Algebraic Abstract Data Types

Didier Buchs

Université de Genève

20 septembre 2018

Algebraic Abstract Data Types

- Informal introduction
- AADT Signature
- Terms with variables
- Equations and axioms
- Examples
- Graceful presentations

Formal and Mathematical basis

- Algebraic view
  - heterogeneous algebra (Birkhoff) = sets + operations
  - Logical view of their properties (Horn clauses)
- Computer science
  - Type = set of data + operations
  - Some code for describing the behavior of these types
- Support of an Abstraction point of view
  - Information hiding (realization hiding)
  - Functional approach (data hiding)

$\forall x, cond \Rightarrow predicat$

$pred_1 \wedge pred_2 \wedge \dots \wedge pred_n$

TAA = Type + propriétés

Informal example : Manipulation of strings

- Mandatory operations :
  - An empty string (new)
  - Concatenation of two strings (append)
  - Concatenation of one character to the string (add to)
  - Computation of the length (size)
  - Test of emptiness (isEmpty?)
  - Equality of two strings (=)
  - Selection of the first element (first)
- Necessary types for defining the string abstract data type :
  - character : the character AADT
  - natural : the type of the natural numbers
  - boolean : the type of the boolean values

$\mathbb{N}$

size : string → nat
new : → string
append : string , string → string
add_to : char , string → string

isEmpty : string → bool
= : string, string → bool

Signature

Definition of set of values and operations = signatures

- signatures
  - sorts names (or types)
  - operations names with profile (arity) nameofoperation : domain $->$ co-domain

```
Adt StringSpec;
   Interface
      sorts string, character, natural, boolean;
   Operations
       new: () -> string;
       append _ _: string, string -> string;
       add _ to _: character, string -> string;
       size _ : string -> natural;
       isEmpty? _ : string -> boolean;
       _ = _: string, string -> boolean;
       first _ : string -> character;
```

Remarks on the syntax : generalized prefix,infix and postfix notations

Prefix :

```
append _ _: string, string -> string;
```

constructible terms

```
append x y
append( x y )
(append x y)
```

Remarks on the syntax(2)

Infix :

```
_ = _: string, string -> boolean;
```

constructible terms

```
x = y
```

```
(x = y)
```

Remarks on the syntax(3)

Mixfix :

```
add _ to _: character, string-> string;
```

constructible terms

```
add append( x y) to c
add c to append( x y)
add first(x) to y
```

termes
$\Leftrightarrow$ expressions du langage ou les compositions des fonctions

Remarks on the signature

Terminology :

- string is the sort of interest
- character, natural et boolean are auxiliary sorts

Observation operations :

```
_ = _: string, string -> boolean;
size _ : string -> natural;
isEmpty? _ : string -> boolean;
first _ : string -> character;
```

Definition (Observer)

An observer is an operation with the profile :
interest sort and ev. auxiliary sorts $->$ auxiliary sort

Remarks on the signature(2)

Modifier operations :

```
new: () -> string;
add _ to _: character, string-> string;
append _ _: string, string -> string;
```

Definition (Modifier)

A modifier is an operation with the profile :
interest sort and ev. auxiliary sorts $->$ interest sort

A subclass of modifier is the operations generating all values of the domain.

Definition (Generator)

A generator is an operation with the profile :
interest sort and ev. auxiliary sorts $->$ interest sort

Boolean Type bool

Generators
t, f : → bool ;

operations
or, and : bool, bool → bool
not : bool → bool

axioms
not(t) = f
not(f) = t
} properties
...

Type nat

Generators
0 : → nat
S : nat → nat

operations
= : nat, nat → bool
+, −, *, / : nat, nat → nat

Definition of S-sorted set

We recall here some usual definitions.

$S = \{bool, nat, char, string\} \subseteq \mathbb{S}$

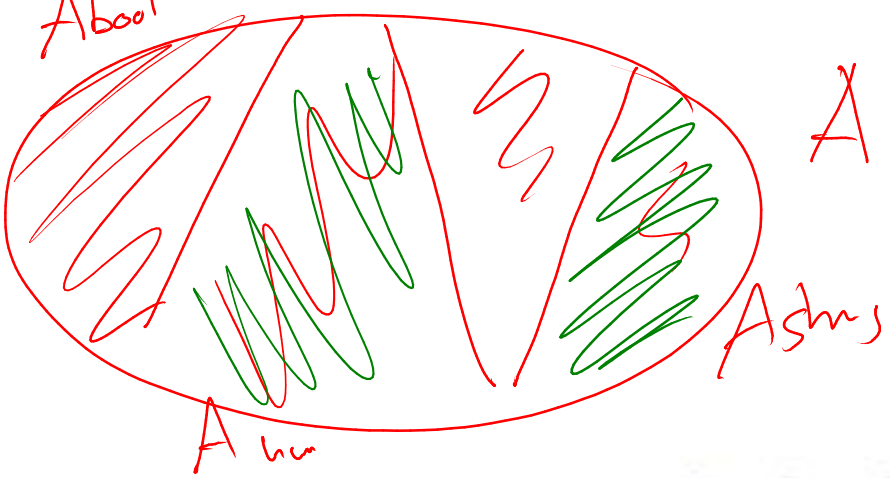
Definition (S-Sorted Set)

Let $S \subseteq \mathbf{S}$ be a finite set. A *S-sorted set A* is a disjoint union of a family of sets indexed by S ($A = \bigcup_{s \in S} A_s$), noted as $A = (A_s)_{s \in S}$.

Remark : In general this is a disjoint partition, for non-disjoint partition there is theory of ordered sorts.

Example :

$\mathbb{N} \subseteq \mathbb{Z}$



l'univers des nombres type

nat < integer

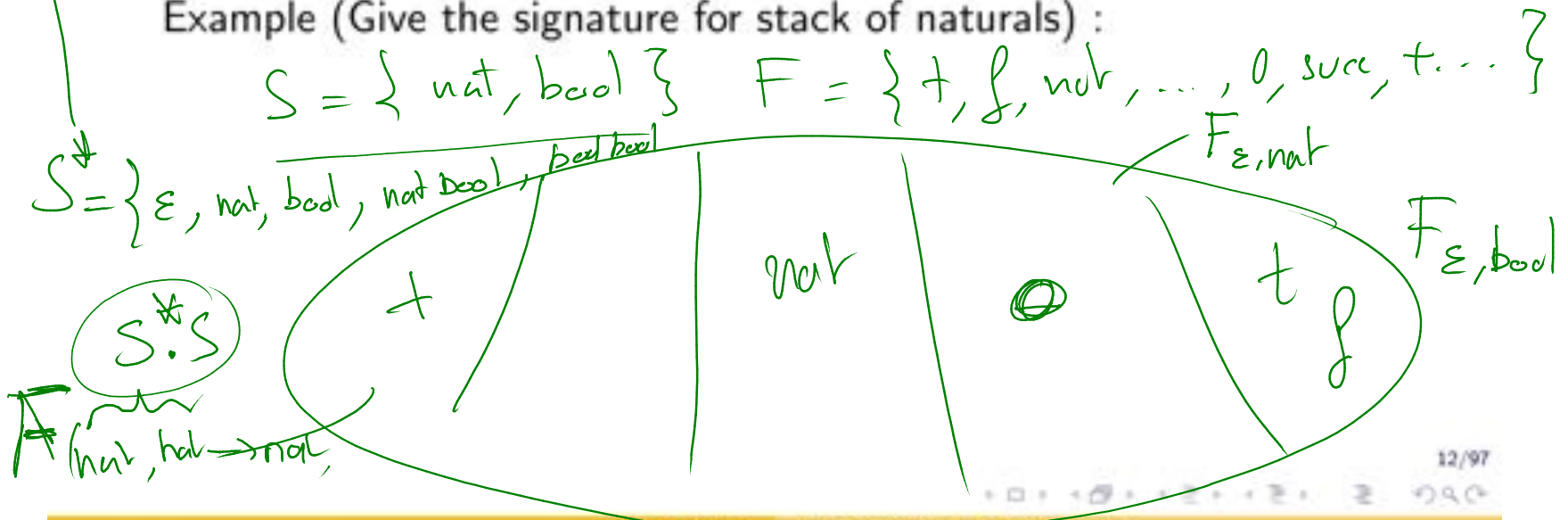
Definition of signature

Based on S-sets we have :

Definition (Signature)

A *signature* is a couple $\Sigma = \langle S, F \rangle$, where $S \subseteq \mathbf{S}$ is a finite set of sorts and $F = (F_{w,s})_{w \in S^*, s \in S}$ is a $(S^* \times S)$-sorted set of function names of $\mathbf{F}$. Each $f \in F_{\epsilon,s}$ is called a *constant*.

Example (Give the signature for stack of naturals) :



Definition of terms

Definition (Terms of a Signature)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted set of variables. The set of terms of Σ over X is a S-sorted set $T_{\Sigma,X}$, where each set $(T_{\Sigma,X})_s$ is inductively defined as follows :

- each variable $x \in X_s$ is a term of sort s, i.e., $x \in (T_{\Sigma,X})_s$
- each constant $f \in F_{\epsilon,s}$ is a term of sort s, i.e., $f \in (T_{\Sigma,X})_s$
- for all operations that are not a constant $f \in F_{w,s}$, with $w = s_1 \dots s_n$, and for all n-tuple of terms $(t_1 \dots t_n)$ such that all $t_i \in (T_{\Sigma,X})_{s_i}$ $(1 \leq i \leq n)$, $f(t_1 \dots t_n) \in (T_{\Sigma,X})_s$

What means this term ?

```
add c to x = append(x y)
append (IsEmpty(new),add x to c)
```

Definition of axioms

Definition (Axioms on variables)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted set of variables. The *axioms on variables* X are equational terms $t = t'$ such that $t, t' \in (T_{\Sigma,X})_{s,}$.

