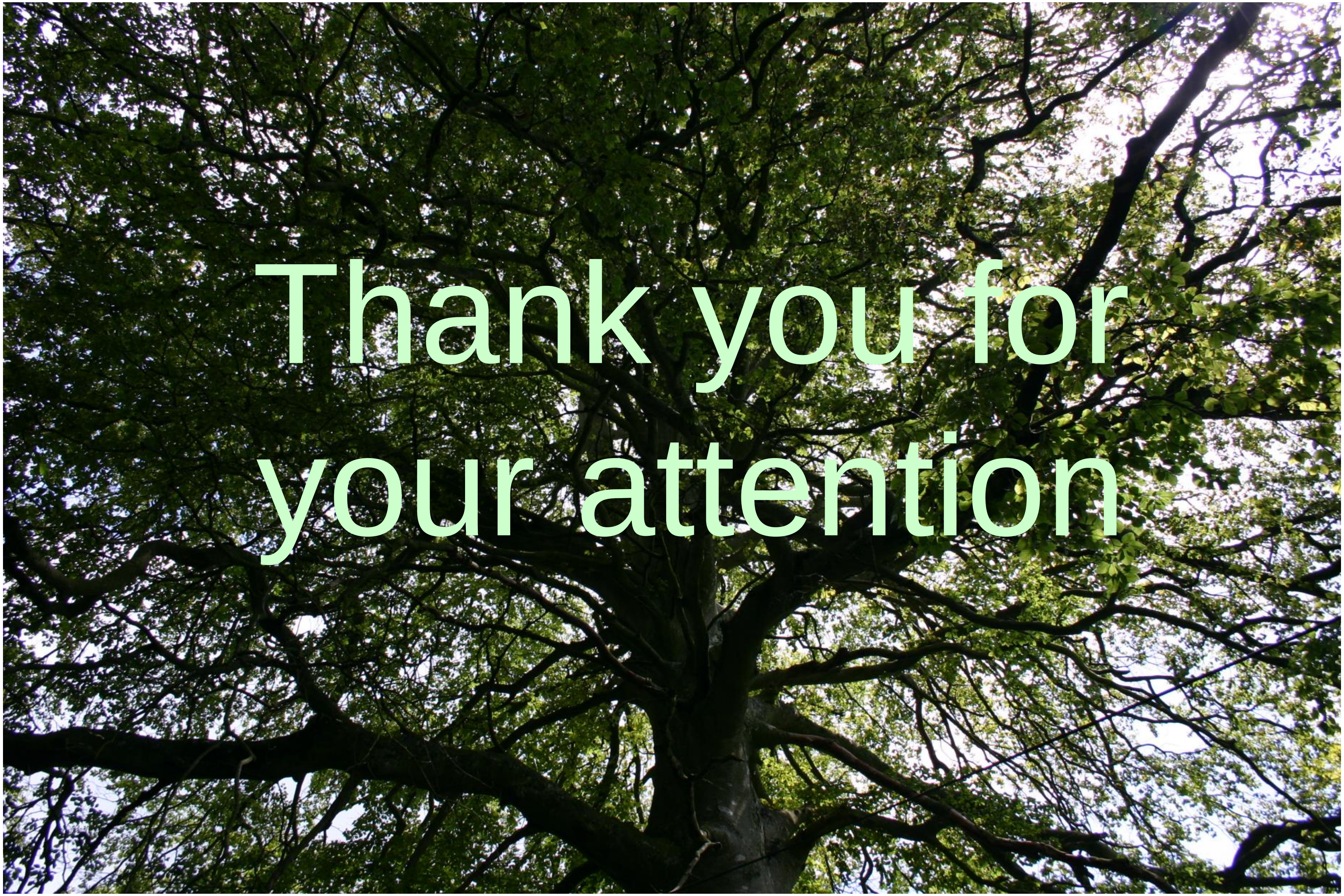
Examples of definitions:

```
Sde.[let ide be tex]           = data-variable.(Sid.[ide], Sty.[tex])
Sde.[set ide as tex]           = declare-typ-con.(Sid.[ide], Sty.[tex])
Sde.[(dec-1; dec-2)]           = sequence-vde.(Sde.[dec-1], Sde.[dec-2])

Sin.[ide := dae]               = assign.(Sid.[ide], Sw.[dae])
Sin.[while dae do ins od]      = while.(Sw.[dae], Si.[ins])
Sin.[(ins-1;ins-2)]            = sequence-ins.(Si.[ins-1], Si.[ins-2])

Spr.[(dec ; ins)]              = create-program.(Sde.[dec], Sin.[ins])
```

The semantics of Lingua-1
(the imperative layer; programmer's version)

Sin.[**while** dae **do** ins **od**] = while.(Sw.[dae], Si.[ins])

Sin.[**while** dae **do** ins **od**].sta =
  is-error.sta ➔ sta
  Sde.[dae].sta = ? ➔ ?
  Sde.[dae].sta : Error ➔ sta ◄ Sde.[dae].sta
  let
    (dat, bod) = Sde.[dae].sta
  sort.bod ≠ (‘Boolean’) ➔ sta ◄ ‘Boolean-expected’
  dat = ff ➔ sta
  true ➔ Sin.[**while** dae **do** ins **od**].(Sin.[ins].sta)


Thank you for
your attention