A Denotational Engineering of Programming Languages

…

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

Andrzej Jacek Blikle

April 24th, 2020

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
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

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

Instructions may be:

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

Trivial declarations and instructions are used in procedure declarations.

An auxiliary metapredicate

declared : Identifier $\mapsto$ State $\mapsto$ BooleanE

identifier declared

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

An engineering decision protecting an identifier against multiple declaration.

Declarations of data variables

declare-dat-var : Identifier x TypExpDen ⟼ DecDen

declare-dat-var.(ide, ted).sta =

is-error.sta ➔ sta

declared.ide.sta ➔ sta ◄ 'variable-declared'

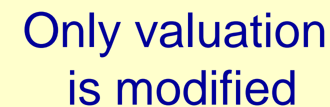
let

(env, (vat, 'OK')) = sta

typ = ted.sta

typ : Error ➔ sta ◄ typ

true ➔ (env, (vat[ide/(Ω, typ)], 'OK'))



Declarations of body constants

declare-bod-con : Identifier x BodExpDen ⟼ DecDen

declare-bod-con.(ide, ted).sta =
  is-error.sta ➔ sta
  declared.ide.sta ➔ sta ◄ ‘identifier-not-free’
  let
    bod = ted.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

Conservative denotations

DEF an imperative denotation dim is called conservative if

1. if error.sta ≠ 'OK' then dim.sta = sta, and
2. if error.sta = 'OK' and 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

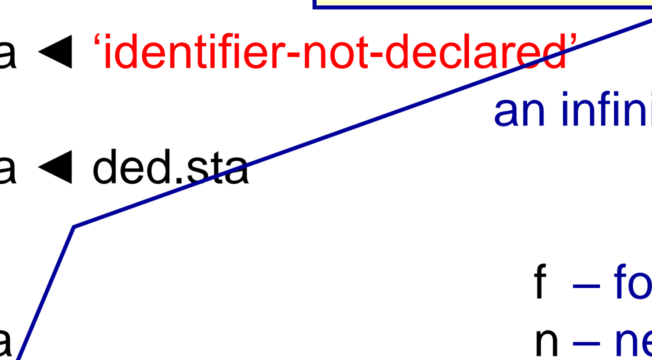
DEF A constructor of data-expression denotations is called diligent if it transforms transparent denotations into transparent denotations.

A

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-n), 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 = ? ➔ 'identifier-not-declared'
  let
    ((com, yok-f) = vat.ide f - former
  yok-n.com ≠ (tt, ('Boolean') ➔ '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) = ded.sta
  sort.bod ≠ (‘Boolean’) ➔ sta ◄ ‘Boolean-expected’
  dat = tt ➔ ind-1.sta
  true ➔ ind-2.sta

both ind-i.sta may be undefined

While instructions

while : DatExpDen x InsDen $\mapsto$ InsDen

while.(ded, ind).sta =
  is-error.sta ➔ sta
  ded.sta = ? ➔ ?
  ded.sta : Error ➔ sta ◄ ded.sta
  let
    (dat, bod) = ded.sta
  sort.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 $\rightarrow$ 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 =
  not is-error.sta ➔ sta
  let
    (env, (vat, err)) = sta
    sta-1 = sta ◄ ‘OK’
  ded.sta-1 = ? ➔ ?
  let
    val = ded.sta-1
  val : Error ➔ sta ◄ val © ‘error-handling-not-executed’
  sort.val ≠ (‘word’) ➔ ‘sta ◄ ‘word-expected’ © ‘error-handling-not-executed’
let
  (wor, (‘word’), yok) = val
  wor ≠ err ➔ sta
  ind.sta-1 = ? ➔ ?
  let
    sta-2 = ind.sta-1
  is-error.sta-2 ➔ sta ◄ error.sta-2 ©‘error-handling-not-executed’
  true ➔ sta-2 ◄ err ©‘error-handling-executed’

this expression should evaluate to the handled error

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

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 const. declaration
set-type Identifier **as** TypExp **tes** |	type const. 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.

Thank you for
your attention