Example : x+0 = x

Remark : Variables are universally quantified

$X + 0 = X \iff \forall x \in \mathbb{N}, \; x + 0 = x$

size

word, c : char
s : string
r : string

size(new) = 0

size(add c to s) = size(s) + 1

" = s(size(s))

append(new, s) ≡ s

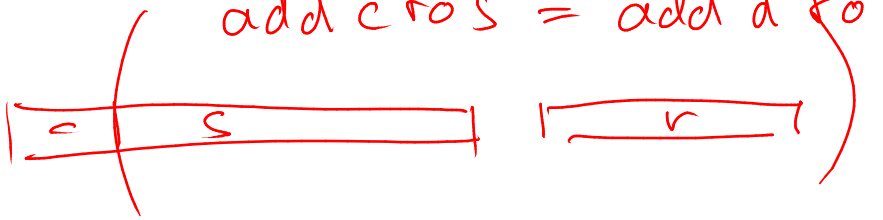
append(add c to s, r) ≡ add(c, append(s, r))

(new = new) ≡ t

(new = add(c to s)) ≡ f

(add c to s = new) ≡ f

add c to s = add d to r ≡ (c = d), (s = r)



String Axioms

```
Axioms
isEmpty?(new) = true;
isEmpty?(add c to x) = false;
#( new) = 0;
#(add c to x) = # (x) + 1;
append(new, x) = x;
append(add c to x, y) = add c to (append( x,y));
(new = new) = true;
(add c to x = new) = false;
(new = add c to x) = false;
(add c to x = add d to y) = (c = d) and (x = y);
+ axioms of first
Where
x,y:string; c,d:character;
End StringSpec;
```

type char

0 → char
1 → a
a → char
⋮
z → char

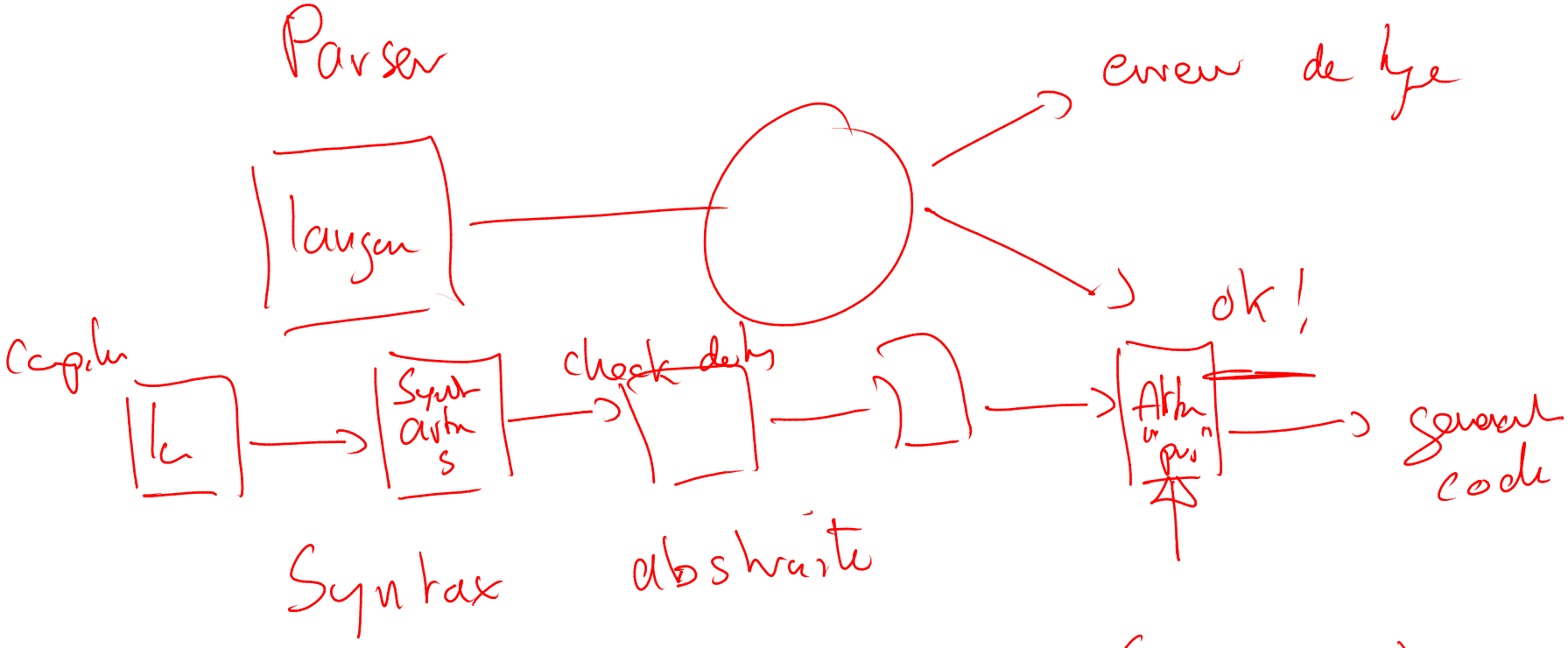
ASCII

operat
< : char, char → bool
ordre : char → nat
0 < 0 = false
0 < 1 = true
1 < 2
a < b =

String Axioms(2)

Be carefull ! ! : The symbol = is either a signature operator and a meta-operator of the basic logic of the specification language.
Signature of auxiliary sorts :

```
true: -> boolean;
false: -> boolean;
not _ : boolean -> boolean;
_ and_ ,_ or_ : boolean, boolean -> boolean ;
0: -> natural; 1: -> natural;
succ: natural -> natural;
_ + _ ,_ -_ ,_ *_ ,_ /_ : natural, natural -> natural ;
_ =_ : natural, natural -> boolean;
a: () -> character; b: () -> character;
....
_ =_ : character, character -> boolean;
```


Parser
lexer
erreur de type
ok !
Syntax
abstraite
Expressions correctes

Boolean Axioms

```
Adt Booleans;
Interface
Sorts boolean;
Operations
true , false : -> boolean;
not _ : boolean -> boolean;
_ and _ ,_ or _ , _ xor _ ,_ = _: boolean boolean -> boolea
Body
Axioms
not(true) = false; not(false) = true;
(true and b) = b; (false and b) = false;
(true or b) = true; (false or b) = b;
(false xor b) = b; (true xor b) = not(b);
(true = true) = true; (true = false) = false;
(false = true) = false; (false = false) = true;
Where b : boolean;
```

Exercise

Write the axioms of a sort Stack with the signature ;

```
Adt Stack;
Interface
Use Naturals, Booleans;
Sorts stack;
Operations
empty : -> stack;
push _ _: natural stack -> stack;
pop _ : stack -> stack;
top _ : stack -> natural;
_ = _ : stack stack -> boolean;
```

axioms ?

Exercise

Axioms of a sort Stack :

top(push e p) = e

top (empty) = 0

nil

nil → nat

mais implicite

nil + n ⇒ nil

" le problème general des fonctions partielles dans les langages de programmation soit — exception — en −1 en C

Conditional Axioms

Positive conditional axioms (Horn clause with equality) :

Definition (Axioms on variables)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted set of variables. The *conditional axioms on variables* X are $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \Rightarrow t = t'$ such that $t, t' \in (T_{\Sigma,X})_s, t_1, t'_1 \in (T_{\Sigma,X})_{s_1}, \ldots, t_n, t'_n \in (T_{\Sigma,X})_{s_n},$.

```
isEmpty(x)= false =>
first(add c to x) = first x;
isEmpty(x)= true =>
first(add c to x) = c;
```

Is it necessary ?

Graceful presentations

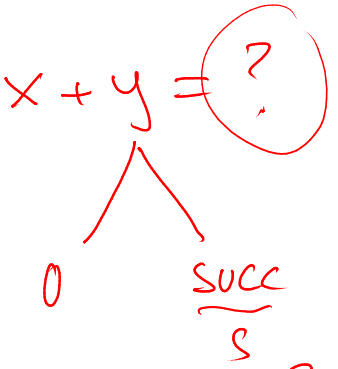
Graceful presentations It is a method for writing axioms without :

- the possibility of writing contradictory axioms
- forgetting cases.

Principle for each operation of the signature :

- Write on the left of the equation a term starting with the name of this operation.
- Iterate on all parameter of the operation the following principle from left to right :
  - Use a variable for this parameter
  - If it is not possible to write a valid axiom for this variable decompose using the generators
  - If a generator is not sufficient for the decomposition in sub case use conditions

General Property : sufficient completeness and hierarchical consistence are guaranteed



Example of axiomatisation

```
x + y = ?
```

decomposition of the second parameter with both constructors !

```
x + 0 = x;
x + succ(y) = succ(x+y);
```

Exercise : Application to

```
x > y = ?
```

$x > 0$

$0 > 0 = \text{false}$

$succ(y) > 0 = \text{true}$

$x > succ(y)$

$0 > succ(y) = \text{false}$

$succ(x) > succ(y) = x > y$

Example of axiomatisation : Sets of naturals

sort set

generators empty -> set
add nat, set -> set

operators union, inter, diff : set, set -> set
subset : set, set -> bool
in : nat, set -> bool
nbr : set -> nat
get : set -> nat
it+1 : set -> set

union(s, union(s', s'')) = union(s', union(s'', s))
union(s, union(s', s'')) = union(union(s, s'), s'')

union(s, s) = s ?

