An experiment with denotational semantics (reversing the traditional order of things)

Presentation and book (PL & GB) on
http://www.moznainaczej.com.pl/inzynieria-denotacyjna
http://www.moznainaczej.com.pl/denotational-engineering

Andrzej Blikle
in cooperation with Piotr Chrząstowski-Wachtel

March 8th, 2019

The philosophy of the method

What I am trying to do?

To suggest a way of improving the quality of programs.

THE QUALITY OF A PROGRAM:

1. the compliance of that prog.-specification with user's expectations
2. the compliance of that prog. with its specifications

Currently for Pascal-like languages (no concurrency).
That was my research area in the years 1970-1990.

Why I dare to tacle the problem?

The state of the art in IT industry

An example of a disclosure *There is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.* ***The entire risk as to the quality and performance of the program is with you.*** *Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.*

The state of the art. in IT science

The KeY Book; From Theory to Practice (Springer 2016)
For a long time, the term formal verification was almost synonymous with functional verification. In the last years, it became more and more clear that ***full functional verification is an elusive goal for almost all application scenarios.*** *(…) Not verification but specification is the real bottleneck in functional verification.*

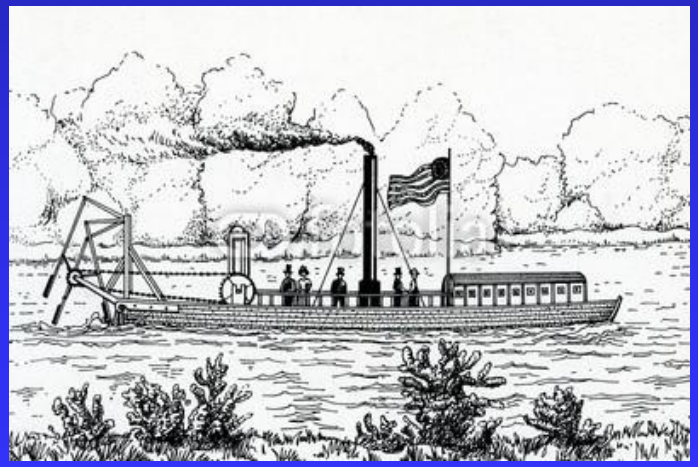
Why earlier attempts failed?

(although some experiments are still in the course)

In order to build a logic of programs
a mathematical semantics must be defined.

Two historical attempts to the definitions of mathematical semantics:

An operational semantics (VDL);
describe a virtual computer



Denotational semantics (VDM)
S : Language → Denotations
S(P♦Q) = S(P) • S(Q)

Ada and Chill, 1980.

S : AlgSyn → AlgDen

SEMANTICS
A homomorphism
of many-sorter algebras

Can a denotational semantics be written for any language?

My hypothesis
Probably not – at least not for the languages that I know.

And certainly this hasn't been done so far.

