Lingua Project

(2) Data and their types

(Sections 2.1, 4.1, 4.2 and 4.3)

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

Andrzej Jacek Blikle

February 1st, 2025

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 family of Lingua's layers

Applicative layer	data, types, values, objects, expressions
Imperative layer	declarations, instructions, procedures (with recursion)
Lingua-SQL	API for SQL databases (a sketch of)
Lingua-concurrency	constructors of programs with a simple concurrency (simple Petri nets)
Lingua-V	constructors of programs; programs proved by construction

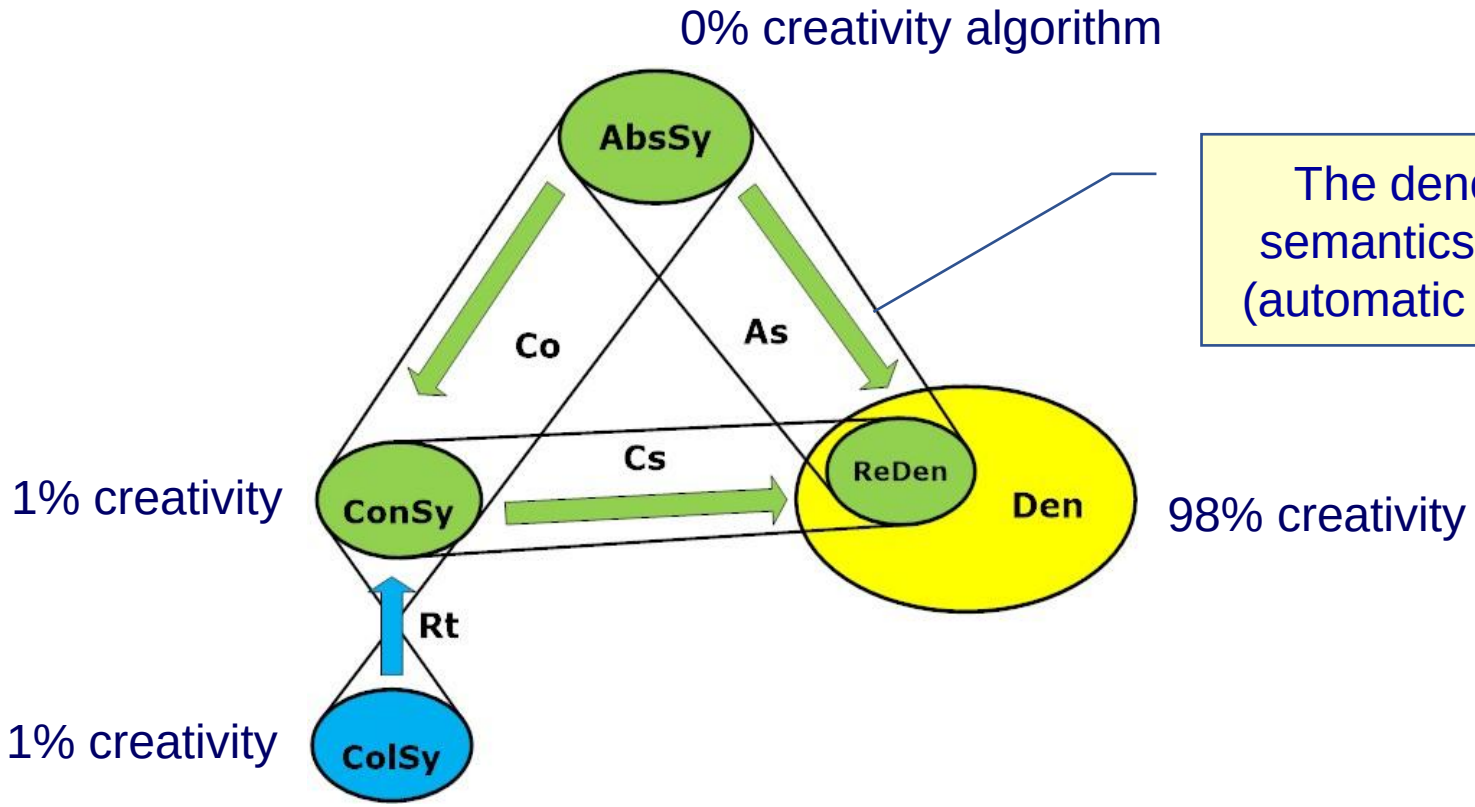
But a standard can be derived from it

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

A repetition of a denotational model of a programming language



Some more notation

Curried functions (Haskell Curry):

function	f : A x B x C → D
may be regarded as	f : A → B → C → D
we write	f.a.b.c = ((f.a).b).c