② a+(b+c) = (a+b)+c
assoc

? ! get doit être déterministe

① ⇒ ②
② ⇒ ① ?

axioms (entre generators)

add(n, add(m, s)) = add(m, add(n, s))

① add(n, add(n, s)) = add(n, s)

{1,3} ∪ {2} ordre d'insertion

ordre non important = {1,2} ∪ {3} non importe

$$union(s, empty) = s$$

$$union(s, add(n, s')) = add(n, union(s, s'))$$

$$nbr(empty) = 0$$

$$in(n, s) = true \Rightarrow nbr(add(n, s)) = nbr(s)$$

$$in(n, s) = false \Rightarrow nbr(add(n, s)) = S(nbr(s))$$

$get(s) = m \wedge in(m, s) \Rightarrow true$ $\forall n$

$in(n, s) = true \Rightarrow get(s) = n$

$get(add(n, empty)) = n$

⚠ au voisinage

~~$get(add(n, s)) = n$~~

$$get(add(n, add(m, empty))) = n$$

$$get(add(m, add(n, empty))) = n$$

$\Big\}$ $n = m$ $\forall n, m$

$0 = succ(0)$

$in(m,s) = true \wedge get(s) = m \Rightarrow true = true$

$\forall x \left(a(x) \Rightarrow b \right)$

$\forall x \left(\overline{a(x)} \vee b \right)$

$\left(\overline{\forall x \; a(x)} \right) \vee b$

$\exists x \; \overline{a(x)} \vee b$

$\overline{a \wedge b} = \bar{a} \vee \bar{b}$

Example of axiomatisation : Tables of naturals

index → ℕ

index, elem

sur générateurs

V[3] := 2

sort table

ops empty : → table

_[_] := _ : table, index, elem → table

opérateurs

size : table → nat

_[_] : table index → elem

sort : table → table

exist : table, index → bool

axiom ⇒ (t[i] := m)[j] := n = t[i] := n

(i=j) = true

(i=j) = false (t[i] := m)[j] := n = ((t[j] := n)[i] := m)

$$size(empty) = 0$$

$$exist(s,i) = true \Rightarrow size(s[i] := m) = size(s)$$

$$exist(s,i) = false \Rightarrow size(s[i] := m) = S(size(s))$$

$$exist(empty, i) = false$$

$$exist(s[i] := m, j) = (i = j) \vee exist(s, j)$$

$$exist(s, i) = exist(s, i)\ true(s)$$

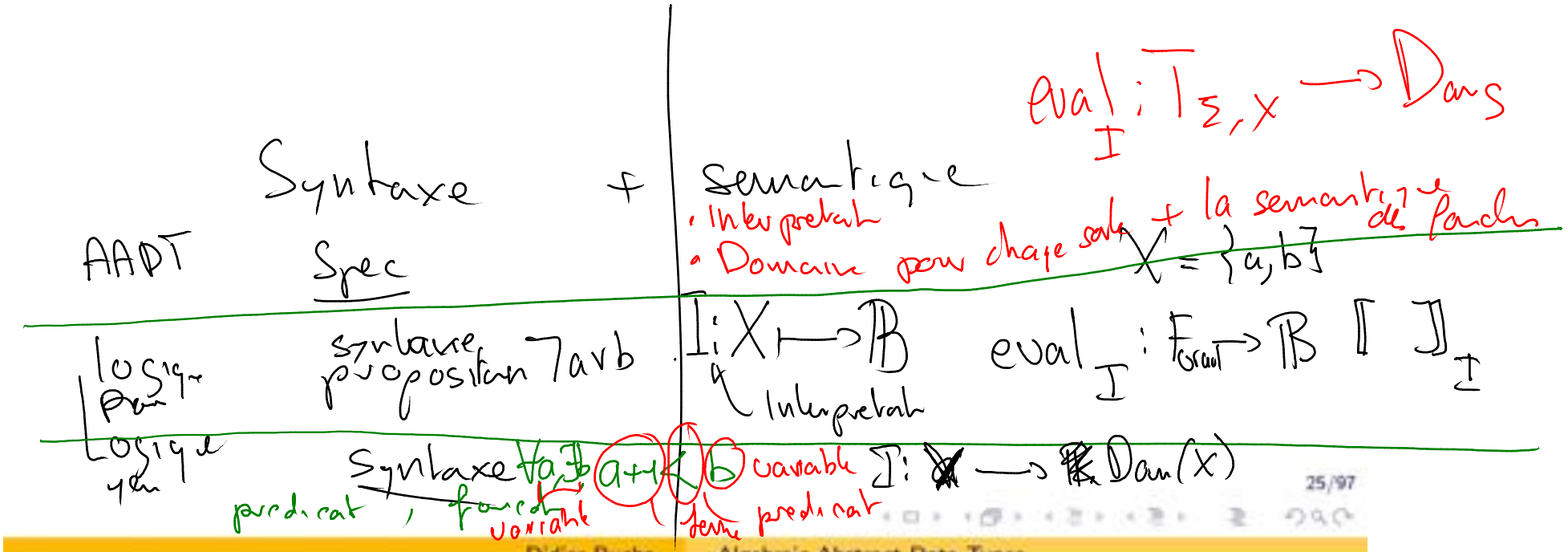
$$true(s) = exist(s, i)$$

Definition of algebraic specification

Definition (algebraic specification)

A *many sorted algebraic specification* $Spec = \langle S, F, X, AX \rangle$ is a signature extended by a collection of axioms E on variables X.

In what follows, let $\Sigma = \langle S, F \rangle$ be a complete signature.



Notion of models/Implementation

We can consider models i.e. structures that represents the semantics of the specification as possible implementations

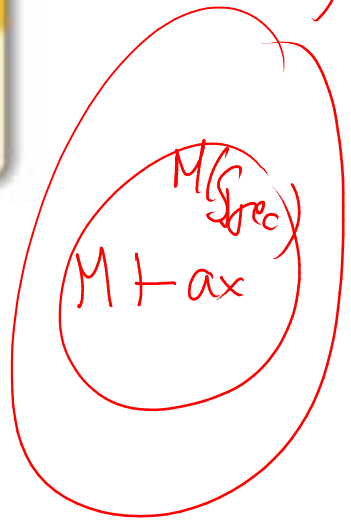
Definition (Models)

Given a specification $Spec =< \Sigma, X, AX >$, the class of its models is : $Mod(Spec)$ and $\forall ax \in AX, \forall M \in Mod(Spec), M \vdash ax$

A model is a set of value and operations called $\Sigma - algebra$, we will not detailed this notion.
It must be noted that there exist a unique 'morphism' $eval : T_\Sigma \rightarrow A$, where $A \in Mod(Spec)$, extended with interpretations $I : X \rightarrow A$ to a unique $eval_I$.

Definition (Satisfaction relation)

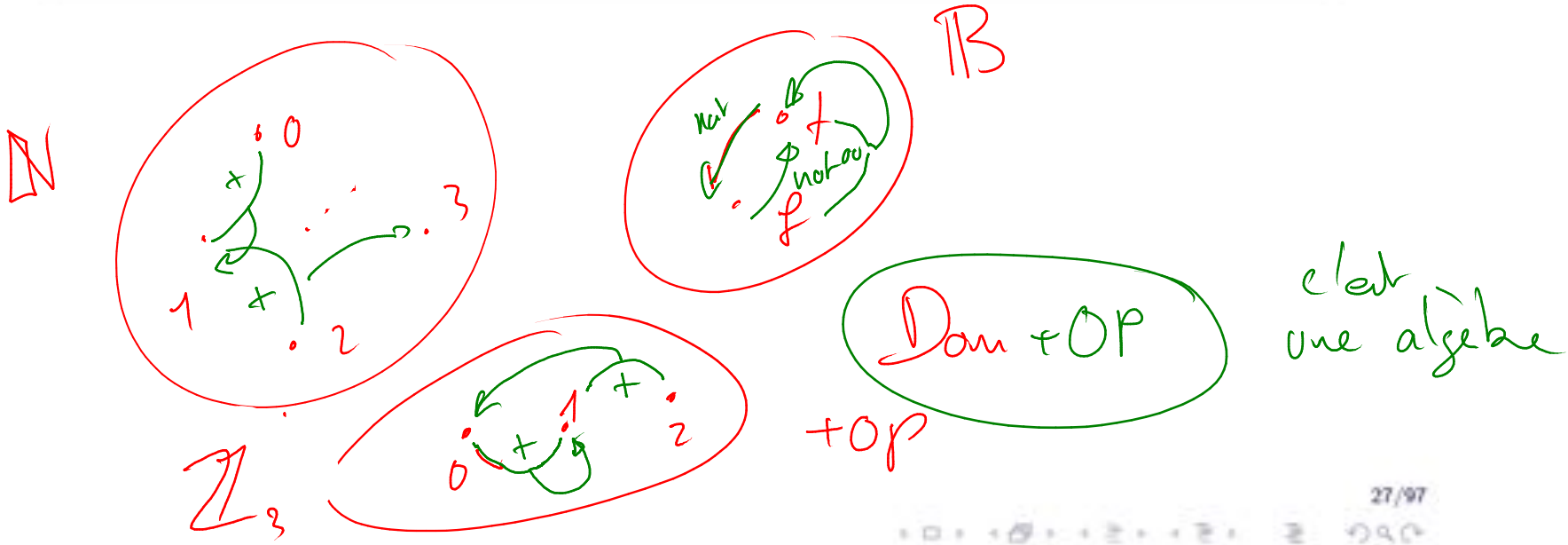
Given a specification $Spec =< \Sigma, X, AX >$, and $t_1, t_2 \in T_{\Sigma,X}, M \in Mod(Spec), M \vdash t_1 = t_2 \Leftrightarrow \forall I, eval_I(t_1) = eval_I(t_2)$



Definition of models

Definition (Σ-Algebra)

A Σ-algebra is a couple $A = \langle D, O \rangle$, in which D is a *S-sorted set* of values ($D = D_{s_1} \cup \ldots \cup D_{s_n}$) and O is a set of functions, such that for each function name $f \in O_{w,s}$, $w = s_1 \ldots s_n$ there is a function $f^A \in O$, defined as $f^A : D_{s_1} \times \ldots \times D_{s_n} \to D_s$.