A traditional approach to building denotational semantics

First syntax:
how to talk about

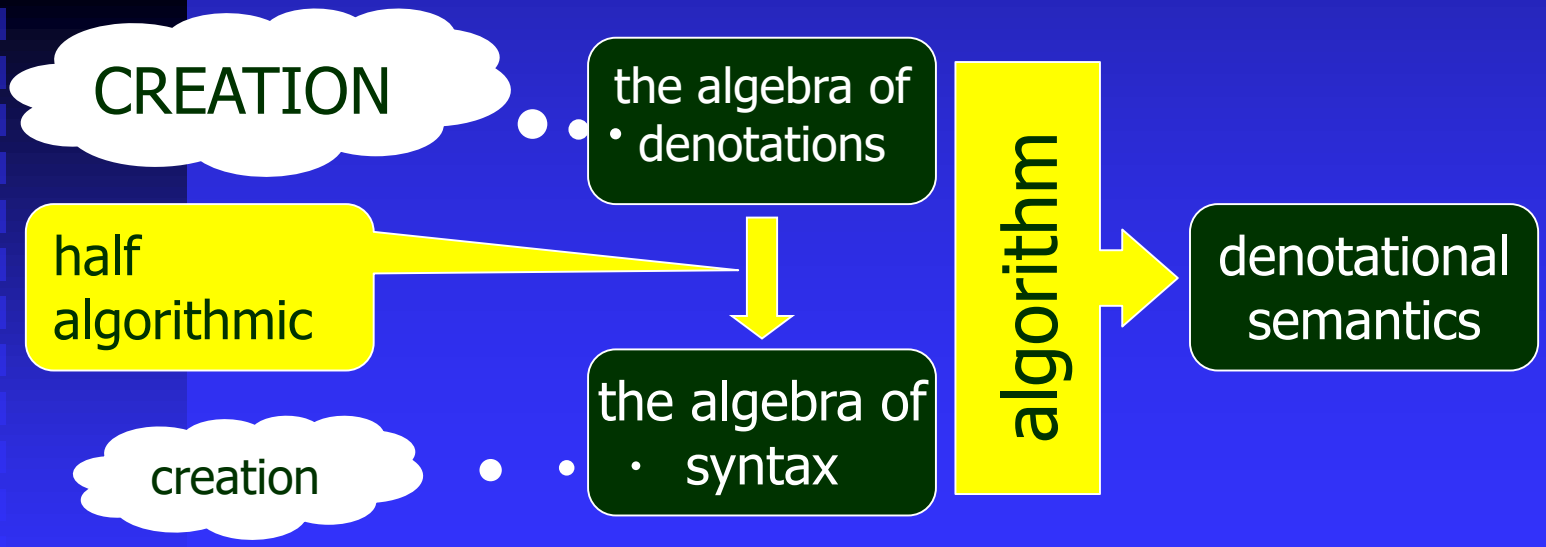
Then denotations
what to talk about

This order has a historical justification. When people started to think about semantics the syntaxes were already there.

Let's reverse the usual order of things

First describe the world of denotation: an algebra of the denotations of programs' components.

Then derive from it the corresponding syntax



When we have a languages with
denotational semantics,
