# A Denotational Engineering of Programming Languages

...

Part 2:Many-sorted algebras

(Sections 2.10 – 2.14 of the book)

Andrzej Jacek Blikle

March 11th, 2021

# Many-sorted algebras

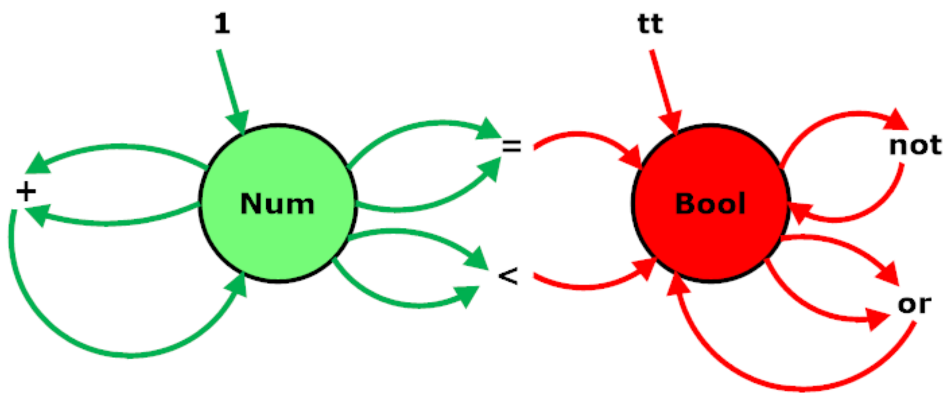
A BAD NEWS

This theory is technically a bit complicated.

A GOOD NEWS

You do not need to master it very deeply.

# Many-sorted algebras intuitively




signature

similar algebras

signature

The algebra of data NumBool

| | | | |
|---|---|---|---|
| 1 | : | | ⟼ NumE |
| + | : | NumE x NumE | ⟼ NumE |
| * | : | NumE x NumE | ⟼ NumE |
| = | : | NumE x NumE | ⟼ BoolE |
| < | : | NumE x NumE | ⟼ BoolE |
| tt | : | | ⟼ BoolE |
| not | : | BoolE | ⟼ BoolE |
| or | : | BoolE x BoolE | ⟼ BoolE |

The algebra of expressions NumBoolExp

| | | | |
|---|---|---|---|
| 1 | : | | ⟼ NumExp |
| + | : | NumExp x NumExp | ⟼ NumExp |
| * | : | NumExp x NumExp | ⟼ NumExp |
| = | : | NumExp x NumExp | ⟼ BoolExp |
| < | : | NumExp x NumExp | ⟼ BoolExp |
| tt | : | | ⟼ BoolExp |
| not | : | BoolExp | ⟼ BoolExp |
| or | : | BoolExp x BoolExp | ⟼ BoolExp |

Due to abstract errors all functions (in this case) can be made total

# Abstract and concrete syntax

Algebra of expressions NumBoolExp repeated

| | | | | | |
|---|---|---|---|---|---|
| 1 : | ↦ NumExp | | < : NumExp x NumExp | ↦ BoolExp | |
| + : NumExp x NumExp | ↦ NumExp | | tt : | ↦ BoolExp | |
| * : NumExp x NumExp | ↦ NumExp | | not : BoolExp | ↦ BoolExp | |
| = : NumExp x NumExp | ↦ BoolExp | | or : BoolExp x BoolExp | ↦ BoolExp | |

Abstract syntax Prefix notation: **not(<(+(1,*(1,1),+(1,1)))**

NumExp = **1** | **+**(NumExp**,** NumExp) | * (NumExp, NumExp)
BoolExp = **tt** | **=**(NumExp**,** NumExp) | **<**(NumExp, NumExp) |
**not**(BoolExp) | **or**(BoolExp**,** BoolExp)

Concrete syntax Infix notation: **not((1+(1*1)) < (1+1))**

NumExp = **1** | (NumExp **+** NumExp) | (NumExp * NumExp)
BoolExp = **tt** | (NumExp **=** NumExp) | (NumExp **<** NumExp) |
**not**(BoolExp) | (BoolExp **or** BoolExp)