Definition of model homomorphisms

Definition (Σ-Morphism)

Given $A = (D_A, O_A), B = (D_B, O_B)$ Σ-algebras. A Σ-Morphism is an application $\mu : A \to B$ st.
for all $f \in O_{w,s}, f \in O^A_{w,s}, f \in O^B_{w,s}, w = s_1 \dots s_n$,
$d_1, ..., d_n : d_1 \in D_{A,s_1}, ...d_n \in D_{A,s_n}$
$\mu(f^A(d_1, ..., d_n)) = f^B(\mu(d_1), ..., \mu(d_n))$.

eval is a Σ-Morphism

What about properties ?

If we consider the specification of naturals :

① `x + 0 = x;`

② `x + succ(y) = succ(x+y);`

`+ + succ(z) = succ(+ +z)`

What is :

`y + succ(succ(0)) = succ(succ(y))?`

and :

`succ(succ(0)) + y = succ(succ(y))?`

so, what about :

`x+y = y + x ?`

règle substitution

② subst = (x=y , y=succ(0)) à ②

⇒ y + succ(succ(0)) = succ(y + succ(0)) ③

(x=y , y=0) à ②

(y + succ(0)) = succ(y+0) ④

succ(y + succ(0)) = succ(succ(y+0))

substitutivité

a=b ⇒ succ(a) = succ(b)

③ ④

$t_1 = t_2 \wedge t_2 = t_3 \Rightarrow t_1 = t_3$ règle de transitivité

$y + succ(succ(0)) = succ(succ(y+0))$ ⑤

substitution

① $x + 0 = x$

$succ(succ(0)) = succ(succ(x))$ ⑥

transitivité ⑤ et ⑥ + substitution $(x = y)$

$y + succ(succ(0)) = succ(succ(y))$

Proofs in algebraic specifications :Plan

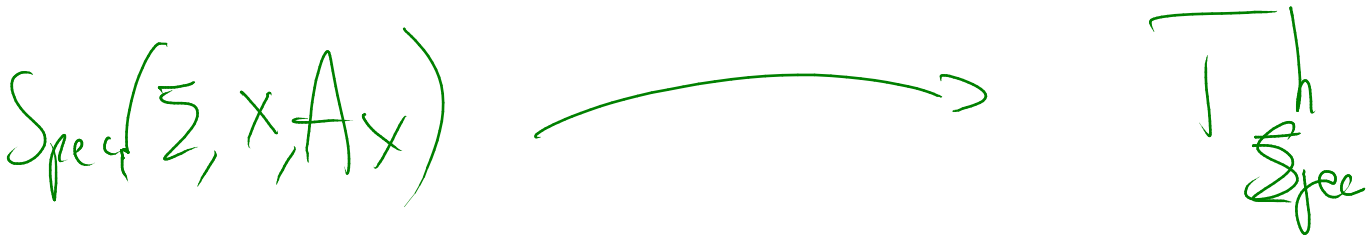
- Equational Proofs :
  - Equational theories
  - Inductive theories
  - Deduction systems
- Rewriting
  - Rewrite Systems
  - Properties of rewrite systems
  - Simulation by resolution, prolog translation
  - Constructors in rewriting

Proofs in algebraic specifications

Aim : Proof of specification properties in a systematic way. How to proceed from the axioms :

- Use equational rules (reflexivity,symmetry, transitivity)
- Use functional composition rules
- Use variable substitution rules

$\Rightarrow$ we obtain equational theorems



Example : Proofs in algebraic specifications

Axioms
$0 + x = x$
$succ(x) + y = succ(x + y)$

To prove : $succ(0) + succ(succ(0)) = succ(succ(succ(0)))$

axiom : $succ(x) + y = succ(x + y)$ **substitution** rule with $s = \{x = 0, y = succ(succ(0))\}$
(1) $succ(0) + succ(succ(0)) = succ(0 + succ(succ(0)))$

axiom : $0 + x = x$
substitution rule with $s = \{x = succ(succ(0))\}$
(2) $0 + succ(succ(0)) = succ(succ(0))$

Substitutivity rule with operation succ on (2)
(3) $succ(0 + succ(succ(0))) = succ(succ(succ(0)))$

Transitivity rule : (1) and (3) $\Rightarrow$
$succ(0) + succ(succ(0)) = succ(succ(succ(0)))$
CQFD

Limits of deduction theories

Problem : interesting theorems are more complex : $x + y = y + x$
Which is not deductible in the deduction system.

Inductive definition of equational theories

Given $Spec = < \Sigma, X, AX >$, $\Sigma = < S, OP >$ an algebraic specification
For any $t, t', t_i, t_i \in T_{\Sigma,s}$ the definition of $Th(Spec)$ is :

$$Axioms : t = t' \in Ax \Rightarrow t = t' \in Th(Spec)$$
$$Reflexivity : \forall t \in T_{\Sigma,s}, t = t \in Th(Spec)$$
$$Symmetry : t = t' \in Th(Spec) \Rightarrow t' = t \in Th(Spec)$$
$$Transitivity : t = t' \in Th(Spec) \wedge t' = t'' \in Th(Spec) \Rightarrow t = t'' \in Th(Spec)$$
$$Substitutivity : \forall f \in OP, t_1 = t'_1 \in Th(Spec) \wedge .. \wedge t_n = t'_n \in Th(Spec)$$
$$\Rightarrow f(t_1, t_2, ..., t_n) = f(t'_1, t'_2, ..., t'_n) \in Th(Spec)$$
$$Subst. : x \in X, u \in T_{\Sigma,s}, t = t' \in Th(Spec) \Rightarrow t[u/x] = t'[u/x] \in Th(Spec)$$
$$Cut : Cond_1 \wedge u = u' \wedge Cond_2 \Rightarrow t = t' \in Th(Spec)$$
$$and\ Cond \Rightarrow u = u' \in Th(Spec)$$
$$then\ Cond_1 \wedge Cond \wedge Cond_2 \Rightarrow t = t' \in Th(Spec)$$

Equational theory validity and completeness

Theorem (validity and completeness)

Given $Spec =< \Sigma, X, AX >$,
$\forall t_1, t_2 \in T_{\Sigma,X}$,
$t_1 = t_2 \in Th(Spec) \Leftrightarrow \forall M \in Mod(Spec), M \vdash t_1 = t_2$

This is the validity and completeness of the deduction

Theories and properties in implementations

$\forall M \in Mod(Spec), M \vdash t_1 = t_2$ indicates that the equation is valid in all implementations

Some equations are valid in only specific implementation (ex : true = false). This is for instance the case for very simple models such as the final one.

ex : dans $\mathbb{N}$

$\Updownarrow$ mêmes propriétés vraies

dans $\mathbb{Z}_3$ + $succ(succ(succ(0))) = 0$

propriétés vraies dans $\mathbb{Z}_3$ mais pas dans $\mathbb{N}$

Exercices :

prove : succ(succ(0)) - succ(succ(0)) = 0

prove : succ(succ(succ(0))) - succ(succ(0)) = 0

prove : x - x = 0

① $X - 0 = X$

$X - succ(y) =$

② $0 - succ(y) = 0$

③ $succ(x) - succ(y) = X - y$

- $0 - 0 = 0$ ok ! ①
- $succ(x) - succ(x) = 0$ ③ + sub$_3$ $y \Rightarrow x$

$x - x = 0 \Rightarrow$

$\forall x P(x) \Leftrightarrow P(0) \wedge (P(x) = P(succ(x)))$ $x+1$

Inductive theories

Aim : provide more general theorems deductible from the axioms.
Additional rule :

Definition (Induction rule)

Given G a formula such that x is a free variable,
If $\forall t, G[t/x] \in Th(Spec) \Rightarrow \forall x, G \in Th_{Ind}(Spec)$

Remark : $Th(Spec) \subseteq Th_{Ind}(Spec)$ and the induction rule define the inductive theories.