we can think about proving programs correct.

Is proving programs correct
a right way
to validate programs?

Two problems:

1. A proof is usually longer then a theorem.
2. Programs are usually incorrect.

Let's reverse the usual order again

A mathematician
First a theorem, then a proof

An engineer
First a project (proof), then a product (e.g. a bridge)

Proof rules should be replaced by sound program-construction rules

Validating programming

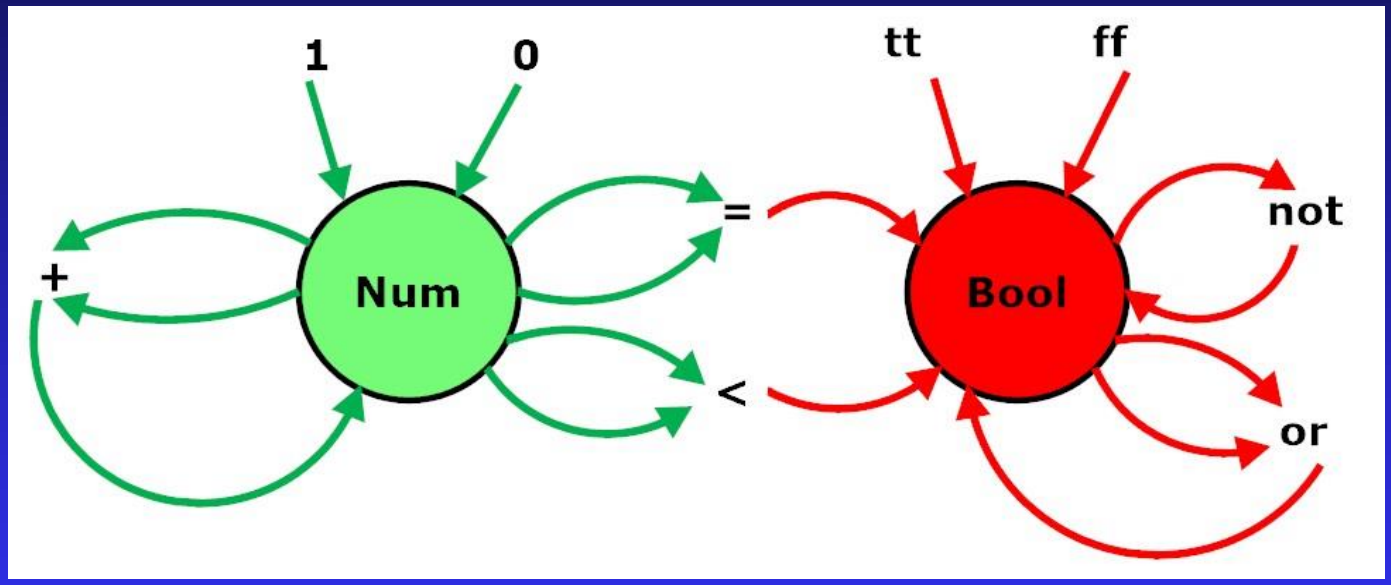
The general idea of a denotational model