Definednes of functions:

f.a = ! f.a is defined

f.a = ? f.a is undefined

'!' and '?' are not math. entities

Case-by-case definitions of functions:

```
f.a =
  p-1.a  ➔ g-1.a     otherwise
  p-2.a  ➔ g-2.a     otherwise
  …
  true   ➔ g-n.a     (in remaining cases)
```

Definitions with local variables:

```
f.a =
  p-1.a  ➔ g-1.a     otherwise
    let b = h.a
  p-2.b  ➔ g-2.b     otherwise
  …
  true   ➔ g-n.a     (in remaining
                      cases)
```

Overwriting of functions:

```
(f♦g).a =
  g.a = !  ➔ g.a
  true     ➔ f.a
```

Overwriting of functions:

```
f[a-1/b-1,…,a-n/b-n].a =
  a = a-1  ➔ b-1
  …
  a = a-n  ➔ b-n
  true     ➔ f.a
```

Finite explicit functions:

```
[a-1/b-1,…,a-n/b-n].a =
  a = a-1  ➔ b-1
  …
  a = a-n  ➔ b-n
```

A

The transparency of a function

We introduce a universal set of abstract errors (words)

Error = {'division by zero', 'overflow', 'index out of range',…}

For every domain Dom we introduce a domain with errors

DomE = Dom | Error

A function

fun : DomE x … x DomE → DomE

is said to be transparent for errors if

fun.(d-1,…,d-n) = d-k where d-k is the first argument in Error.

The majority of functions that we will use (but not all!) will be transparent.

A

Propositional calculus of J. Mc'Carthy
(non-transparent operations)

if x ≠ 0 **and** 1/x < 10 **then** x := x+1 **else** x := x–1 **fi**
If **and** would be transparent, then our program aborts for x = 0.

The solution of John McCarthy:

ff **and-m** ee = ff — lazy evaluation from left to right

error or undefinedness

or-m	**tt**	**ff**	**ee**
tt	tt	tt	tt
ff	tt	ff	ee
ee	ee	ee	ee

and-m	**tt**	**ff**	**ee**
tt	tt	ff	ee
ff	ff	ff	ff
ee	ee	ee	ee

not-m	
tt	ff
ff	tt
ee	ee

No commutativity
p **or-m** q ≠ q **or-m** p
p **and-m** q ≠ q **and-m** p

-m stands for McCarthy

Propositional calculus of S. Kleene

Even „more lazy” than McCarthy’s calculus

or-k	**tt**	**ff**	**ee**
tt	tt	tt	tt
ff	tt	ff	ee
ee	tt	ee	ee

and-k	**tt**	**ff**	**ee**
tt	tt	ff	ee
ff	ff	ff	ff
ee	ee	ff	ee

not-k	
tt	ff
ff	tt
ee	ee

Commutativity

p **or-k** q = q **or-k** p

p **and-k** q = q **and-k** p

in particular (non-transparency)

tt **or-k** ee = ee **or-k** tt = tt

ff **and-k** ee = ee **and-k** ff = ff

If ee may be an infinite computation Kleene's calculus requires a simultaneous evaluation of arguments.

Domain equations

Fixed-point equations (recursive equations):
F(x) = x solutions not always exist!

A legal domain recursion – the least solution exists and is unique

Value = Integer | Record
Record = Identifier ⟹ Value

An illegal domain recursion – no solution

State = Identifier ⟹ Value | Procedure
Procedure = State → State

This operator makes the recursion illegal

In classical set-theory
functions that can take themselves as their arguments (i.e., f(f))
do not exist

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,
- will possibly initiate a recovery action.

A

Lingua as a strongly-typed language

1. A type describes the structure of a data (number, array,…).
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 a language design

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-operations.

test wiarygodności

For every primary constructor we define a trust test which protects the constructor from being applied if it can't yield an acceptable result.