# Abstract, concrete and colloquial syntax

Abstract syntax – algorithmically derivable from algebra's signature

`not(<(+(1,*(1,1),+(1,1)))`

no creativity

Concrete syntax – an isomomorphic transformation: abstract → concrete

`not((1+(1*1)) < (1+1))`

creativity

Colloquial syntax – a restoring transformation: concrete ← colloquial

`not(1+1*1 < 1+1)` — here the omission of "unnecessary" parentheses

Colloquial assumptions

creativity

* binds stronger than +

* and + bind stronger than <

# Denotational semantics of concrete syntax

A two-sorted homomorphism

SN : NumExp $\longmapsto$ NumE
SB : BooExp $\longmapsto$ BoolE

Semantic clauses (two examples):

The meaning of a whole is a combination of the meanings of its parts

SN[`1`] = 1

SN[`(exp-1 + exp-2)`] =
SN[`exp-1`] + SN[`exp-2`] ≥ max ➔ 'overflow'
**true** ➔ SN[`exp-1`] + SN[`exp-2`]

max — maximal number for a current implementation

# Many-sorted algebras formally

Alg = (Sig, Car, Fun, car, fun) — algebra
Sig = (Cn, Fn, ar, so) — signature

Car — a finite family of sets called carriers
Fun — a finite family of function called constructors
Cn — finite set of words; carrier names
Fn — finite set of words; function names
ar : Fn $\mapsto$ $Cn^{c*}$ — arity    car : Cn $\mapsto$ Car
so : Fn $\mapsto$ Cn — sort    fun : Fn $\mapsto$ Fun

e.g. ar.less = (number, number), so.less = boolean

similar algebras — have the same (or isomorphic) signature

an extension of Alg results from Alg by adding:

- new carriers, and/or
- new elements to the existing carriers, and/or
- new functions

# Similarity and homomorphism

$\underline{Alg}_i$ = (Sig, $Car_i$, $Fun_i$, $car_i$, $fun_i$) for i = 1,2 — similar algebras
Sig = (Cn, Fn, ar, so) — common signature

$\underline{Alg}_1$ is a subalgebra of similar $\underline{Alg}_2$ if

- $car_1.cn \subset car_2.cn$ for any cn : Cn
- constructors of $Fun_1$ coincide with the corresponding constructors of $Fun_2$ on their domains

a homomorphism $H : \underline{Alg}_1 \longmapsto \underline{Alg}_2$, $H = \{h.cn \mid cn : Cn\}$
$h.cn : Car_1.cn \longmapsto Car_2.cn$

$ar.fn = (cn_1,\ldots,cn_n)$ — arity
$so.fn = cn$ — sort
$(a_1,\ldots,a_n) : car_1.cn_1 \times \ldots \times car_1.cn_n$

kernel of H in $\underline{Alg}_2$ — the image of $\underline{Alg}_1$ in $\underline{Alg}_2$