These ideas have been published in my papers
in the years 1971 – 1989
(some with Antoni Mazurkiewicz and Andrzej Tarlecki)

MATHEMATICAL TOOLS

- fixed-point theory in CPO's
- set-theoretic domain equations (no Scott's reflexive domains)
- three-valued predicate calculus
- many-sorter algebras
- abstract errors for error-handling mechanizm

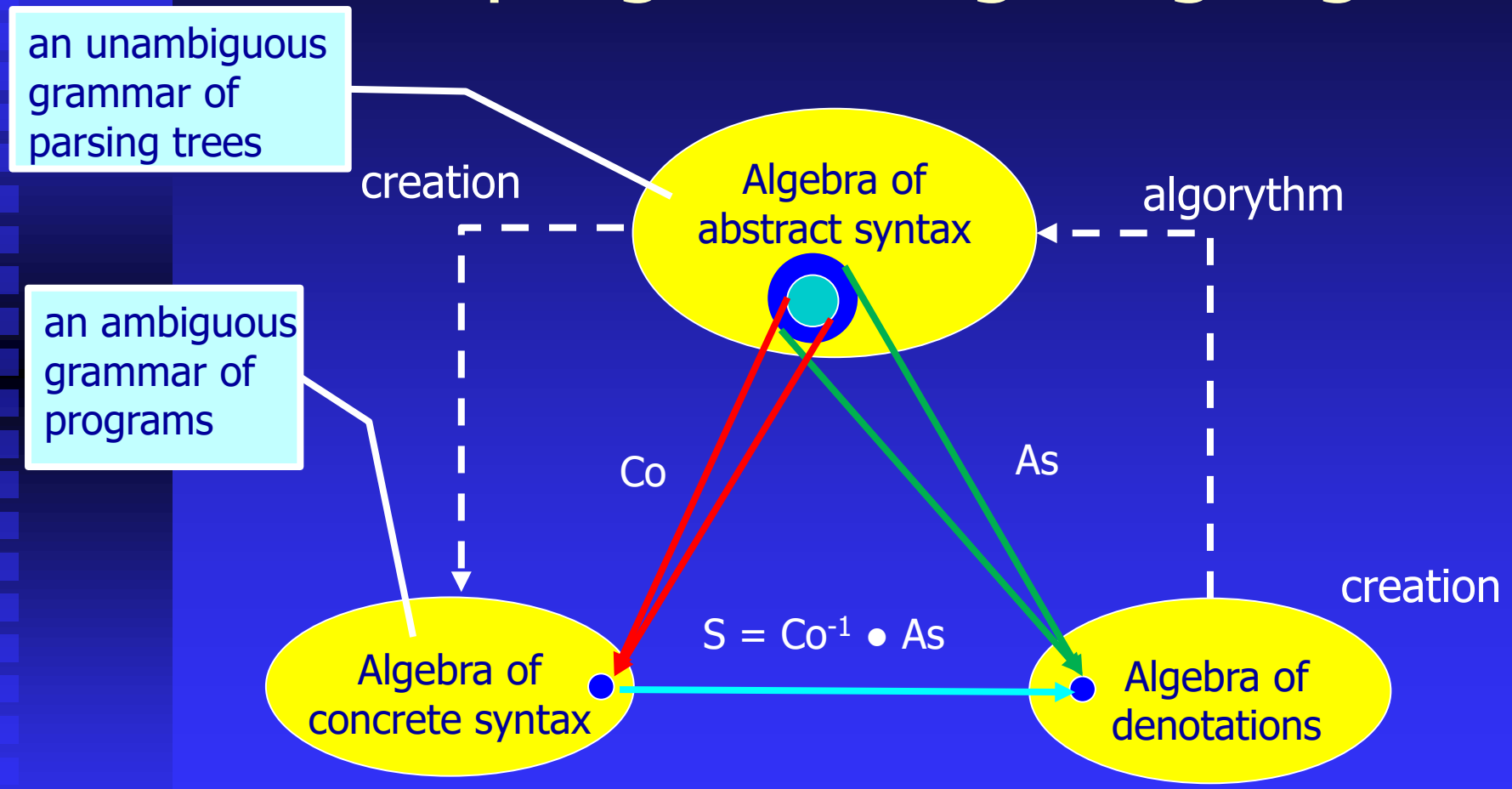
An example of a many-sorted algebra



TWO SORTS OF THE ELEMENTS OF THE ALGEBRA:

- numbers, e.g. real numbers
- Boolean values

A denotational model of a programming language



If Co glues not more than As, then the (unique) homomorphism S exists.

A toy example, part 1

An algebra of denotations

Carriers

Ide = {x, y, z,...}

ExpDen = State → Number

InsDen = State → State

State = Ide ⇒ Number

Constructors

var : Ide ↦ ExpDen

plus : ExpDen x ExpDen ↦ ExpDen

times : ExpDen x ExpDen ↦ ExpDen

assign : Ide x ExpDen ↦ InsDen

compose : InsDen x InsDen ↦ InsDen

Notation:

A → B; partial fun.

A ↦ B; total fun.

A ⇒ B; finite fun.

ALGORITHM

The algebra (grammar) of abstract syntax

Ide = {x, y, z,...}

Exp = var(Ide) | plus(Exp, Exp) | times(Exp, Exp)

Ins = assign(Ide, Exp) | compose(Ins, Ins)

ALGORITHM

The semantics of abstract syntax (As)

Sid : Ide ↦ Ide (identity)

Sex : Exp ↦ ExpDen

Sin : Ins ↦ InsDen

The algebra (grammar) of abstract syntax

Ide = {x, y, z}

Exp = var(Ide) | plus(Exp, Exp) | times(Exp, Exp)

Ins = assign(Ide, Exp) | compose(Ins, Ins)

CREATION
assisted

The algebra (grammar) of concrete syntax

Ide = {x, y, z}

Exp = Ide | (Exp + Exp) | (Exp * Exp)

