A Denotational Engineering of Programming Languages

...

Part 3: Lingua-A – From data to values

(Sections 4.1 – 4.3 of the book)

Andrzej Jacek Blikle

March 11th, 2021

The main goals of our project (a repetition)

A reverse approach to the correctness of programs:
Constructing correct programs instead of proving programs correct

1. To perform this task we build a programming language equipped with:
   - program-construction rules that guarantee program correctness,
   - error-detection mechanism with error diagnosis/elaboration.
2. The prove the soundness of construction rules we need a mathematical semantics of the language.
3. Our choice is denotational semantics.

A reverse approach to building a language
Syntax derived from semantics

Lingua – a virtual language to illustrate our approach

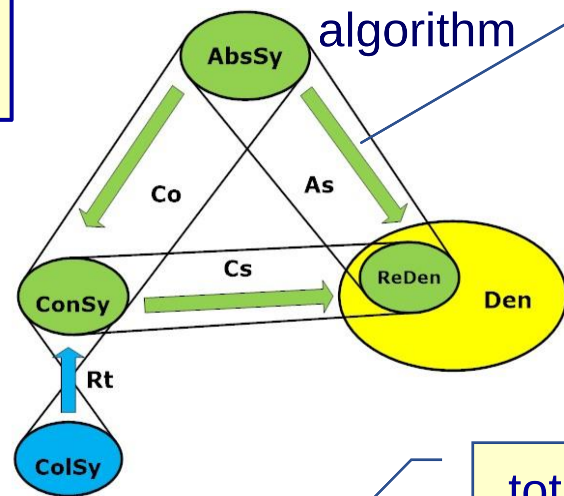
A

A repetition of an algebraic model of a programming language

If Co is an isomorphism, Cs exists and
Cs = Co^{-1} ● As

0% creativity algorithm

The denotational semantics of ConSy (automatic derivation)



1% creativity

98% creativity

1% creativity

partial functions

total functions

Example of a carrier and two constructors:

ded : DatExpDen = State → Value | Error (*data-expression denotations*)

plus : DatExpDen x DatExpDen ⟼ DatExpDen (*plus constructor*)

less : DatExpDen x DatExpDen ⟼ DatExpDen (*less constructor*)

Examples of three syntaxes of expressions

less(plus(x, times(z,y)), times(x,z)) (*abstract-syntax expression*)

((x +(z*y))<(x*z)) (*concrete-syntax expression*)

x + z*y < x*z (*colloquial-syntax expression*)

A family of Lingua languages

Lingua-A	an applicative part: data, types, expressions
Lingua-1	assignments, structural instructions, declarations
Lingua-2	procedures and recursion
LinguaV-2	tools for building correct (validated) programs
Lingua-SQL	an API for SQL databases
Lingua-OO	object-oriented programming

But a standard can be derived from it

Lingua discussed in this course is only an example used to illustrate the method of Denotational Engineering.

Lingua is not regarded as a proposal of a standard!

A

Four priorities about Lingua

- Simplicity of the model — the simplicity of denotations, syntax, and semantics; e.g., the resignation from **goto** instruction and self-applicative procedures.
- Simplicity of metaprogram construction rules; e.g., the assumption that the declarations of variables, types, and procedures should always be placed at the beginning of a program.
- Protection against “oversight errors” of a programmer; e.g., the resignation of global variables in imperative procedures and of side-effects in functional procedures.
- User-oriented semantics (easy to understand) rather than implementor-oriented semantics (easy to implement).

The role of types in Lingua

If we do not provide (...) correct values to functions, we should not expect consistent results.

DuBois Paul, *MySQL*

Instead we implement the rule:

Whenever we provide incorrect values to a function, program will generate an error message which:

- will indicate the cause of the error,
- and possibly will initiates a recovery action.

A

Lingua is a strongly-typed language

1. A type describes the structure of a data (number, array,…) and possibly some other properties (e.g. integrity constraints in SQL).
2. Types are self-standing mathematical beings (rather than sets of data), but each type defines uniquely a set of data of that type called its "clan".
3. Each variable has a type assigned to it; this type remains fixed in the course of program execution.
4. Programs operate on values which are pairs (data, type). Values are assigned to identifiers in memory states, and expressions evaluate to values.
5. Type analysis precedes the following actions :
    - assigning a value to a variable,
    - applying an operation to its arguments (to values),
    - passing actual parameters to a procedure,
    - returning formal reference parameters of a procedure.
6. An algebra of types provides tools for the construction of user-defined types.

