A Denotational Engineering of Programming Languages

…

Part 4: Lingua-A – Expressions

(Sections 4.4 – 4.9 of the book)

Andrzej Jacek Blikle

March 18th, 2021

Values and states

Preliminaries

val : Value = {(dat, typ) | dat : CLAN-Ty.typ}

val : PsValue = {(Ω, typ) | typ : Type }

A pseudo-data

val = (dat, typ) = (dat, (bod, yok)) = ((dat, bod), yok) = (com, yok)

type com

sta	: State	= Env x Store	
env	: Env	= TypEnv x ProEnv	environments
sto	: Store	= Valuation x (Error | {'OK'})	
vat	: Valuation	= Identifier ⟹ (Value | PsValue)	
tye	: TypEnv	= Identifier ⟹ (Type | Body)	type environments
pre	: ProEnv	= Identifier ⟹ Procedure	procedure environments

Variable identifiers

Procedure names

Type constant identifiers

sta = ((tye, pre), (vat, err)) err : Error | {'OK'}

this structure is not accidental

States

Auxiliary functions

error : State $\mapsto$ Error | {'OK'} — error-selection operator for states
error.(env, (vat, err)) = err

is-error : State $\mapsto$ Boolean — error-detection predicate for states
is-error.sta =
  error.sta ≠ 'OK' ➔ tt
  true ➔ ff

is-error : Store $\mapsto$ Boolean — error-detection predicate for stores
is-error.(vat, err) =
  err ≠ 'OK' ➔ tt
  true ➔ ff

◄ : State x Error $\mapsto$ State — error-insertion operator for states
  (env, (vat, err)) ◄ err-1 =
    (env, (vat, err-1))

Carriers of the algebra of expression denotations

Carriers of <u>AlgExpDen</u>:

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

Here no functions on states since transfers and yokes are not storable; (an engineering decision).

Data-expression denotations

ded : DatExpDen = State → ValueE

Partial functions due to functional procedures!

Transparent denotations

ded.(env, (vat, err)) = err whenever err ≠ 'OK'.

DEF A constructor of denotations is called diligent if it builds transparent denotations.

rzetelny

All constructors of expression denotations will be diligent.

All reachable expression denotations will be transparent.

Four groups of constructors of expression denotations

1) one constructor of variables,
2) constructors derived from (non-Boolean) value constructors (one for each),
3) Boolean constructors,
4) one constructor for conditional expressions.

Data-expression denotations

Constructors derived from value constructors

A constructor of variables

ded-variable : Identifier $\mapsto$ DatExpDen

A metaconstructor (creates constructors of data-expression denotations)

Cdd : Constructors of Values $\mapsto$ Constructors of DatExpDen

Zero-argument constructors

Cdd.[va-create-id.ide] : $\mapsto$ Identifier for ide : Identifier
Cdd.[va-create-bo.boo] : $\mapsto$ DatExpDen for boo : Boolean
Cdd.[va-create-in.int] : $\mapsto$ DatExpDen for int : IntegerS
Cdd.[va-create-wo.wor] : $\mapsto$ DatExpDen for wor : WordS

Comparison constructors

Cdd.[ded-equal] : DatExpDen x DatExpDen $\mapsto$ DatExpDen
Cdd.[ded-less] : DatExpDen x DatExpDen $\mapsto$ DatExpDen

Arithmetic constructors

Cdd.[va-add-in] : DatExpDen x DatExpDen $\mapsto$ DatExpDen
Cdd.[va-divide-in] : DatExpDen x DatExpDen $\mapsto$ DatExpDen

etc. for integers and reals

Word constructors

Cdd.[va-glue] : DatExpDen x DatExpDen $\mapsto$ DatExpDen

Data-expression denotations

Constructors derived from value constructors

List constructors

Cdd.[va-create-li]	: DatExpDen	$\mapsto$	DatExpDen
Cdd.[va-push]	: DatExpDen x DatExpDen	$\mapsto$	DatExpDen
Cdd.[va-top]	: DatExpDen	$\mapsto$	DatExpDen
Cdd.[va-pop]	: DatExpDen	$\mapsto$	DatExpDen