Ins = Ide := Exp | Ins ; Ins

acceptable ambiguity

The algebra (grammar) of colloquial syntax

Ide = {x, y, z}

Exp = Ide | (Exp + Exp) | (Exp * Exp)
Exp + Exp | Exp * Exp

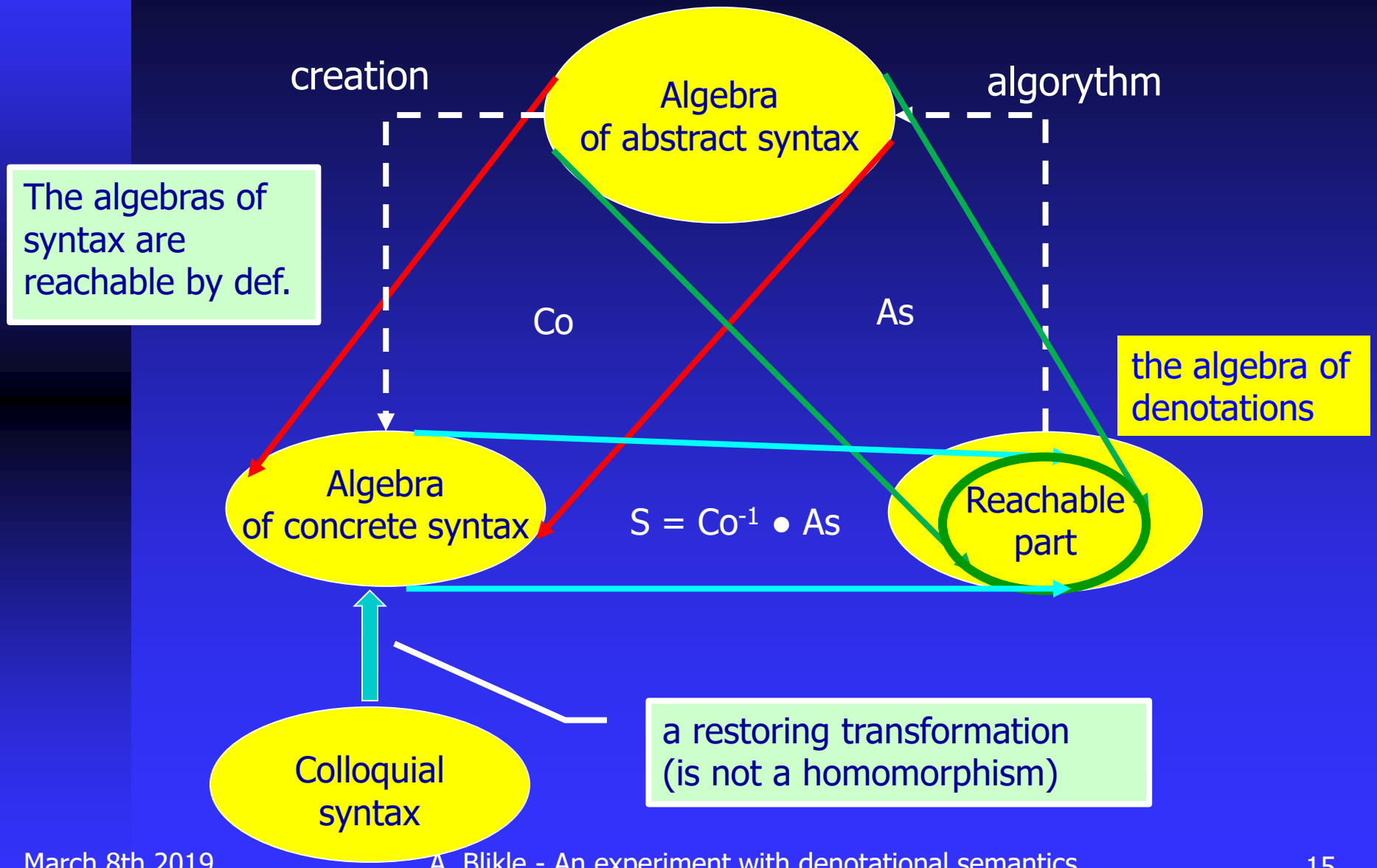
Ins = Ide := Exp | Ins ; Ins

CREATION
assisted

There is no denotational semantics for colloquial syntax!