A

Data domains and primary constructors

First step of **Lingua Project**

Data domains determine data that the future language will manipulate.

Primary constructors determine ways in which these data will be manipulated.

Every primary constructors should be definable by operations available on an implementation platform (IP).

test wiarygodności

For every primary constructor we define a trust test which protects the constructor from being applied where it cannot yield an acceptable result.

e.g. division by zero, overflow, etc.

A

Data domains

Data

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

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}

wor : Word = {‘}Character*{‘} with len.wor $\leq 2^{24} - 5$ an example

dat : SimpleDat = Boolean | Integer | Real | Word

lis : List = Data^{c*}

arr : Array = Integer $\Longrightarrow$ Data

rec : Record = Identifier $\Longrightarrow$ Data

dat : Data = SimpleData| List | Array | Record

domain recursion

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

A

Primary operations and source operations

IP operations (IPO) – provided by implementation platform
primary operations (PO) – defined operations that "call" IPO operations

trust.PO : Domain_1 x …x Domain_n $\longmapsto$ Error | {‘OK’} – trust test

if trust.PO.(arg_1,…,arg_n) = 'OK' then PO.(arg_1,…,arg_n) yields correct result and
PO.(arg_1,…,arg_n) = IPO.(arg_1,…,arg_n)

Example

trust.divide-PO.(rea-1, rea-2) =
rea-i : Error ➔ rea-i for i = 1, 2
rea-2 = 0 ➔ ‘division-by-zero’
(rea-1 / rea-2) !: [- $1{,}8 \times 10^{308}$, $1{,}8 \times 10^{308}$] ➔ ‘overflow’
true ➔ ‘OK’

divide-PO.(rea-1, rea-2) =
trust.divide.(rea-1, rea-2) : Error ➔ trust.divide-in.(rea-1, rea-2)
true ➔ divide-IPO.(rea-1, rea-2)

Primary operations

Assumed as examples for our couse

Families of zero-argument operations (constants)

create-id.ide	: $\mapsto$ Identifier	for all	ide	: Identifier
create-bo.boo	: $\mapsto$ Boolean	for all	boo	: Boolean
create-in.int	: $\mapsto$ Integer	for all	int	: IntegerS
create-re.rea	: $\mapsto$ Real	for all	rea	: RealS
create-wo.wor	: $\mapsto$ Word	for all	wor	: WordS

For instance:

create-id.size.() = size where size : Identifier
create-bo.tt.() = tt
create-in.127.() = 127

Notation:

DomainE = Domain | Error

Primary constructors (cont.)

Assumed as examples for our course

Comparison constructors

equal	: DataE x DataE	$\longmapsto$	BooleanE
less	: DataE x DataE	$\longmapsto$	BooleanE

Integer constructors

add-in	: IntegerE x IntegerE	$\longmapsto$	IntegerE
divide-in	: IntegerE x IntegerE	$\longmapsto$	IntegerE

etc.

add-re	: RealE x Real E	$\longmapsto$	RealE
divide-re	: RealE x RealE	$\longmapsto$	RealE

etc.

Word constructors

glue	: WordE x WordE	$\longmapsto$	WordE

List constructors

create-li	: DataE	$\longmapsto$	ListE
push	: DataE x ListE	$\longmapsto$	ListE
top	: ListE	$\longmapsto$	DataE
pop	: ListE	$\longmapsto$	ListE

Primary constructors (cont.)

Assumed as examples for our course

Array constructors

create-ar : DataE $\longmapsto$ ArrayE
put-to-ar : DataE x ArrayE $\longmapsto$ ArrayE
change-in-ar : ArrayE x IntegerE x DataE $\longmapsto$ ArrayE
get-from-ar : ArrayE x IntegerE $\longmapsto$ DataE

Record constructors

create-re : Identifier x DataE $\longmapsto$ RecordE
put-to-re : DataE x RecordE x Identifier $\longmapsto$ RecordE
get-from-re : RecordE x Identifier $\longmapsto$ DataE
change-in-re : RecordE x Identifier x DataE $\longmapsto$ RecordE

There are no Boolean constructors – and, or, not – on our list, since they have to be defined separately for yokes and data-expression denotations (due to their laziness).

Primary constructors (cont.)