Array constructors

Cdd.[va-create-ar]	: DatExpDen	$\mapsto$	DatExpDen
Cdd.[va-put-to-ar]	: DatExpDen x DatExpDen	$\mapsto$	DatExpDen
Cdd.[va-change-in-ar]	: DatExpDen x DatExpDen x DatExpDen	$\mapsto$	DatExpDen
Cdd.[va-get-from-ar]	: DatExpDen x DatExpDen	$\mapsto$	DatExpDen

Record constructors

Cdd.[va-create-re]	: Identifier x DatExpDen	$\mapsto$	DatExpDen
Cdd.[va-put-to-re]	: DatExpDen x DatExpDen x Identifier	$\mapsto$	DatExpDen
Cdd.[va-get-from-re]	: DatExpDen x Identifier	$\mapsto$	DatExpDen
Cdd.[va-change-in-re]	: DatExpDen x Identifier x DatExpDen	$\mapsto$	DatExpDen

Data-expression denotations

Constructors

Boolean constructors

ded-and : DatExpDen x DatExpDen $\longmapsto$ DatExpDen
ded-or : DatExpDen x DatExpDen $\longmapsto$ DatExpDen
ded-not : DatExpDen $\longmapsto$ DatExpDen

Conditional-expression constructor

when : DatExpDen x DatExpDen x DatExpDen $\longmapsto$ DatExpDen

Data-expression denotations

Data-variable constructor

dat-variable : Identifier $\mapsto$ DatExpDen

dat-variable.ide.sta =
  is-error.sta ➔ error.sta
  let
    (env, (vat, 'OK')) = sta
  vat.ide = ? ➔ 'undeclared-variable'
  let
    (dat, typ) = vat.ide
  dat = Ω ➔ 'uninitialized-variable'
  true ➔ (dat, typ)

diligence of dat-variable

rzetelność

We eliminate a possible pseudo values from further computations which means that they are never "sent" to a composite constructor as an arguments.

Data-expression denotations

Constructors derived from composite-constructors

vco : ValIde-1 x … x ValIde-n ⟼ ValueE
Cdd[vco] : DatExpDenIde-1 x … x DatExpDenIde-n ⟼ DatExpDen
Cdd – metaconstructor of constructors of data-expression denotations

Cdd[vco].(arg-1,…,arg-n).sta = n ≥ 0
  is-error.sta ➔ error.sta
  for i = 1;n
  do
    (arg-i !: Identifier) **and** arg-i.sta = ? ➔ ?
      let
        val-i =
          arg-i : Identifier ➔ arg-i
          true ➔ arg-i.sta
  od
  true ➔ vco.(val-1,…,val-n)

diligence of Cdd[cva]

Performs further error-analysis

This scheme <u>is not applicable</u> to boolean constructors, since they are not transparent.

Data-expression denotations

An example of a derived constructor

va-divide-in : ValueE x ValueE ⟼ ValueE
Cdd[va-divide-re] : DatExpDen x DatExpDen ⟼ DatExpDen

Cdd[va-divide-re].[ded-1, ded-2].sta =
  is-error.sta ➔ error.sta
  ded-i.sta = ? ➔ ? for i = 1,2
    let
      val-i = ded-i.sta for i = 1,2
   true ➔ va-divide-re.(val-1, val-2)