Inductive definition of inductive theories

Given $Spec =< \Sigma, X, AX >$, $\Sigma =< S, OP >$ an algebraic specification
For any $t, t', t_i, t_i \in T_{\Sigma,s}$ the definition of $Th(Spec)$ is :

$$Axioms : t = t' \in Ax \Rightarrow t = t' \in Th_{Ind}(Spec)$$
$$Reflexivity : \forall t \in T_{\Sigma,s}, t = t \in Th_{Ind}(Spec)$$
$$Symmetry : t = t' \in Th_{Ind}(Spec) \Rightarrow t' = t \in Th_{Ind}(Spec)$$
$$Transitivity : t = t' \in Th_{Ind}(Spec) \wedge t' = t'' \in Th_{Ind}(Spec) \Rightarrow t = t'' \in Th_{Ind}(Spec)$$
$$Substitutivity : \forall f \in OP, t_1 = t'_1 \in Th_{Ind}(Spec) \wedge .. \wedge t_n = t'_n \in Th_{Ind}(Spec) \Rightarrow f(t_1, t_2, ..., t_n) = f(t'_1, t'_2, ..., t'_n) \in Th_{Ind}(Spec)$$
$$Subst. : x \in X, u \in T_{\Sigma,s}, t = t' \in Th_{Ind}(Spec) \Rightarrow t[u/x] = t'[u/x] \in Th_{Ind}(Spec)$$

...

Inductive definition of inductive theories (cnt'd)

Given $Spec =< \Sigma, X, AX >$, $\Sigma =< S, OP >$ an algebraic specification
For any $t, t', t_i, t_i \in T_{\Sigma,s}$ the definition of $Th(Spec)$ is :

...

$$Cut : Cond_1 \wedge u = u' \wedge Cond_2 \Rightarrow t = t' \in Th_{Ind}(Spec)$$
$$and\ Cond \Rightarrow u = u' \in Th_{Ind}(Spec)$$
$$then\ Cond_1 \wedge Cond \wedge Cond_2 \Rightarrow t = t' \in Th_{Ind}(Spec)$$
$$Induction : x \in (X_s \cap (Var(t) \cup Var(t'))),$$
$$\bigwedge_{v_i \in (T_{\Sigma,X})_s} (t = t')[v_i/x] \in Th_{Ind}(Spec) \Rightarrow t = t' \in Th_{Ind}(Spec)$$

règle d'induction

$X + y = y + X$

How to prove such often infinite conjunction of specific proofs ?

By structural induction on the generators :

- it can be done rather informally in a natural deduction style.
- Formally using judgment, from sequent calculus, of the form. $t_1 = t_2 \vdash t_1 = t_2[f(x)]$, where f is a generator.

validity and completeness

Theorem (validity and completeness)

Given $Spec =< \Sigma, X, AX >$,
$\forall t_1, t_2 \in T_{\Sigma,X}$,
$t_1 = t_2 \in Th_{Ind}(Spec) \Leftrightarrow \forall M \in Mod_{Gen}(Spec), M \vdash t_1 = t_2$

This is the validity and completeness of inductive theories for finitely generated models, i.e. models where all values are reachable from a syntactic term (eval is surjective)

valeurs sont représentables par des générateurs

Example of induction

Given the boolean specification with operations : true, false et not
We want to deduce : not(not(b)) = b
The predicate defining the property can be written :
P(b) = not(not(b)) = b
Proof :
Base cases : P(true) = not(not(true)) = not(false) = true ;
P(false) = not(not(false)) = not(true) = false ;
Induction Step : not(not(b)) = b implies not(not(not(b))) = not(b)
substitutability rule with 'not' applied to not(not(b)) = b
implies not(not(not(b))) = not(b)
It must be noted that having finitely generated models wrt the generators allow to only use generators in the proofs.

Exercise

Prove the commutativity of addition within naturals with 0, succ, + and the axioms

$x+0 = x$

$x + \text{succ } y = \text{succ } (x + y)$:

$P(x,y) = x + y = y + x$

Exercise ctn'd

Properties in initial and final models

The models can be ordered following an order relation :
$M_1 \leq M_2 \Leftrightarrow Th(M_1) \subseteq Th(M_2)$
Where $Th(M) \Leftrightarrow \{t_1 = t_2 | t_1, t_2 \in T_{\Sigma,X}$ and $M \vdash t_1 = t_2\}$

- The order relation forms a lattice for Horn clauses.
- All equalities between close terms valid in the initial models are valid in the other models (The lower model).
- Initial model $\Rightarrow$ minimal set of equalities between close terms. There is only one initial model for Horn clause theories.
- For each equality valid in the final model there exists a model different from the initial model for which this equality is true.
- final model $\Rightarrow$ maximal set of equalities between close terms, i.e such that $\forall t_1, t_2 \in T_{\Sigma,X}$, $Triv \vdash t_1 = t_2$

Example : $Triv_{Bool} \vdash true = false$

Contradiction and incompleteness

What is happening if contradiction occurs?

P

$f : D \to D'$ $P \wedge \neg P$ $f(t) = f(t')$?

What is happening if incomplete definitions are given?

B

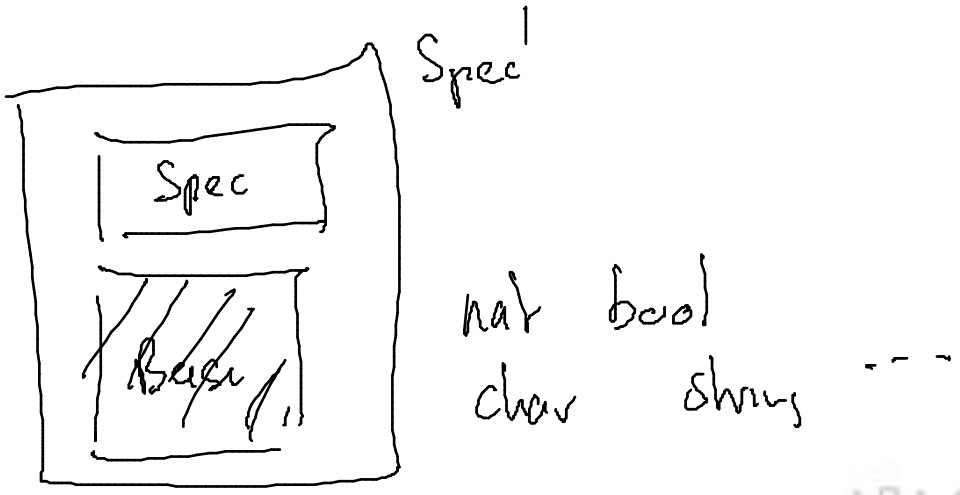
not (true) = false

not (false)

In fact these notions are not well defined!

Hierarchies in Algebraic Specifications

- Deal with progressive development of systems
- Flat specification implies no bound on the effect of axioms.
- There is no isolation mechanism in standard logic.
- Need to isolate properties ⇒
  no perturbation over hierarchical levels when using specifications.



Hierarchical Specification

- Hierarchical constraints ⇒ reflect decomposition of the process of building specifications.
- The modules of the specification should be implemented separately.
- Kind of perturbations :
  - junk : values are added when a module is extended ⇒ sufficient completeness.
  - confusion : values are collapsed when extending a module ⇒ hierarchical consistency.

Hierarchical models

HMod(Spec) s.t. Spec = ΔSpec + Spec0
The restriction (forget) to sub-modules will preserve their semantics.[1] The semantics of the ground module is chosen as an initial semantics (for ex. booleans with true ≠ false)

Definition (Hierarchical models)

$HMod(Spec) = \{m \in Mod(Spec) | U(m) \in Mod(Spec0)\}$

1. The forgetful functor is noted U.

Example :Suffisent Completeness

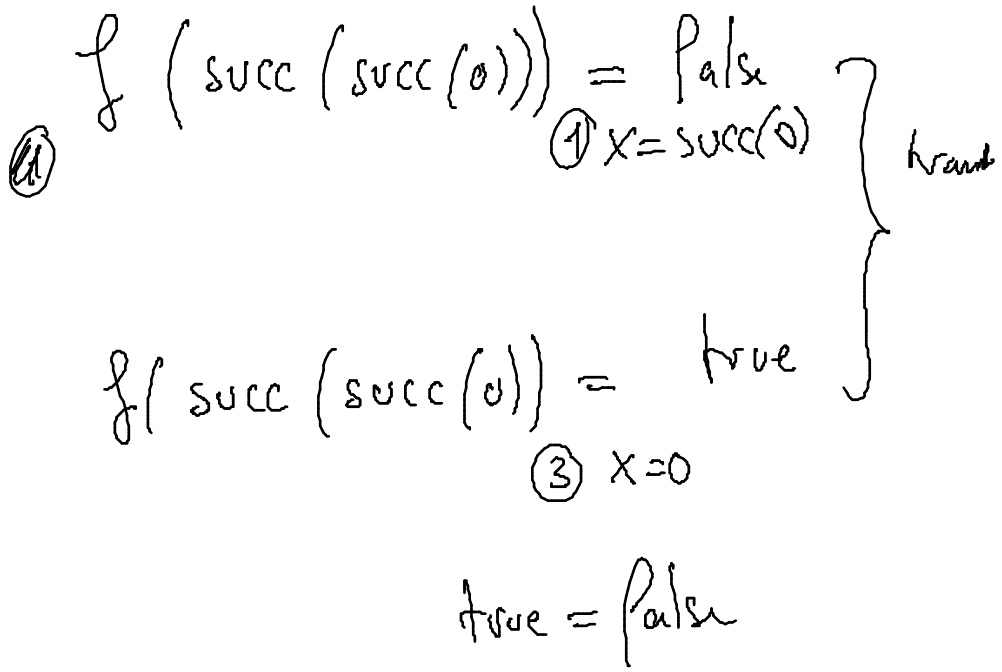
```
Adt Passuffcomplet;
Interface
  Use Naturals,Booleans;
  Operations
     f : natural-> boolean;
   Body
     Axioms
         f(succ(x)) = false;
      Where x: natural;
End Passuffcomplet;
```

⇒ problem : a new value exists : f(0) of type boolean in the ~~initial~~ ~~model~~.

not(f(0)) , and(true , f(0)) ---- ∈ bool

Example : hierarchical Consistency

```
Adt Pasconsistant;
Interface
  Use Naturals,Booleans;
  Operations
    f : natural-> boolean;
  Body
     Axioms
   (1)  f(succ(x)) = false;
   (2)  f(0) = true;
   (3)  f(succ(succ(x)) = true;
    Where x: natural;
End Pasconsistant;
```



⇒ problem : we have now that true = false in the initial model.