not acceptable ambiguity

A model with a colloquial syntax



Lingua – an example languages
where to explain selected applications of the model

- Booleans, numbers, words, lists, arrays, record and their arbitrary combinations plus SQL data-bases
- three-valued propositional calculus for Boolean expressions
- abstract errors incorporated into the algebras of denotations
- user-defined structured types,
- basic programming constructors (:=, if-then-else-fi, while-do-od)
- procedures with recursion and multirecursion
- sound program-constructors based on Hoare's logic with clean termination (three-valued predicate calculus)

not covered in this presentation

Data and their domain equations

ide : Identifier = …

GENERAL

boo : Boolean = {tt, ff}
num : Number = …
wor : Word = {`}Alphabet*{`}
lis : List = Data^{c*}
arr : Array = Number ⟹ Data
rec : Record = Identifier ⟹ Data
dat : Data = Boolean | Number | Word | List | Array | Record

Domain equations define larger sets of data than their future reachable parts.

No abstract errors at this stage!

SQL

dat : SimData = Boolean | Number | Word | Date | Time … | {θ}
row : Row = Identifier ⟹ SimData
tab : Table = Row^{c*}

an empty field of a table

Bodies and composites

a common body of all the elements of a list/array

GENERAL

bod : Body = {('Boolean'), ('number'), ('word')} |
{'L'} x Body | (*list bodies*)
{'A'} x Body | (*array bodies*)
{'R'} x (Identifier ⟹ Body) (*record bodies*)

SQL

sbo : SimBody = {('Boolean'), ('number'), ('word'), ('date'),…}
bod : RowBody = {'Rq'} x (Identifier ⟹ SimBody)
bod : TabBody = {'Tq'} x Row x (Identifier ⟹ SimBody)

CLAN-bo : Body ⟼ Set.Data
CLAN-bo.('number') = Number
CLAN-bo.('L', ('number')) = Number^{c*}

com : Composite = {(dat, bod) | dat : CLAN-bo.bod}
com : BooComposite = {(boo, ('Boolean')) | boo : {tt, ff} }
com : CompositeE = Composite | Error
com : BooCompositeE = BooComposite | Error

errors are words (messages)

Transfers and yokes

tra : Transfer = CompositeE ⟼ CompositeE
yok : Yoke = CompositeE ⟼ BooCompositeE

Yokes describe properties of composites

EXAMPLES OF TRANSFER EXPRESSIONS

record.salary - if the argument carries a record with attribute 'salary', then the associated data and body otherwise error message, e.g., 'record-expected'
record.salary + **record**.bonus

record.salary + **record**.bonus < 7000 - a yoke of records
all-list (**record**.salary + **record**.bonus < 7000) - a yoke of lists of rec.
SMALLINT - a SQL yoke of numbers
DECIMAL(p,s) - a SQL yoke of numbers

In Lingua-SQL yokes describe integrity constraints except the subordination relations between tables. The latter are described in a different way.

Types and values

```
typ : Type    = Body x Yoke
typ : TypeE   = Type | Error

val : Value   = {(dat, (bod, yok)) |
                        dat : CLAN-bo.bod &
                        (dat, bod).yok =  (tt, ('Boolean')) }
```

values	are assigned to identifiers in states
types	are assigned to identifiers in states and are the results of type-expression evaluations
composites	are the results of data-expression evaluations

Types are storable in states, but composites and transfers are not.
This is an engineering decision rather than a mathematical neccessity.