An engineering decision about arrays

create-ar : DataE ⟼ ArrayE

create-ar.dat =
 dat : Error → dat
 trust.create-ar.dat ≠ 'OK' → trust.create-ar.dat
 true → [1/dat]

implementation dependent

put-to-ar : DataE x ArrayE ⟼ ArrayE

put-to-ar.(dat, arr) =
 dat : Error → dat
 arr : Error → arr
 trust.put-to-ar.(dat, arr) ≠ 'OK' → trust.put-to-ar.(dat, arr)
 let
  n = max-ind.arr (the largest index in arr)
 true → arr[n+1/dat]

the domain of indexes is of the form {1,…,n}

A

Our goal: values as well-typed data

value = (data, type) (dana, typ)
type = (body, yoke) (korpus, jarzmo)
composite = (data, body)

the structure of data

other properties of data (of composites)

an example of a record body:
[name / ('word'),
salary / ('integer'),
commission / ('integer')]

an example of a yoke:
salary + commission < 7000
small integer
(integrity constraints in SQL)

- data expressions will evaluate to values
- values will be saved in memory states

- type expressions will evaluate to types
- types will be saved in memory states

Bodies: finitistic structures of data

(korpusy)

SimBod = {(‘Boolean’)} | {(‘integer’)} | ('real') |{(‘word’)} (*simple bodies*)
LisBod = {‘L’} x Body (*list bodies*)
ArrBod = {‘A’} x Body (*array bodies*)
RecBod = {‘R’} x (Identifier ⇒ Body) (*record bodies*)
bod : Body = SimBod | ListBod | ArrBod | RecBod

body record

korpus rekordowy

bod : BodyE = Body | Error

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

Examples of bodies:

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

A

Clans of bodies

(to associate data with their bodies)

CLAN-Bo : BodyE ⟼ Sub.Data

With every body we assign a set of data.

CLAN-Bo.err = Ø for err : Error

CLAN-Bo.('Boolean') = Boolean

CLAN-Bo.('integer') = Integer

CLAN-Bo.('word') = Word

CLAN-Bo.('L', bod) = $(\text{CLAN-Bo.bod})^{c*}$

CLAN-Bo.('A', bod) = Integer ⟹ CLAN-Bo.bod

CLAN-Bo.('R', [ide-1/bod-1,…,ide-n/bod-n]) =

{ [ide-1/dat-1,…,ide-n/dat-n] | dat-i : CLAN-Bo.bod-i for i = 1;n}

Not all data have bodies.

Clans of different bodies are disjoint.

BOD : Data → Body (partial function)

BOD.dat — the body of dat.

BOD.dat = bod where dat : CLAN-Bo.bod

BOD.ide = ide by definition

non-homogeneous list have no bodies, e.g.:

BOD.('abc', 23, tt) = ?

Expressions will not generate data which have no bodies.

A

Algebra of bodies

Anticipating the future constructors of composites, values, and data-expression denotations

<u>AlgBod</u>
Ide : Identifier = …
bod : BodyE = Body | Error

Error detection at the level of bodies will protect composite- and value constructors from receiving inappropriate arguments.

Zero-argument body constructors

bo-create-id.ide : $\mapsto$ Identifier for all ide (a family of constructors)
bo-create-boo : $\mapsto$ BodyE
bo-create-int : $\mapsto$ BodyE
bo-create-wor : $\mapsto$ BodyE

For every primary constructor pco we assign a corresponding body constructor bo-pco.

Their definitions

bo-create-id.ide.( ) = ide for all ide
bo-create-boo.( ) = ('Boolean')
bo-create-int.( ) = ('integer')
bo-create-wor.( ) = ('word')

Algebra of bodies
Constructors of simple bodies

All body constructors will be transparent for errors.

Comparison constructors

bo-equal : BodyE x BodyE ⟼ BodyE
bo-less : BodyE x BodyE ⟼ BodyE

Arithmetic constructors

bo-add-in : BodyE x BodyE ⟼ BodyE
bo-divide-in : BodyE x BodyE ⟼ BodyE
etc. for integers and reals

Word constructor

bo-glue : BodyE x BodyE ⟼ BodyE

List constructors

bo-create-li : BodyE ⟼ BodyE
bo-push : BodyE x BodyE ⟼ BodyE
bo-top : BodyE ⟼ BodyE
bo-pop : BodyE ⟼ BodyE

Algebra of bodies

Constructors of structured bodies

Array constructors

bo-create-ar	: BodyE	$\mapsto$ BodyE
bo-put-to-ar	: BodyE x BodyE	$\mapsto$ BodyE
bo-check-in-ar	: BodyE x BodyE x BodyE	$\mapsto$ BodyE
bo-get-from-ar	: BodyE x BodyE	$\mapsto$ BodyE

Record constructors

bo-create-re	: Identifier x BodyE	$\mapsto$ BodyE
bo-put-to-re	: Identifier x BodyE x BodyE	$\mapsto$ BodyE
bo-get-from-re	: BodyE x Identifier	$\mapsto$ BodyE
bo-check-in-re	: BodyE x Identifier x BodyE	$\mapsto$ BodyE

No Boolean constructors!