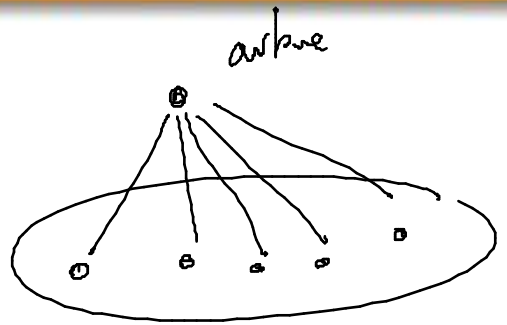
Other algebraic specification formalisms

Various extensions of the basic presented approach :

- Partial functions ⇒ definition predicate
- Sub-sort definitions ⇒ inclusion relation between algebras
- Exceptional cases ⇒ exceptions as labels on values
- and : State algebras, concurrency, bounds

$\mathbb{N} \subseteq \mathbb{Z}$

Exercise : Modeling Trees (1)



sort tree

sort foreest

balanced tree

tree

axioms

$\forall s_{sn_i, sn_j} \in n \; / \; \left| taille(sn_i^n) - taille(sn_j^n) \right| \leq 1$

Exercise : Modeling Trees (2)

Non determinism in implementation

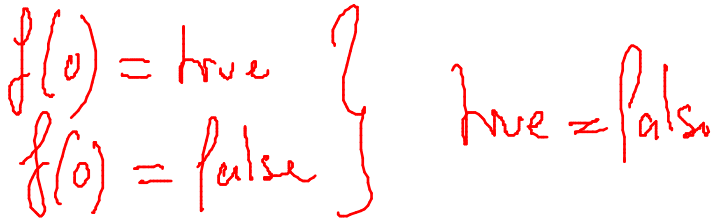
Naive approach to define non-deterministim in implementation :

```
Adt NaiveNondeterminism;

f: natural -> boolean;

Axiom
  f(x) = b;

  Where x:natural; b : boolean;
```



What are the resulting properties of such specification ?
x, b are universally quantified $\Rightarrow$ i.e. $f(0) = true$ and
$f(0) = false \Rightarrow true = false$
Which is not a valid hierarchical model

Correct non determinism in implementation

Consistent non-deterministim in implementation :

```
Adt Nondeterminism;
f: natural -> boolean;
Axiom
  f(x) = b => ((b = true) or (b=false)) = true;

  Where x:natural; b : boolean;
```

What are the resulting properties of such specification ?
x, b are universally quantified $\Rightarrow$ but in the condition it is existentially quantified .
As the functions are deterministic, a correct model has a 'f' function which choose a boolean value.
Exercise :
define the choose function in sets.

choose : $S \rightarrow v \in S$

Rewriting

Rewriting is a technique to :

- automate proofs
- compute terms evaluation,
- do prototyping

Principle of proof of properties :
Orientation of equations ⇒ rewrite rules

$a = b$

$a \to b$
$b \to a$

deux possibilité

Problems :

- which direction, is the set of rewriting rules complete?
- termination
- confluence

Introduction to rewriting principles

- From axiom **not true = false** , we can built the rewrite rule $not(true) \leadsto_1 false$
- Abstract rewriting system is a proof view of the rewriting process (as opposed to its operational view) defined as relation between equivalent terms.
- Operational mechanism used to do the rewriting process :
  - filtering = choice of the rule by matching the left term to the term to rewrite.
  - substitution of the matching part with the specialised right part of the rule
- Closure of the rewrite rules to build a normal form (irreductible form)

Abstract Rewrite Systems

Definition (Abstract rewrite system)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted set of variables.

- An abstract term rewriting system (ARS) is $A = (T_{\Sigma,X}, \rightarrow)$, where : $\rightarrow \subseteq T_{\Sigma,X} \times T_{\Sigma,X}$

A rewrite step $l \rightarrow r \in \rightarrow$ can be completed by the already defined deduction principles. We would like to omit the rules that can introduce non terminating process (for instance symmetry and reflexion)

Closure of Abstract Rewrite Systems

Definition (Closure of Abstract Rewrite system)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted set of variables and an abstract term rewriting system (ARS) $A = (T_{\Sigma,X}, \rightarrow)$, where : We define $Closure(A) = (T_{\Sigma,X}, \rightarrow^*)$: an abstract rewrite system obtained by applying the following rules.

- $\forall \sigma, (t, t') \in \rightarrow \Rightarrow (\sigma t, \sigma t') \in \rightarrow^*$
- $\forall f \in \Sigma, (t_i, t_i') \in \rightarrow^* \Rightarrow (ft_1, ..., t_n, ft_1', ..., t_n') \in \rightarrow^*$
- $\forall (t, t') and (t', t'') \in \rightarrow^* \Rightarrow (t, t'') \in \rightarrow^*$

From the rewrite rules, a rewrite relation can be computed as extension of the effect of all rewrite rules, according to variable *substitution* and encapsulation in function application (*substitutivity*).

Closure of Abstract Rewrite Systems

Definition (Rewrite rules)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted set of variables.

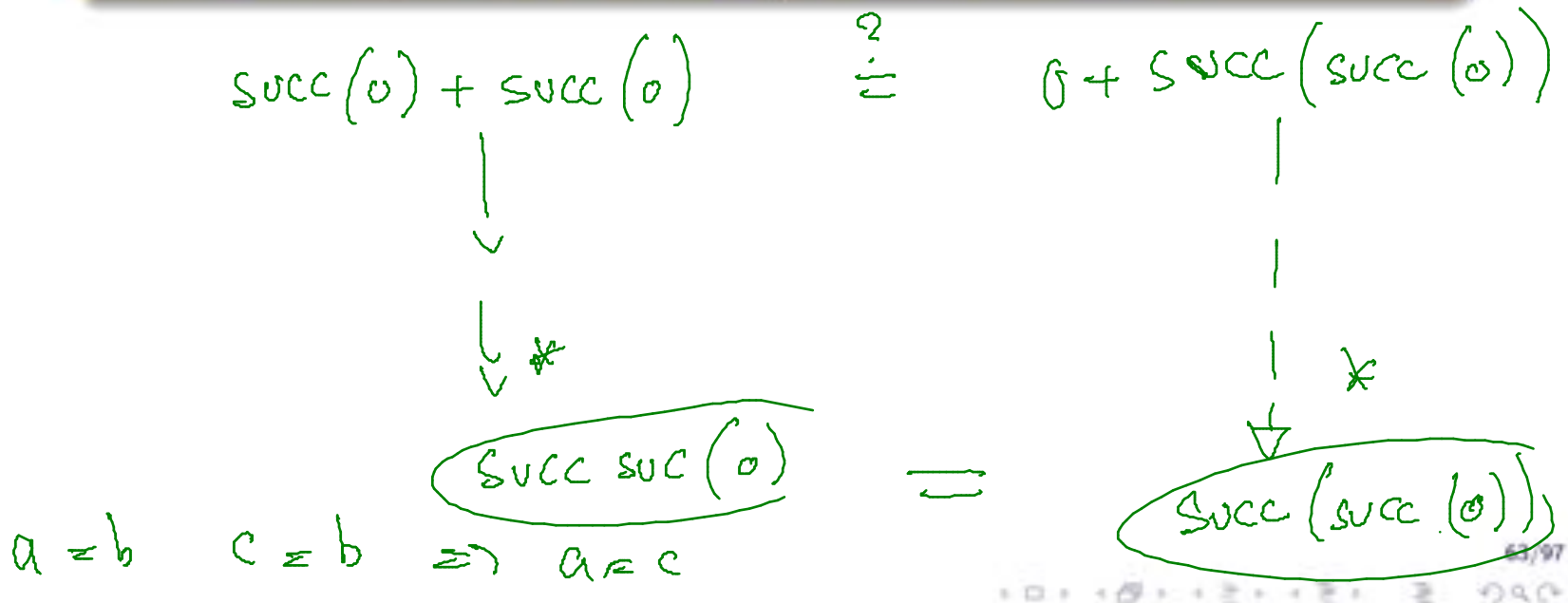
- We note $Rew_{\Sigma,X} \subseteq T_\Sigma(X) \times T_\Sigma(X)$ a set of rewrite rules for a given signature and variables.

A rewrite rule $l \rightsquigarrow r$ can be derived from axioms $l = r$ by just taking the left and right part of the equality. In general this is not sufficient.