$$h.cn.(fun_1.fn.(a_1,\ldots,a_n)) = fun_2.fn.(h.cn_1.a_1,\ldots,h.cn_n.a_n)$$

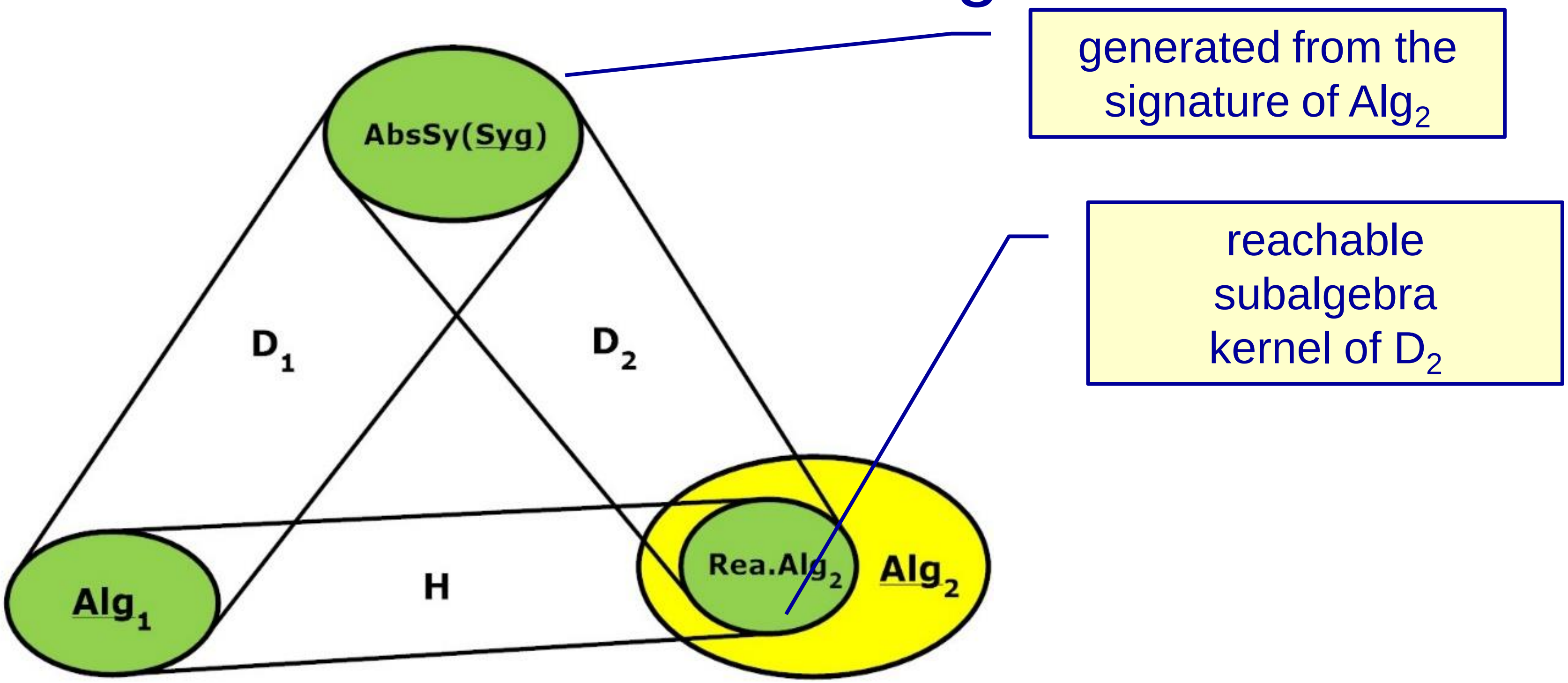
# Reachable algebras and abstract syntax

reachable subalgebra — all elements constructible by constructors
reachable algebra — identical to its (unique) reachable subalgebra
Int = (PosInt, 1, +) is a reachable subalgebra of Num = (Number, 1,+)

abstract syntax over Sig a reachable algebra denoted <u>AbsSyn(Sig)</u>
Sig = (Cn, Fn, ar, so)
carriers — formal languages over Alphabet = Fn | { (, ) } | {,}
with every fn : Fn we assign a constructor of languages

+ : Integer x Integer ⟼ Integer — a constructor of numbers
[+] : IntExp x IntExp ⟼ IntExp — a corresponding constructor of expr.
$[+].(exp_1, exp_2)$ = '+' © '(' © $exp_1$ © ',' © $exp_2$ © ')'
= $+(exp_1, exp_2)$ — a simplified notation for constructors
equational grammar:
IntExp = {1.} © {(} © {)} |
{+} © {(} © IntExp © {,} © IntExp © {)}
= 1 |
+(IntExp, IntExp) — a simplified notation for grammars