Algebra of bodies

Examples of constructor definitions

bo-create-lis.bod =
  bod : Error ➔ bod
  true ➔ (‘L’, bod)

bo-push.(bod-e, bod-l) = push element bod-e on list bod-l
  bod-i : Error ➔ bod-i for i = e,l
  sort.bod-l ≠ ‘L’ ➔ ‘list-expected’
  let
    (‘L’, bod) = bod-l
  bod-e ≠ bod ➔ ‘conflict-of-bodies’
  true ➔ bod-l

homogeneity of lists

A

Algebra of bodies

Examples of constructor definitions

bo-check-in-re.(bod-r, ide, bod-e) =
  bod-i : Error ➔ bod-i   for i = r,e
  sort.bod-r ≠ 'R' ➔ 'record-expected'
  let
    ('R', bod-rb) = bod-r   -rb for „record body"
  bod-rb.ide = ? ➔ 'no-such-attribute'
  let
    bod-ab = bod-rb.ide   -ab for "attribute body"
  bod-e ≠ bod-ab ➔ 'conflict-of-bodies'
  true ➔ bod-r

body must not change

This costructor only checks if the new body coinsides with the former. It may raise an error, but does not change record body.

Algebra of bodies

Examples of constructor definitions

bo-equal.(bod-1, bod-2) =
- bod-i : Error ➔ bod-i **for** i = 1,2
- bod-1 ≠ bod-2 ➔ ‘different-bodies’
- not-comparable.bod-1 ➔ ‘not-comparable’
- **true** ➔ (‘Boolean’)

an implementation-dependent predicate

Algebra of composites
Domains

com : Composite = { (dat, bod) : Data x Body | dat : CLAN-Bo.bod }
com : CompositeE = Composite | Error
com : BooCompositeE = { (tt, ('Boolean')), (ff, ('Boolean')) }

AlgCom:
Identifier = …
CompositeE = …

A

Algebra of composites

Constructors of simple composites

Zero-argument constructors (indexed families)

co-create-id.ide	: $\longmapsto$ Identifier	for ide	: Identifier
co-create-bo.boo	: $\longmapsto$ Composite	for boo	: Boolean
co-create-in.int	: $\longmapsto$ Composite	for int	: IntegerS
co-create-wo.wor	: $\longmapsto$ Composite	for wor	: WordS

Comparison constructors

co-equal : CompositeE x CompositeE $\longmapsto$ CompositeE
co-less : CompositeE x CompositeE $\longmapsto$ CompositeE

Arithmetic constructors

co-add : CompositeE x CompositeE $\longmapsto$ CompositeE
co-divide : CompositeE x CompositeE $\longmapsto$ CompositeE

etc. for integers and reals

Word constructors

co-glue : CompositeE x CompositeE $\longmapsto$ CompositeE

Algebra of composites

Constructors of structured composites

List constructors

co-create-li	: CompositeE	$\longmapsto$ CompositeE
co-push	: CompositeE x CompositeE	$\longmapsto$ CompositeE
co-top	: CompositeE	$\longmapsto$ CompositeE
co-pop	: CompositeE	$\longmapsto$ CompositeE

Array constructors

co-create-ar	: CompositeE	$\longmapsto$ CompositeE
co put-to-ar	: CompositeE x CompositeE	$\longmapsto$ CompositeE
co-change-in-ar	: CompositeE x CompositeE x CompositeE	$\longmapsto$ CompositeE
co-get-from-ar	: CompositeE x CompositeE	$\longmapsto$ CompositeE

Record

co-create-re	: Identifier x CompositeE	$\longmapsto$ CompositeE
co-put-to-re	: Identifier x CompositeE x CompositeE	$\longmapsto$ CompositeE
co-get-from-re	: CompositeE x Identifier	$\longmapsto$ CompositeE
co-change-in-re	: CompositeE x Identifier x CompositeE	$\longmapsto$ CompositeE

No Boolean constructors again!