Proof of equalities

Definition (Rewrite theories)

Given $Spec =< \Sigma, X, AX >$ and an abstract term rewriting system $A = (T_{\Sigma,X}, \rightarrow)$, and its closure :
$\forall t_1, t_2 \in T_{\Sigma,X}$,
$t_1 = t_2 \in Th_{\rightarrow}(Spec) \Leftrightarrow \exists t \in T_{\Sigma,X}, t_1 \rightarrow^* t \wedge t_2 \rightarrow^* t$



Operational Rewriting of terms

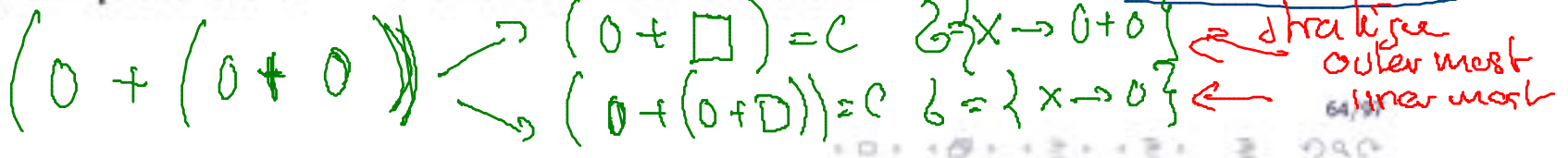
Definition (Rewrite step)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted set of variables and $l \rightsquigarrow r$,$l, r \in T_\Sigma(X)$ a rewrite rule.

- $filter(t, l) =< \sigma, c > \Leftrightarrow \exists \sigma \in S \; \exists c$,
  - $t = c[\sigma l]$ [a]
  - $t' = c[\sigma r]$
- $< t, t' > \in Rew_{l \rightsquigarrow r}$ a rewrite step

a. $c[\_]$ denotes the context of a term, i.e. a term with a place holder

Considering $\rightsquigarrow^*$ the transitive closure of $\rightsquigarrow$, they are supposed to be confluent and with finite termination, i.e
$\forall t, \exists$ unique e s.t $t \rightsquigarrow^* e$ and e is not reducible.



Context

Definition (Context and Subterms)

Let $\Sigma = \langle S, \leq, F \rangle$ be an order-sorted signature and X be a S-sorted variable set, let also $\Box \notin F \cup X$ be a special constant symbol called a placeholder.

- A context C of a term $t \in T_{\Sigma,X}$ is a term $(T_{\Sigma \cup \Box,X})_s$
- if $C_t[\Box_1, \ldots, \Box_n]$ is a context with n occurrences of $\Box$ and $t_1, \ldots, t_n$ are terms $\in (T_{\Sigma \cup \Box,X})_s$, then $C_t[t_1, \ldots, t_n]$ is the result of replacing the $\Box_i$ by the t_i.
- A term $st \in (T_{\Sigma,X})_s$ is a *subterm* of $t \in (T_{\Sigma,X})_s$ noted $st \subseteq t$ if there exists a context C of term t denoted $C_t[\Box]$ such that $t = C_t[st]$.

Example : $t = suc(suc(suc(0)))$, $C_t = suc(suc(\Box))$, $st = suc(0)$.

Substitution

Definition (Substitution)

Let $\Sigma = \langle S, F \rangle$ be a signature and X be a S-sorted variable set. A substitution σ is mapping $\sigma : X_s \to (T_{\Sigma,X})_s$ where $s \in S$. Every substitution σ extends uniquely to a morphism $\sigma^{\#} : (T_{\Sigma,X})_s \to (T_{\Sigma,X})_s$, where $s \in S$

- $\sigma^{\#}(f(t_1, \ldots, t_n)) = f(\sigma^{\#}(t_1), \ldots, \sigma^{\#}(t_n)))$
- $\sigma^{\#}(f_s) = f_s$ with $f_s \in F_{\epsilon,s}$
- $\sigma^{\#}(x_s) = \sigma(x_s)$

Example : $\sigma : \{x_s \to a_s; y_s \to b_s\}$ with $x_s, y_s \in X_s$, a_s, $b_s \in F_{\epsilon,s}$. $t = f(x_s, y_s)$, $\sigma^{\#}(t) = f(a_s, b_s)$.

Example of rewriting in algebraic specification

We will provide an example of algebraic specification in order to illustrate rewriting issues.
The example will cover several simple sorts and simple axioms interdependant to each other.

$S = \{nat, bool\}$

$OP = \{+ : nat, nat \rightarrow nat, 0 :\rightarrow nat, suc : nat \rightarrow nat, 1 :\rightarrow nat$
$not : bool \rightarrow bool, true :\rightarrow bool, false :\rightarrow bool, >: nat, nat \rightarrow$
$bool\}$

$X_{nat} = \{x, y, z\}$ and $X_{bool} = \{a, b\}$

The axioms are :

$x + 0 = x\,;\ x + suc(y) = suc(x + y)\,;\ 1 = suc(0)$

$not(true) = false\,;\ not(false) = true$

$0 > x = false\,;\ (suc(x) > 0) = true\,;\ (suc(x) > suc(y)) = x > y$

Example of rewriting algebraic specification terms(2)

The rewrite rules are :

$x + 0 \leadsto_1 x$;

$x + suc(y) \leadsto_2 suc(x + y)$;

$1 \leadsto_3 succ(0)$

$not(true) \leadsto_4 false$;

$not(false) \leadsto_5 true$

$0 > x \leadsto_6 false$;

$(suc(x) > 0) \leadsto_7 true$;

$(suc(x) > suc(y)) \leadsto_8 x > y$

Example of rewriting algebraic specification terms(3)

Rewriting the terms can be computed as follows[2] :

- $1 > 0 \rightsquigarrow_3 suc(0) > 0 \rightsquigarrow_{7,x=0}$ **true**
- $1 + 1 \rightsquigarrow_3 suc(0) + 1 \rightsquigarrow_3 suc(0) + suc(0) \rightsquigarrow_{2,x=suc(0),y=0}$ $suc(suc(0) + 0) \rightsquigarrow_{1,x=suc(0)}$ **suc(suc(0))** [3]
- $(suc(1) + 1) > 1 + 1 \rightsquigarrow_3 (suc(suc(0)) + 1) > 1 + 1 \rightsquigarrow_3$ $(suc(suc(0)) + suc(0)) > 1 + 1 \rightsquigarrow_{2,x=suc(suc(0)),y=0}$ $suc(suc(suc(0)) + 0) > 1 + 1 \rightsquigarrow_1 suc(suc(suc(0))) >$ $1 + 1 \rightsquigarrow_{3/3/2/1}$ [4] $suc(suc(suc(0)) > suc(suc(0)) \rightsquigarrow_{8/8}$ $suc(0) > 0 \rightsquigarrow_7$ **true**

forme normale

stratégie ?

2. bold terms are canonical terms
3. reuse of already evaluated terms
4. reuse of several reductions

Proof of equalities

Definition (Rewrite theories)

Given $Spec =< \Sigma, X, AX >$ and a set of rewrite rules defining the relation $\rightsquigarrow$
$\forall t_1, t_2 \in T_{\Sigma,X}$,
$t_1 = t_2 \in Th_{\rightsquigarrow}(Spec) \Leftrightarrow \exists t \in T_{\Sigma,X}, t_1 \rightsquigarrow^* t \wedge t_2 \rightsquigarrow^* t$

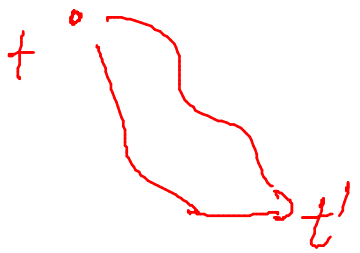
Theorem (abstract and operational rules are identical)

Given $Spec =< \Sigma, X, AX >$ *and a set of rewrite rules defining the relation* $\rightarrow$ *and* $\rightsquigarrow$.
$Th_{\rightarrow}(Spec) \Leftrightarrow Th_{\rightsquigarrow}(Spec)$

Properties of rewrite rules

Convergence, confluence of a rewrite system :

- Property to reach for all terms a unique normal form, without taking care of the strategy.



Termination of a rewrite system :

- Property to reach for any terms, in a finite number of steps a normal form.

The use of graceful presentation will help in finding a good rewrite system.

a → b
b → a
} termine pas

Operational view :

Rewriting = rules + application mechanism + strategy

rewrite step

Possible strategies :

- left-right-inner-most
- left-right-outer-most

There is no optimal strategy (see in the book Rewrite systems, Jouannaud, Dershowitz p. 39).

Strategies : Operational rewriting a la TOM

Based on elementary rewrite rules, we can apply on terms a basic rewrite step.
$Rew_{Ax}[t] : T_\Sigma \rightarrow (T_\Sigma \cup \{fail\})$

$\exists \sigma,$
$(\sigma(l) = t) \Rightarrow Rew_{Ax \cup \{<l,r>\}}[t] = \sigma(r)$
$Rew_{Ax}[t] = fail$ otherwise

This is the application of the rule at the root of the term.
Strategies can fail if there is no possible rule application.
We use, in the tools, an order on the rules to provide with deterministic behaviours.