Here all further error analysis

Data-expression denotations

Boolean constructors (McCarthy's laziness)

ded-and : DatExpDen x DatExpDen $\mapsto$ DatExpDen

ded-and.(ded-1, ded-2).sta =
  is-error.sta ➔ error.sta
  ded-1.sta = ? ➔ ?
  let
    val-1 = ded-1.sta
    val-1 : Error ➔ val-1
  let
    (dat-1, bod-1, yok-1) = val-1
  dat-1 = ff ➔ ff
  bod-1 ≠ ('Boolean') ➔ 'Boolean-expected'
    ded-2.sta = ? ➔ ?
  let
    val-2 = ded-2.sta
  val-2 : Error ➔ val-2
  let
    (dat-2, bod-2, yok-2) = val-2
  bod-2 ≠ ('Boolean') ➔ 'Boolean-expected'
  true ➔ (dat-2, ('Boolean'), TT)

ded-or and ded-not are defined in an analogous way

Here we possibly avoid an infinite evaluation or an error from ded-2

Data-expression denotations

A conditional expression

when : DatExpDen x DatExpDen x DatExpDen ⟼ DatExpDen

when.(ded-1, ded-2, ded-3).sta =
  is-error.sta → error.sta
  ded-1.sta = ? → ?
  let
    val-1 = ded-1.sta
  val-1 : Error → val-1
  let
    (dat-1, bod-1, yok-1) = val-1
  bod-1 ≠ (‘Boolean’) → ‘Boolean-expected’
  dat-1 = tt → ded-2.sta
  dat-1 = ff → ded-3.sta

Lazy evaluation

The type of the result is not fixed!

Future syntax:

```
if x > 0 then x+2 else ‘abcd’ fi
if x > 0 then sqrt(x) else sqrt(-x) fi
```

Transparency would cause a problem for all x ≠ 0.

Body-expression denotations

Constructors

bod-constant : Identifier $\mapsto$ BodExpDen

bod-constant.ide.sta =
  is-error.sta ➔ error.sta
  let
    ((tye, pre), sto) = sta
  tye.ide = ? ➔ ‘body-constant-undefined’
  not tye.ide : Body ➔ 'body-expected'
  true ➔ tye.ide

Body constants retrieve bodies from type environments.

Body, once assigned to a body constant is never changed.

Metaconstructor of constructors of body-expression denotations

Cbd : Constructors of bodies $\mapsto$ Constructors of body-expression denotations

Constructors derived from body constructors

Cbd.[bo-create-bo] : $\mapsto$ BodExpDen
Cbd.[bo-create-nu] : $\mapsto$ BodExpDen
Cbd.[bo-create-wo] : $\mapsto$ BodExpDen
Cbd.[bo-create-ar] : BodExpDen $\mapsto$ BodExpDen
Cbd.[bo-create-re] : BodExpDen x Identifier $\mapsto$ BodExpDen
Cbd.[bo-put-to-re] : BodExpDen x Identifier x BodExpDen $\mapsto$ BodExpDen

Body-expression denotations

Constructors derived from body constructors

bco : BodIde-1 x … x BodIde-n ⟼ BodyE

Cbd.[bco] : BodExpDenIde-1 x … x BodExpDenIde-n ⟼ BodExpDen

Cbd – metaconstructor of body-expression denotations

```
Cbd.[bco].(arg-1,…,arg-n).sta =
  is-error.sta  ➔ error.sta
  let
    bod-i =     for i = 1;n
      arg-i : Identifier ➔ arg-i
      true               ➔ arg-i.sta
  true          ➔bco.(bod-1,…,bod-n)
```

takes care for errors

Body-expression denotations

Two examples of derived constructors

Cbd.[bo-create-in] : $\mapsto$ BodExpDen

Cbd.[bo-create-in].().sta =
is-error.sta ➔ error.sta
true ➔ bo-create-in.()

Cbd.[bo-put-to-re] : BodExpDen x Identifier x BodExpDen $\mapsto$ BodExpDen

Cbd.[bo-put-to-re].(bed-e, ide, bed-r.).sta = e for "element", r for "record"
is-error.sta ➔ error.sta
let
bod-i = bed-i.sta for i = e, r
true ➔ bo-put-to-re.(bod-e, ide, bod-r)

takes care for errors

Type-expression denotations

Two constructors only

typ-constant : Identifier ⟼ TypExpDen

typ-constant.ide.sta =
  is-error.sta ➔ error.sta
  let
    ((tye, pre), sto) = sta
  tye.ide = ? ➔ 'type-constant-undefined'
  tye.ide : Body ➔ 'type-expected'
  true ➔ tye.ide

create-ty : BodExpDen x YokExpDen ⟼ TypExpDen
create-ty.(bed, yok).sta =
  is-error.sta ➔ error.sta
  let
    bod = bed.sta
  bod : Error ➔ bod
  true ➔ (bod, yok)

The mechanisms of creating values, and assigning them to variables, will raise error messages in cases of empty types.