A

Algebra of composites

General assumptions about constructors

General scheme of definitions:

1. check if the argument composites are not errors, and if they are not then,
2. compute the resulting body, and if no error is signalized then,
3. compute the resulting data (<u>trust check</u>), and if no error is signalized then,
4. combine body and data into composite.

All composite constructors will be transparent for errors

Two auxiliary functions:

data.(dat, bod) = dat
body.(dat, bod) = bod

data.ide = ide
body.ide = ide

A

Algebra of composites

Examples of constructor definitions

co-create-id.ide : $\mapsto$ Identifier for ide : Identifier
e.g. create-id.length.() = length

co-create-bo.boo : $\mapsto$ Composite for boo : Boolean
e.g. create-bo.tt.() = (tt, ('Boolean')

co-create-re.rea : $\mapsto$ Composite for num : RealS
e.g. create-re.23,75 = (23,75, ('real'))

co-create-wo.wor : $\mapsto$ Composite for wor : WordS
e.g. create-wo.'abc' = ('abc', ('word'))

S – syntactically representable

Assumptions about constants:

(1) Do not generate errors.
(2) Do not generate oversized data
(3) Generate syntactically representable composites
 (can be typed in keyboard)

We do not assume that our operations generate only S-representable composites

A

Algebra of composites
Numerical division

```
co-divide-in.(com-1, com-2) =
  com-i : Error    ➔ com-i        for i = 1, 2
  let
    (dat-i, bod-i) = com-i        for i = 1, 2
    bod = bo-divide-in.(bod-1, bod-2)
  bod : Error      ➔ bod
  let
    int =divide-in.(dat-1, dat-2)   - primary operation (performs trust check)
  int : Error      ➔ int
  true             ➔ (int, ('integer'))
```

Algebra of yokes
Domains and first examples

jarzma

Yokes describe these properties of composites that cannot be described by bodies.

AlgYok
ide : Identifier = …
tra : Transfer = CompositeE ⟼ CompositeE
yok: Yoke = CompositeE ⟼ BooCompositeE

No error elements in carriers

CLAN-Yo : Yoke ⟼ Sub.Composite
CLAN-Yo.yok = {com | yok.com = (tt, ('Boolean'))}

AlgYok is one-level-up wrt AlgCom since its elements are composite constructors.

Examples of transfer expressions:

```
2 , value , value + 2
record.salary + record.commission
```

Examples of yoke expressions:

TT - always satisfied

```
small integer, sorted list,
1,93 ≤ value ≤ 2,47
record.salary + record.commission < 7000
```

Yokes appear in SQL as integrity constraints.

A

Algebra of yokes
Six groups of constructors

(1) Constructors of identifiers

create-id.ide : $\mapsto$ Identifier for ide : Identifier

(2) Identity transfer

pass : $\mapsto$ Transfer

pass.().com = com for com : CompositeE

tra-i – either transfer or ide

ide.com = ide

cco : ComIde-1 x … x ComIde-n $\mapsto$ CompositeE

Tc[cco].(tra-1, …, tra-n).com = cco.(tra-1.com, …, tra-n.com)

not for boolean cco

(3) Examples of constructors of transfers based on simple-composite operations

Tc[co-create-in.int] :		$\mapsto$ Transfer	for int : IntegerS
Tc[co-create-wo.wor] :		$\mapsto$ Transfer	for wor : WordS
Tc[co-add-in]	: Transfer x Transfer	$\mapsto$ Transfer	
Tc[co-divide-in]	: Transfer x Transfer	$\mapsto$ Transfer	
Tc[sum]	: Transfer	$\mapsto$ Transfer	
Tc[max]	: Transfer	$\mapsto$ Transfer	

From outside of AlgCom

Algebra of yokes

An example of a constructor

Tc[cco].(tra-1, …, tra-n).com = cco.(tra-1.com, …, tra-n.com)

The denotation of transfer expression

value+2

Tr[co-add-in].(pass.(), Tc[create-in.2]).com =
co-add-in.(pass.().com, Tc[create-in.2].com) =
co-add-in.(com, (2, ('integer'))

co-add-in.(com, (2, ('integer'))) =
com : Error ➔ com
let
(dat, bod) = com
bod ≠ ('integer') ➔ 'integer-required'
let
new-dat = add-in.(dat, 2)
new-dat : Error ➔ new-dat
true ➔ (new-dat, ('integer'))

This example explains why we need pass constructor.