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

A

Data domains

ide : Identifier — a finite subset of Character^{+}

Data

dat : Data = SimpleData | List | Array | Record
dat : SimpleDat = Boolean | Integer | Real | Text
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}
tex : Text = {'}Character*{'} with len.tex ≤ $2^{24} - 5$ an example
lis : List = Data^{c*}
arr : Array = Integer ⟹ Data
rec : Record = Identifier ⟹ Data

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

Data constructors – typical examples

comparison constructors

da-equal	: DataE x DataE	⟼ BooleanE
da-less	: DataE x DataE	⟼ BooleanE

integer constructors

da-add-in	: IntegerE x IntegerE	⟼ IntegerE
da-subtract-in	: IntegerE x IntegerE	⟼ IntegerE
…		

real-number constructors

da-add-re	: RealE x RealE	⟼ RealE
…		

array constructors

da-create-ar	:	⟼ ArrayE
da-put-to-ar	: ArrayE x DataE	⟼ ArrayE
da-get-from-ar	: ArrayE x IntegerE	⟼ DataE

record constructors

da-create-rc	: Identifier x DataE	⟼ RecordE
da-put-to-rc	: DataE x RecordE x Identifier	⟼ RecordE
da-get-from-rc	: RecordE x Identifier	⟼ DataE
da-change-in-rc	: RecordE x Identifier x DataE	⟼ RecordE

no boolean-data constructors at this stage

Prime constructors and data constructors

IP-constructors – assumed to be offered by an implementation platform
primary constructors – designer-defined operations that "call" IP-constructors
trust tests – designer-defined operations

Example

trust.da-divide-re : RealE x RealE ⟼ Error | {‘OK’} trust test

trust.da-divide-re.(rea-1, rea-2) =

rea-i : Error	➔ rea-i for i = 1, 2
rea-2 = 0	➔ ‘division-by-zero not allowed’
IP-divide-re.(rea-1, rea-2) > max-re	➔ ‘overflow’
IP-divide-re.(rea-1, rea-2) < min-re	➔ ‘underflow’
true	➔ 'OK'

IP-constructor

da-divide-re.(rea-1, rea-2) =

trust.da-divide-re.(rea-1, rea-2) : Error	➔ trust.da-divide-re.(rea-1, rea-2)
true	➔ IP-divide-re.(rea-1, rea-2)

Data constructors (cont.)
An example of an engineering decision

da-create-ar : DataE ⟼ ArrayE

da-create-ar.dat =
  dat : Error ➔ dat
  trust.da-create-ar.dat ≠ ‘OK’ ➔ trust.da-create-ar.dat
  true ➔ [1/dat]

da-put-to-ar : DataE x ArrayE ⟼ ArrayE

da-put-to-ar.(dat, arr) =
  dat : Error ➔ dat
  arr : Error ➔ arr
  trust.da-put-to-ar.(dat, arr) ≠ ‘OK’ ➔ trust.da-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

The types of data - datatypes

Datatypes are finitistic elements that describe structures of (corresponding) data

typ : DatTyp =

{‘integer’, ‘real’, ‘boolean’, ‘text’}	|	simple types
{‘L’} x DatTyp	|	list types
{‘A’} x DatTyp	|	array types
{‘R’} x (Identifier ⟹ DatTyp)		record types

type record

record typów

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

Examples of types:

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

A

Clans of datatypes

(to associate data with their types)

CLAN-ty : DatTyp ⟼ Sub.Data

Not all data have types.

Clans of different types are disjoint.

CLAN-ty.(‘boolean’) = Boolean

CLAN-ty.('integer') = Integer

CLAN-ty.(‘text’) = Text

CLAN-ty.(‘L’, bod) = (CLAN-ty.bod)^{c*}

CLAN-ty.(‘A’, bod) = Integer ⟹ CLAN-ty.bod

CLAN-ty.(‘R’, [ide-1/bod-1,…,ide-n/bod-n]) =

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

an auxiliary function – the sort of a type

sort-t : DatTyp ⟼ {‘boolean’, ‘integer’, ‘real’, ‘text’, ‘L’, ‘A’, ‘R’}

sort-t.typ =

typ = 'boolean' ➔ 'boolean'

…

typ : {'L'} x DatTyp ➔ 'L'

…

A

Constructors of datatypes
preliminary assumptions

typ : DatTypE = DatTyp | Error

Error detection at the level of types protects (future) value constructors from receiving inappropriate arguments.

For every data constructor da-con we assign a corresponding type constructor ty-con.

All type constructors will be transparent for errors.

Constructors of datatypes (cont.)