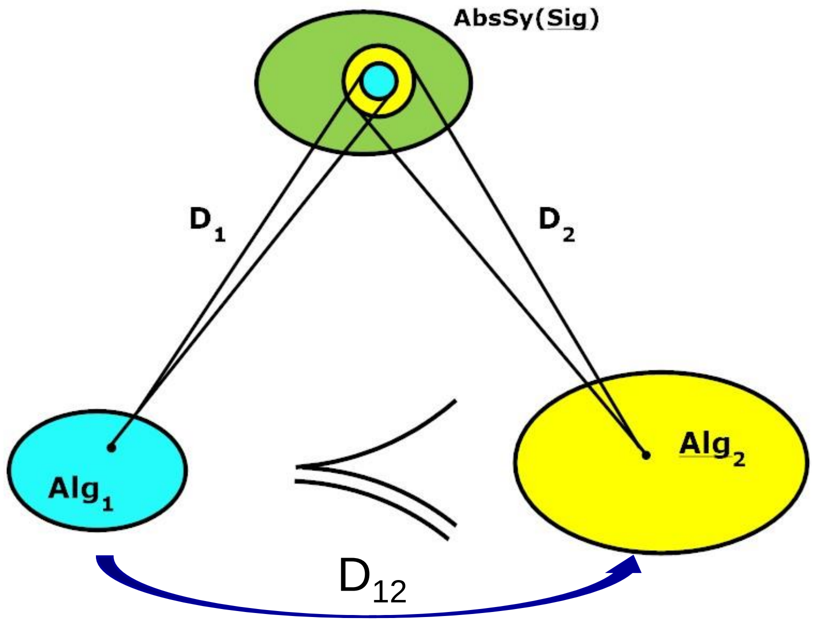
# Two facts about algebras




For every Alg with Sig there is exactly one homomorphism
$D_2$ : AbsSyn(Sig) $\longmapsto$ Alg

If $\underline{Alg}_1$ and $\underline{Alg}_2$ are similar and $\underline{Alg}_1$ is reachable then there is at most one homomorphism
H : $\underline{Alg}_1 \longmapsto \underline{Alg}_2$ (H : $\underline{Alg}_1 \longmapsto$ Reachable.$\underline{Alg}_2$)

# Ambiguous and unambiguous algebras




An algebra is called ambiguous, if its unique homomorphism from abstract syntax is not a one-one homomorphism (if it is gluing)

Algebra $Alg_1$ is said to be not more ambiguous than $Alg_2$, if $D_1$ is gluing not more then $D_2$.

If

- $Alg_1$ and $Alg_2$ are similar (have a common signature) and
- $Alg_1$ is reachable,

then

the (unique) homomorphism $D_{12} : Alg_1 \longmapsto Alg_2$ exists iff $Alg_1 \preccurlyeq Alg_2$.

If $D_1$ is an isomorphism then $Alg_1$ is unambiguous, and the (unique) homomorphism $D_{12} : Alg_1 \longmapsto Alg_2$ exists ($D_{12} = D_1^{-1} \bullet D_2$)

# Syntactic algebras versus grammars

An algebra is called a syntactic algebra, if it is a reachable algebra of words.

DEF A skeleton function: $f.(x_1,\ldots,x_k) = w_1x_1\ldots w_kx_nw_{k+1}$.
$(w_{1,}\ldots w_k, w_{k+1})$ — skeleton

F.(exp-b, $ins_1$, $ins_2$) = **if** exp-b **then** $ins_1$ **else** $ins_2$ **fi** — F is skeleton f.
F.(exp-b, $ins_1$, $ins_2$) = **if** exp-b **then** $ins_2$ **else** $ins_1$ **fi** — F is not skeleton f

DEF A contex-free algebra – all its constructors are skeleton functions

For every context-free algebra there is an equational grammar that generates is carriers.

For every equational grammar there is a context-free algebra with carriers defined by that grammar

# Colloquial syntax versus traditional approach

CONCRETE SYNTAX

ConExp = 1 | (ConExp + ConExp) |
(ConExp * ConExp)

Parentheses are optional

COLLOQUIAL SYNTAX

ColExp = 1 | (ColExp + ColExp) |
ColExp + ColExp |
(ColExp * ColExp) |
ColExp * ColExp