Algebra of yokes

Six groups of constructors (cont.)

(4) Constructors of transfers based on selection operations for list, arrays and records

Tc[co-top] : Transfer $\mapsto$ Transfer

Tc[co-get-from-ar] : Transfer x Transfer $\mapsto$ Transfer

Tc[co-get-from-re] : Transfer x Identifier $\mapsto$ Transfer

(5) Constructors of yokes based on predicates

Tc[co-create-bo.boo] : $\mapsto$ Yoke for boo : Boolean

Tc[co-equal] : Transfer x Transfer $\mapsto$ Yoke

Tc[co-less] : Transfer x Transfer $\mapsto$ Yoke

Tc[increasing-nu] : Transfer $\mapsto$ Yoke

from outside of AlgCom

Algebra of yokes

Six groups of constructors (end)

(6) Constructors of yokes based on Kleene's operators

As in SQL due to the lack of functional procedures

yo-and : Yoke x Yoke ⟼ Yoke
yo-or : Yoke x Yoke ⟼ Yoke
yo-not : Yoke ⟼ Yoke
all-on-li : Transfer x Yoke ⟼ Yoke
all-in-ar : Transfer x Yoke ⟼ Yoke

from outside of AlgCom

and-Y is commutative (except for errors)

yo-and.(yok-1, yok-2).com =
  com : Error ➔ com
  let
    com-i = yok-i.com for i = 1, 2
  com-1 = (ff, ('Boolean')) ➔ (ff, ('Boolean'))
  com-2 = (ff, ('Boolean')) ➔ (ff, ('Boolean'))
  com-i : Error ➔ com-i for i = 1, 2
  body.com-i ≠ ('Boolean') ➔ 'Boolean expected' for i = 1, 2
  true ➔ (tt, ('Boolean'))

yo-not = Tc[co-not]
yo-or – De Morgan

No procedure calls hence no infinite executions as in the case of Boolean expression.

At least one has to be false.

Algebra of types

typ : Type = Body x Yoke

CLAN-Ty : Type ⟼ Sub.Data

CLAN-Ty.(bod, yok) = {dat | dat : CLAN-Bo.bod **and** (dat, bod) : CLAN-Yo.yok}

<u>AlgTyp</u>

ide	: Identifier	= …
bod	: BodyE	= …
tra	: Transfer	= …
yok	: Yoke	= …
typ	: TypeE	= Type | Error

Types will be assigned in states to:

- data variables,
- type constants
- formal parameters of procedures

Due to yokes types may have subtypes.

Algebra of types

Constructors

Constructors of identifiers

create-id.ide : $\mapsto$ Identifier for ide : Identifier

Selected constructors of bodies

bo-create-bo	:	$\mapsto$ BodyE
bo-create-in	:	$\mapsto$ BodyE
bo-create-wo	:	$\mapsto$ BodyE
bo-create-li	: BodyE	$\mapsto$ BodyE
bo-create-ar	: BodyE	$\mapsto$ BodyE
bo-create-re	: Identifier x BodyE	$\mapsto$ BodyE
bo-put-to-re	: BodyE x BodyE x Identifier	$\mapsto$ BodyE

All constructors of the algebra of yokes (including transfers)

…

Constructors of adding and modifying yokes

create-ty : BodyE x Yoke $\mapsto$ TypeE

Algebra of types

Constructors

ty-create-ty : BodyE x Yok $\mapsto$ TypeE
ty-create-ty.(bod, yok) =
bod : Error ➔ bod
true ➔ (bod, yok)

This constructors allows building types
with empty clans.

Error elaboration mechanisms in Lingua-1 and
Lingua-2 will signalize such cases

Examples of type declarations and the declarations of variables

```
set years_register_type as
 array-type number
 with all-in-arr 2000 ≤ value 2100 ee
tes

set employee_type as
 record-type
  ch-name, fa-name of type word,
  birthyear of type number,
  awards of type years_register_type
 ee
tes
```

here yoke is trivial (TT)

type declarations assign types to type constants in states

```
let smith be employee_type tel
let awards_Smith
 be years_register_type tel
```

variable declarations assign types to variables in states

Algebra of values

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

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

val : ValueE = Value | Error

(dat, bod, TT) – a yokeless value

Carriers of AlgVal

ide : Identifier = …

tra : Transfer = …

yok : Yoke = …

val : ValueE = …

A repetition about values:

- values will be assigned to identifiers in states,
- data expressions will evaluate to values,
- values will be passed to procedures as actual parameters.