A type expression may be either an identifier or a type-creating expression.

Abstract syntax of Lingua-A
denotational and syntactic domains

The signature of AlgExpDen → The algebra AlgExpA → Grammar of abstract syntax

A – stands for "abstract"

denotations	**syntaxes**	**description**
Identifier	Identifier	identifiers
DatExpDen	DatExpA	abstract data expressions
BodExpDen	BodExpA	abstract body expressions
TraExpDen	TraExpA	abstract transfer expressions
YokExpDen	YokExpA	abstract yoke expressions
TypExpDen	TypExpA	abstract type expressions

For every carrier of AlgExpA one grammatical equation.
For every constructor of AlgExpA one component of an equation

Abstract syntax of Lingua-A

examples of syntactic clauses (equational grammar)

green – syntactic elements (words)
black – metavariables

```
dae : DatExpA =
  ...
  dat-variable.(Identifier)                          |
  ...
  and-ded.(DatExpA , DatExpA)                        |
  or-ded.(DatExpA , DatExpA)                         |   Boolean expressions
  Cdd[va-equal].(DatExpA , DatExpA)                  |
  ...
  Cdd[va-glue].(DatExpA, DatExpA)                    |   word expression
  ...
  Cdd[va-create-ar].(DatExpA)                        |
  Cdd[va-put-to-ar].(DatExpA, DatExp)                |   array expressions
  Cdd[va-get-from-ar].(DatExpA, DatExpA)             |
     ...

tex : TypExpA =
  type-constant.(Identifier)                         |   type expressions
  create-type.(BodExpA, YokExpA)                     |
```

Concrete syntax of Lingua-A
examples of syntactic clauses (equational grammar)

```
dae : DatExp =
  …
  Identifier                                  |
  …
  (DatExp and DatExp)                         |
  (DatExp or DatExp)                          |
  (DatExp = DatExp)                           |
  …
  (DatExp © DatExp)                           |

  array DatExp ee                             |
  put-to-arr DatExp new DatExp ee             |
  array  DatExp at DatExp ee                  |
    …

tex : TypExp =
  Identifier                                  |
  type BodExp with YokExp ee
```

variables

Boolean expressions

word expressions

array expressions

type expressions

Colloquial syntax of Lingua-A
examples of rules

instead of	we write
(x + (y + z))	x + y + z : left association
(x + (y * z))	x + y * z : priority of * over +

put-to-arr
  put-to-arr
    array x **ee**
  new x+y **ee**
new 3*y **ee**
	array [x, x+y, 3*y]

Syntactic sugar: spaces, tabulators, carriage returns, boldface and underlining do not affect denotations of syntax.

Semantics of Lingua-A

Cs : AlgExp ⟼ AlgExpDen

A homomorhism with six components:

Sid : Identifier ⟼ Identifier — identity mapping
Sde : DatExp ⟼ DatExpDen
Sbe : BodExp ⟼ BodExpDen
Stre : TraExp ⟼ TraExpDen
Syoe : YokExp ⟼ YokExpDen
Ste : TypExp ⟼ TypExpDen

Examples

Sde : DatExp ⟼ DatExpDen i.e.
Sde : DatExp ⟼ State → ValueE

Sde.[`true`] = Cdd[co-create-bo.tt].() — algebraic form

Sde.[`true`].sta = — direct form
is-error.sta ➔ error.sta
true ➔ (tt, (‘Boolean’), TT)

Semantics of Lingua-A
Implementor's perspective

Sde : DatExp ⟼ DatExpDen   i.e.
Sde : DatExp ⟼ State → ValueE

Algebraic form

Sde.[`(dae-1 / dae-2)`] =
  Cdd[va-divide-re].(Sde.[`dae-1`], Sde.[`dae-2`])

value constructor
(slide 33 part 3)

metaconstructor
(slide 6)