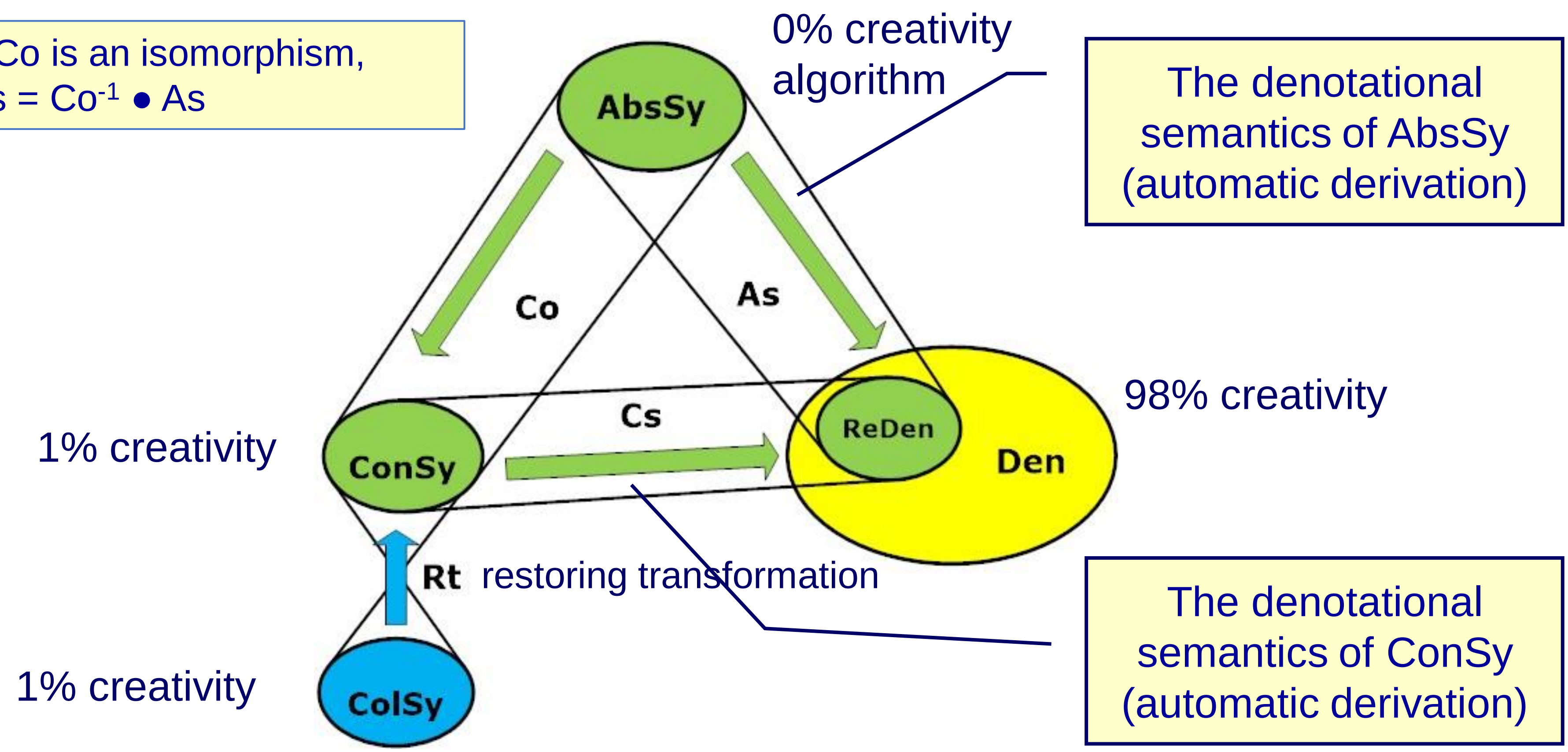
This requires a redefinition of our algebra of denotations

TRADITIONAL APPROACH

Expression = Component | Expression + Component
Component = Factor | Factor * Component
Factor = 1 | (Expression)

# A recapitulation of an algebraic model of a programming language

If Co is an isomorphism,
$Cs = Co^{-1} \bullet As$




Two steps of program execution:

1. restoring transformation
2. interpretation/compilation based on Cs.


Thank you for
your attention