Constructors of AlgVal

1. all constructors of the algebra of yokes (including transfers),
2. value constructors derived from all composite constructors,
3. specific value constructors.

A

Algebra of values

Constructors of simple values

Zero-argument constructors

create-id.ide	:	$\longmapsto$ Identifier	for all ide : Identifier
va-create-bo.boo	:	$\longmapsto$ ValueE	for all boo : Boolean
va-create-in.int	:	$\longmapsto$ ValueE	for all int : IntegerS
va-create-wo.wor	:	$\longmapsto$ ValueE	for all wor : WordS

Comparison constructors

va-equal	: ValueE x ValueE	$\longmapsto$ ValueE
va-less	: ValueE x ValueE	$\longmapsto$ ValueE

some values may be not comparable

Numerical constructors

va-add-in	: ValueE x ValueE	$\longmapsto$ ValueE
va-divide-in	: ValueE x ValueE	$\longmapsto$ ValueE

etc. for integers and reals

Word constructor

va-glue	: ValueE x ValueE	$\longmapsto$ ValueE

Algebra of values

Constructors of structured values

List constructors

va-create-li	: ValueE	$\mapsto$ ValueE
va-push	: ValueE x ValueE	$\mapsto$ ValueE
va-top	: ValueE	$\mapsto$ ValueE
va-pop	: ValueE	$\mapsto$ ValueE

Array constructors

va-create-ar	: ValueE	$\mapsto$ ValueE
va-put-to-ar	: ValueE x ValueE	$\mapsto$ ValueE
va-change-in-ar	: ValueE x ValueE x ValueE	$\mapsto$ ValueE
va-get-from-ar	: ValueE x ValueE	$\mapsto$ ValueE

Record constructors

va-create-re	: Identifier x ValueE	$\mapsto$ ValueE
va-put-to-re	: Identifier x ValueE x ValueE	$\mapsto$ ValueE
va-get-from-re	: ValueE x Identifier	$\mapsto$ ValueE
va-change-in-re	: ValueE x Identifier x ValueE	$\mapsto$ ValueE

Algebra of values

A general scheme of composite-driven transparent constructors

cco : ComIde x … x ComIde $\mapsto$ CompositeE
va-cco : ValIde x … x ValIde $\mapsto$ ValueE

Here we have to take engineering decisions about yoke mechanism in Lingua-WU.

va-cco.(arg-1,…,arg-n) =
  arg-i : Error ➔ arg-i for i = 1;n
  let
    c-arg-i = the i-th argument of cco
      arg-i : Identifier ➔ arg-i
      true ➔ com-i where arg-i = (com-i, yok-i)
    new-com = cco.(c-arg-1,…,c-arg-n)
  new-com : Error ➔ new-com
  let
    new-yoke = … here an engineering decision in each concrete case
    boo-com = new-yok.new-com
  boo-com : Error ➔ boo-com
  boo-com = (ff, (‘Boolean’)) ➔ ‘resulting-yoke-not-satisfied’
  true ➔ (new-com, new-yoke)

A

Algebra of values

Composite-driven constructors – an example

```
va-divide-in.(val-1, val-2) =
    val-i : Error    ➔ val-i                    for i = 1,2
    let
        (com-i, yok-i) = val-i                  for i = 1,2
        com = co-divide-in.(com-1, com-2)
    com : Error      ➔ com
    true             ➔ (com, TT)
```

an engineering decision

this composite constructor cares about error detection (slide 17)

A

Algebra of values

Composite-driven constructors – an example

```
va-push.(val-e, val-l) =                                         push val-e on list val-l
  val-i : Error                 ➔ val-i         for i = e,l
  let
    (dat-i, bod-i, yok-i) = val-i               for i = e,l
    com = co-push.((dat-e, bod-e), (dat-l, bod-l))
  com : Error                   ➔ com
  let
    yo-com = yok-l.com
  yo-com : Error                ➔ yo-com
  yo-com = (ff, (‘Boolean’))➔ ‘resulting-yoke-not-satisfied’
  true                          ➔ (com, yok-l)
```

an engineering decision
(but rather evident)

Algebra of values

A comment on yoke-maniputation mechanizms

At the level of value constructors we only "compute" yokes of values created by value constructors.