States and denotations

STATES

sta : State = Env x Store
sto : Store = Valuation x (Error | {'OK'})
vat : Valuation = Identifier ⟹ Value
env : Env = ProEnv x TypEnv
tye : TypEnv = Identifier ⟹ Type
pre : ProEnv = Identifier ⟹ Procedure
pro : Procedure = ImpPro | FunPro
ipr : ImpPro = ActPar x ActPar ⟼ Store → Store
fpr : FunPro = ActPar ⟼ State → Composite | Error
apa : ActPar = Identifier^{c*}

to avoid selfapplicability

DENOTATIONS

ded : DatExpDen = State ⟶ CompositeE
ted : TypExpDen = State ⟼ TypeE
tra : TraExpDen = Transfer
vdd : VarDecDen = State ⟼ State
tdd : TypDefDen = State ⟼ State
pdd : ProDecDen = State ⟼ State
ind : InsDen = State ⟶ State

selected carriers of the algebra of denotations

The constructors of denotations

(a few examples)

dat-variable	: Identifier	⟼ DatExpDen
typ-constant	: Identifier	⟼ TypExpDen
dat-plus	: DatExpDen x DatExpDen	⟼ DatExpDen
typ-plus	: TypExpDen x TypExpDen	⟼ TypExpDen
call-fun-pro	: Identifier x ActPar	⟼ DatExpDen
assign	: Identifier x DatExpDen	⟼ InsDen
while	: DatExpDen x InsDen	⟼ InsDen

An example of a constructor definition

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

Lingua-SQL

from bird's-eye view

In Lingua we already have
(to refresh the memory)

SQL BODIES
sbo : SimBody = {(‘Boolean’), ('number’), (‘word’), (‘date’),…}
bod : RowBody = {‘Rq’} x (Identifier ⟹ SimBody)
bod : TabBody = {‘Tq’} x Row x (Identifier ⟹ SimBody)

SQL YOKES
record.salary + **record**.bonus < 10.000
SMALLINT
DECIMAL(p,s)

SQL VALUES
RowVal = {(row, bod), tra) | …}
TabVal = {(tab, bod), tra) | …}

Adding: subordination graphs and data-base values

sgr : SubGra = Sub.(Identifier x Identifier x Identifier)

child column parent

dbr : DatBasRec = Identifier ⟹ TabVal - data-base record
dbv : DbaVal = {(dbr, sgr) | dbr satisfies sgr} - data-base value

Data-base values are assigned to identifiers in states.

In order to operate on a data-base, it has to be activated. This means that in the current state:

- its tables are assigned to identifiers,
- its subordination graph is assigned to a system-identifier 'sb-graph'.

A colloquial SQL declaration of a table variable

```
create table Employees with
  Name      Varchar(20)   NOT NULL,
  Salary    Number(5)     DEFAULT 0,
  Bonus     Number(4)     DEFAULT 0,
  Dep_Id    Number(3)     REFERENCES Departments,
  CHECK (Bonus < Salary)
ed
```

CONCRETE-SYNTAX SCHEME

```
create table Employees as
  table-type dat_exp with yok_exp ee
ed;
set reference of Employees at Dep_Id to Departments ei
```

the row of default values

yoke

concrete syntax of an instruction that sets a subordination relations of tables

A further restoration

```
create table Employees as
  table-type dat_exp with yok_exp ee
ed;
```

```
create table Employees as
 table-type
  expand-row
   expand-row
    expand-row
     row Name val empty-word ee
    by Salary val 0 ee
   by Bonus val 0 ee
  by Dep_Id by empty-number ee
with
 all
  varchar(20)(row.Name) and
  …
  row.Bonus < row.Salary
```

dat_exp

yok_exp

Examples of research problems

Theory and software engineering:

- models for script languages, e.g.: HTML, TEX,…
- models for concurrency (a rather hard problem),
- a full system of sound program-construction rules,
- full models for Lingua-like languages

Supporting tools for language designers:

- a generator of abstr. syntax from the def. of algebra of den.,
- a dialog-generator of concrete syntax,
- a support for the creation of colloquial syntax and the restoring transformation,
- a support for the generation of semantic clauses.

Supporting tools for programmers:

- implementations of Lingua-like languages
- program editors supporting correct-program development.

Experimental applications, e.g. in microprogramming.

THANK YOU FOR YOUR ATTENTION