Comparison constructors

ty-equal : DatTypE x DatTypE ⟼ DatTypE
ty-less : DatTypE x DatTypE ⟼ DatTypE

Arithmetic constructors

ty-add-in : DatTypE x DatTypE ⟼ DatTypE
ty-divide-in : DatTypE x DatTypE ⟼ DatTypE
etc. for integers and reals

Text constructor

ty-glue-te : DatTypE x DatTypE ⟼ DatTypE

List constructors

ty-empty-li : DatTypE ⟼ DatTypE
ty-put-to-li : DatTypE x DatTypE ⟼ DatTypE
ty-head-li : DatTypE ⟼ DatTypE
ty-tail-li : DatTypE ⟼ DatTypE

Constructors of datatypes (cont.)

Array constructors

ty-create-ar	: DatTypE	⟼ DatTypE
ty-put-to-ar	: DatTypE x DatTypE	⟼ DatTypE
ty-change-in-ar	: DatTypE x DatTypE x DatTypE	⟼ DatTypE
ty-get-from-ar	: DatTypE x DatTypE	⟼ DatTypE

Record constructors

ty-create-rc	: Identifier x DatTypE	⟼ DatTypE
ty-put-to-rc	: Identifier x DatTypE x DatTypE	⟼ DatTypE
ty-get-from-rc	: DatTypE x Identifier	⟼ DatTypE
ty-change-in-rc	: DatTypE x Identifier x DatTypE	⟼ DatTypE

No boolean constructors!

Constructors of datatypes (cont.)
examples of definitions

```
ty-divide-in.(typ-1, typ-2) =                    division of integers
   typ-i : Error          ➔ typ-i                for i = 1,2
   typ-i ≠ 'integer'      ➔ 'integer expected'   for i = 1,2
   true                   ➔ 'integer'

ty-empty-li.typ =                                creation of an empty list
   typ : Error            ➔ typ
   true                   ➔ ('L', typ)

ty-put-to-li.(typ-e, typ-l) =                    putting a new element on a list
   typ-i : Error          ➔ typ-i                for i = e,l
   sort-t.typ-l ≠ 'L'     ➔ 'list expected'
   let
      ('L', typ) = typ-l
   typ-e ≠ typ            ➔ 'conflict of types'
   true                   ➔ typ-l
```

Typed data
definition

tyd : TypDat = {(dat, typ) | dat : CLAN-ty.typ}

Why shall we use typed data rather than just data?

1. Not all data have types, e.g., nonhomogeneous arrays have no types.
2. Typed data allow us to describe our typing discipline and eliminate typing errors.
3. We can describe type-covering relations.
4. Types will be used in checking the typing adequacy of:
    a. arguments of functions in the evaluation of expressions,
    b. actual parameters of procedures.

Another auxiliary function:
sort-td.(dat, typ) = sort-t.typ

Typed data
the signatures of constructors

With every data constructor we associate a corresponding typed data constructor

Comparison constructors

td-equal	: TypDatE x TypDatE	⟼ TypDatE
td-less	: TypDatE x TypDatE	⟼ TypDatE

Arithmetic constructors for integers

td-add-in	: TypDatE x TypDatE	⟼ TypDatE
td-subtract-in	: TypDatE x TypDatE	⟼ TypDatE
td-multiply-in	: TypDatE x TypDatE	⟼ TypDatE
td-divide-in	: TypDatE x TypDatE	⟼ TypDatE

Arithmetic constructors for reals

(analogous)

Text constructors

td-glue-tx	: TypDatE x TypDatE	⟼ TypDatE

List constructors

td-empty-li	: DatTypE	⟼ TypDatE
td-put-to-li	: TypDatE x TypDatE	⟼ TypDatE
td-head-li	: TypDatE	⟼ TypDatE
td-tail-li	: TypDatE	⟼ TypDatE

And analogously for other data constructors.

Typed data
examples of the definitions of constructors

td-divide-in : TypDatE x TypDatE

td-divide-in.(tyd-1, tyd-2) =
  tyd-i : Error ➔ tyd-i for i = 1, 2
  let
    (dat-i, typ-i) = tyd-i for i = 1, 2
    typ = ty-divide-in.(typ-1, typ-2)
  typ : Error ➔ typ
  let
    dat = da-divide-in.(dat-1, dat-2)
  dat : Error ➔ dat
  true ➔ (dat, typ)

A general rule:

1. check for errors,
2. compute the resulting type; detect errors,
3. compute the resulting data; detect errors,
4. combine data with type.


Thank you for
your attention