We do not have constructors of the type:

change-yoke : ValueE x Yoke $\mapsto$ ValueE

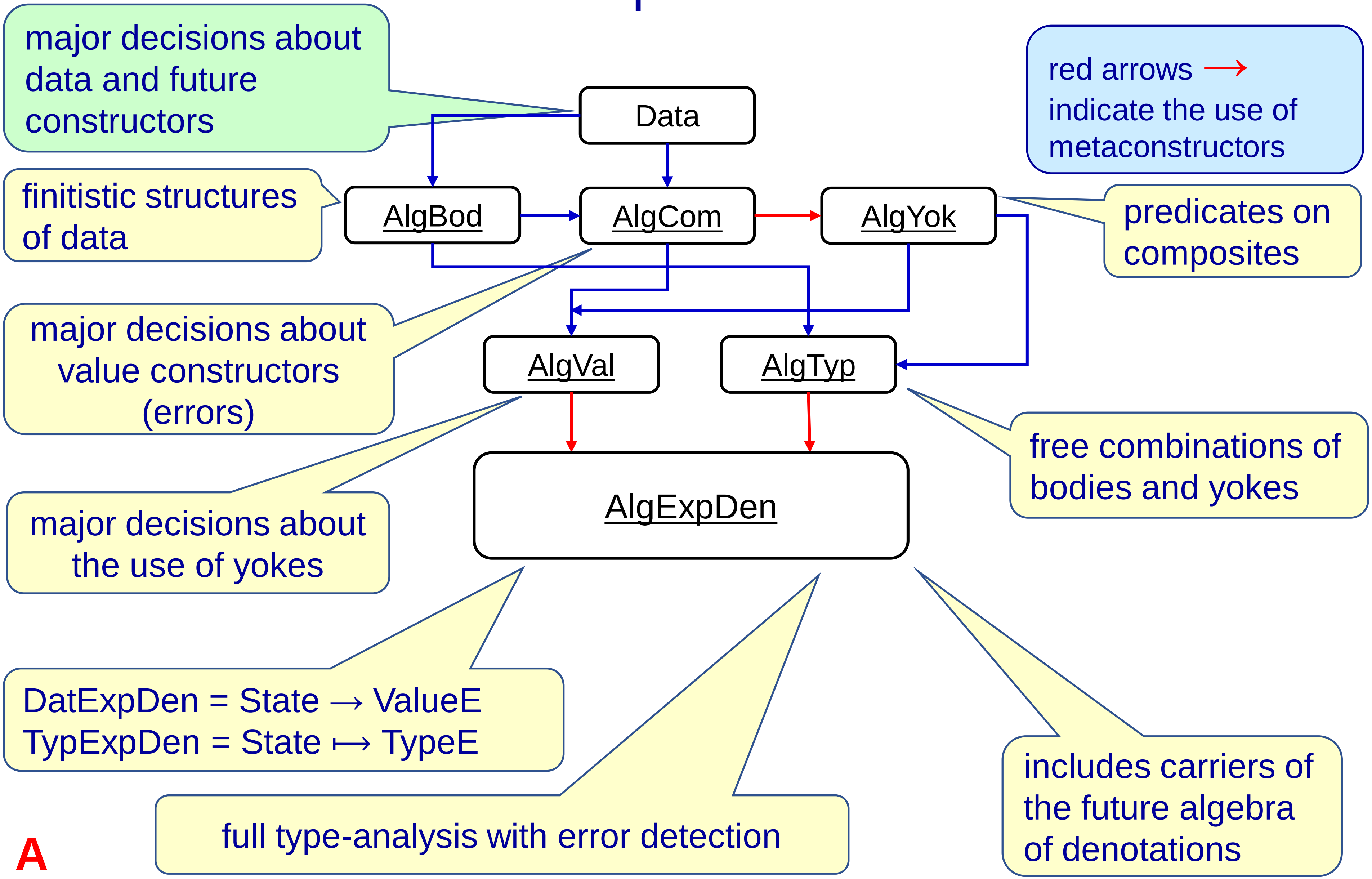
that would allow for the replacement of the yoke of a value. This will also concern data expressions.

The manipulation of yokes assigned to variables will be available at the level of:

- variable declarations,
- yoke replacement instruction.

This is an engineering decision.

Milestones on the way from data to expression denotations




Thank you for
your attention