Semantics of Lingua-A
User's perspective

Sde : DatExp $\mapsto$ DatExpDen i.e.
Sde : DatExp $\mapsto$ State $\rightarrow$ ValueE

Direct form – user oriented (cf. part 3 slide 28)

Sde.[`(dae-1 / dae-2)`].sta =
  is-error.sta ➔ error.sta
  Sde.[`dae-i`].sta = ? ➔ ? for i = 1, 2
  let
    val-i = Sde.[`dae-i`].sta for i = 1, 2
  val-i : Error ➔ val-i for i = 1, 2
  let
    (dat-i, bod-i, yok-i) = Sde.[`dae-i`].sta for i = 1, 2
  bod-i ≠ (‘real’) ➔ ‘real-expected’ for i = 1, 2
  let
    rea = divide-re.(dat-1, dat-2))
  rea : Error ➔ rea
  true ➔ (rea, (‘real’), TT)

primitive operation

A manual of a programming language based on denotational semantics

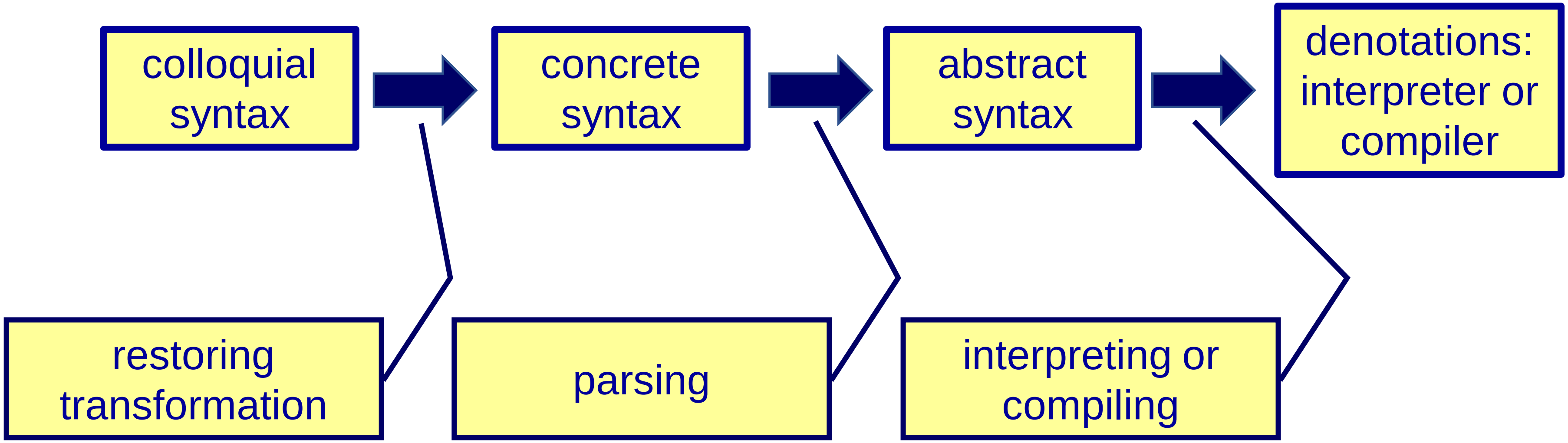
Three parts of a manual

1. concrete syntax described by equational grammar and illustrated by examples,
2. colloquial syntax illustrated by examples of restoring transformations (e.g. as in Sec. 4.4.3),
3. the semantics of concrete syntax, i.e. the association of concrete programs to their denotations without referring to abstract syntax.

Two forms of a manual

A. definitions that refer to (“call”) denotation-algebra constructors defined earlier such definitions will be called algebraic (for implementors)
B. definitions that describe constructors explicitly, such definitions will be called direct. (for users)

Major milestones on a way to program execution




Thank you for
your attention