Implementation of Strategies

Way to find the context of a rewriting step !

$$Strat(S) : (T_\Sigma \cup \{fail\}) \rightarrow (T_\Sigma \cup \{fail\})$$

If $Strat(s)$ is defined, terms t will be rewritten with :

$$Strat(Rew_{Ax})[t]$$

Obviously :

$$(S)[fail] = fail$$

Strategies : Basic operations 1 (TOM)

$$(Identity)[t] = t$$
$$(Fail)[t] = fail$$
$$(Sequence(s1, s2))[t] = fail \Leftarrow (s1)[t] = fail$$
$$(Sequence(s1, s2))[t] = (s2)[t'] \Leftarrow (s1)[t] = t'$$

$$(Choice(s1, s2))[t] = t' \Leftarrow (s1)[t] = t'$$
$$(Choice(s1, s2))[t] = (s2)[t] \Leftarrow (s1)[t] = fail$$

Strategies 2

$$(All(s))[f(t1,...,tn)] = f(t1',...,tn')$$
$$\Leftarrow (s)[t1] = t1', ..., (s)[tn] = tn'$$
$$(All(s))[f(t1,...,tn)] = fail \Leftarrow \exists i, (s)[ti] = fail$$
$$(All(s))[cst] = cst$$

$$(One(s))[f(t1,...,tn)] = f(t1,...,ti',...,tn)$$
$$\Leftarrow (s)[ti] = ti'$$
$$(One(s))[f(t1,...,tn)] = fail$$
$$\Leftarrow (s)[t1] = fail, ..., (s)[tn] = fail$$
$$(One(s))[cst] = fail$$

One is non deterministic ! It is not a functional strategy.

TOM : Strategies Library

μ is the recursion operator.

$$Try(s) = Choice(s, Identity)$$
$$Repeat(s) = \mu x.Choice(Sequence(s, x), Identity())$$
$$OnceBottomUp(s) = \mu x.Choice(One(x), s)$$
$$BottomUp(s) = \mu x.Sequence(All(x), s)$$
$$TopDown(s) = \mu x.Sequence(s, All(x))$$
$$Innermost(s) = \mu x.Sequence(All(Innermost(x)), Try(Sequence(s, x)))$$

Recursion

Fixpoint solution ! !
$eval(\mu x.t) = eval(\mu x.\sigma(t))$ with $\sigma(x) = t$

Formally it is an infinite possible application of the t pattern.

$Repeat(s) = \mu x.Choice(Sequence(s, x), Identity())$
$Repeat(s) =$
$\mu x.Choice(Sequence(s, Choice(Sequence(s, x), Identity())), Identity())$
$Repeat(s) =$
$\mu x.Choice(Sequence(s, Choice(Sequence(s, Choice(Sequence(s, x), Identity()))), Identit$

Operationnaly, it can be evaluated with lazy procedure.

Rewrite system vs. strategies

Using strategies we can define the previously constructed rewrite operations :
Given a set of rewrite rules Rew and its rewrite relation $\leadsto^*$.
$t \leadsto^* t' \Leftrightarrow Innermost(Rew)[t] = t'$

Strategies in Prolog : strategies

```
strataxiom(try(S),choice(S,identity)).
strataxiom(repeat(S),choice(sequence(S,repeat(S)),identity)).
strataxiom(bottomup(S),sequence(all(bottomup(S)),S)).
strataxiom(topdown(S),try(sequence(S,all(topdown(S))))).
strataxiom(innermost(S),sequence(all(innermost(S)),
                        try(sequence(S,innermost(S))))).
identity(T,T).
fail(T,fail).

sequence(S1,S2,T,R):-eval(S1,T,R1),
                      (R1=fail,!,R=fail;
                       eval(S2,R1,R)).
choice(S1,S2,T,R):-eval(S1,T,R1),
                    (R1=fail,!,eval(S2,T,R);
                     R=R1).
all(S,T,R):- T=..[FCT|LP],listeval(S,LP,LR),
          (LR=fail,!,R=fail;R=..[FCT|LR]). /* treatment of f.
```

Strategies in Prolog : evaluation

```
/* application of rules
rules from library (last eval rule) can be compiled if more
efficiency is needed (add two parameters
 systematically for terms)*/

eval(axiom,T,R):- (axiom(T,R),!;R=fail),!. /*must be determi
eval(identity,T,R):-identity(T,R),!.
eval(fail,T,R):-fail(T,R),!.
eval(sequence(S1,S2),T,R):-sequence(S1,S2,T,R),!.
eval(choice(S1,S2),T,R):-choice(S1,S2,T,R),!.
eval(all(S),T,R):-all(S,T,R),!.
eval(S,T,R):- strataxiom(S,CORPUS),print((S,T)),nl,
                        eval(CORPUS,T,R).

listeval(S,[],[]).
listeval(S,[T|LP],RES):-
            eval(S,T,R),(R=fail,!,RES=fail;listeval(S,LP,LR),
            (LR=fail,!,RES=fail;RES=[R|LR])).
```

Strategies in Prolog : axioms

```
/* atomic rewrite rules*/

axiom(X+0,X).
axiom(X+s(Y),s(X+Y)).

/*test queries*/

eval(innermost(axiom),s(s(0))+s(0),R).

eval(innermost(axiom), s(s(0)), R).

eval(topdown(axiom), 0, R).

eval(innermost(axiom),s(s(0))+s(s(0)),R).
```

Problems with Rewriting :

- The equality induced by the rewriting process is not the same as the one deduced from the axioms.
- We would obtain a equivalent system generated from the axioms (its not a decidable problem in all generality)
- Solution by orienting the equations, if the resulting system is confluent and terminate it is equivalent to the initial axioms.

Example of rules for boolean

Orientation from left to right :

- 1) $not(true) \rightsquigarrow false$;
- 2) $not(false) \rightsquigarrow true$;
- 3)$(true\ and\ b) \rightsquigarrow b$;
- 4) $(false\ and\ b) \rightsquigarrow false$;
- 5) $(true\ or\ b) \rightsquigarrow true$;
- 6) $(false\ or\ b) \rightsquigarrow b$;
- 7)$(false\ xor\ b) \rightsquigarrow b$;
- 8) $(true\ xor\ b) \rightsquigarrow not(b)$;

Given a term :

Example : weakness of orientation

```
 sort truc
Operations
    0: -> truc; +: truc truc -> truc; - : truc -> truc;
Axioms
  ax1: 0 + x = x
  ax2: x + (-x) = 0
```

Necessary rewrite rules :

- $0 + x \rightsquigarrow x$
- $x + (-x) \rightsquigarrow 0$
- $-0 \rightsquigarrow 0????$

Orienting is no sufficient i.e. -0 = 0, the proof need ax1 from right to left and axiom 2 from left to right.

Termination

This is the property that for all terms there exist a normal form.
Example : Given the rewrite system, a,b constants, f,g functional symbols and x,y variables :

- 1) $f(a, b, x) \rightsquigarrow f(x, x, x)$;
- 2) $g(x, y) \rightsquigarrow x$;
- 3) $g(x, y) \rightsquigarrow y$;

The sequence :
$f(g(a, b), g(a, b), g(a, b)) \rightsquigarrow f(a, g(a, b), g(a, b)) \rightsquigarrow$
$f(a, b, g(a, b)) \rightsquigarrow f(g(a, b), g(a, b), g(a, b)) \rightsquigarrow$ ... is infinite.
Remark : For a given rewrite system prooving its termination is indecidable. Various proof techniques have been proposed based on the construction of reduction ordering.

Confluence

The confluence property is verified if a rewrite system converge it is to a unique value. Example : Given the rewrite system, a,b,c constants, f,g functional symbols and x variable :

- 1) $f(x,x) \rightsquigarrow a$;
- 2) $f(x,g(x)) \rightsquigarrow b$;
- 3) $c \rightsquigarrow g(c)$;

Is not confluent.
For example, the normal form of f(c,c) is a and b. (c has no normal form).

- 1) $f(c,c) \rightsquigarrow a$;
- 2) $f(c,c) \rightsquigarrow f(c,g(c)) \rightsquigarrow b$;

Properties of rewrite rules

Theorem (validity)

Given $Spec =< \Sigma, X, AX >$, *and a set of rewrite rules obtained by orientation of the axioms defining the relation* $\rightsquigarrow$ *which is confluent and with termination*
$Th_{\rightsquigarrow}(Spec) \subseteq Th(Spec)$

Critical Pairs - Knuth-Bendix theorem

Let $l_1 \rightsquigarrow r_1$ and $l_2 \rightsquigarrow r_2$ be two rules of a term rewriting system.
we suppose that these rules have no variables in common.
If l_1^{sub} is a subterm (and not a variable) of l_1 (or the term itself) with $l_1^{context}[l_1^{sub}] = l_1$ and there exist a most general unifier σ such that $l_1^{sub}\sigma = l_2\sigma$, then $r_1\sigma$ and $l_1^{context}[r_2\sigma]$are called a critical pair.

The fact that all critical pairs of a term rewriting system can be reduced to the same expression, implies that the system is locally confluent.

Critical Pairs - Knuth-Bendix theorem(2)

The axioms of group theory are :

- $0 + x = x$
- $x^{-1} + x = 0$
- $(x + y) + z = x + (y + z)$

cf. exercices

Critical Pairs - Knuth-Bendix theorem(3)

For instance, if $f(x,x) \rightsquigarrow x$ and $g(f(x,y),x) \rightsquigarrow h(x)$, then $g(x,x)$ and $h(x)$ would form a critical pair because they can both be derived from $g(f(x,x),x)$.

Note that it is possible for a critical pair to be produced by one rule, used in two different ways. For instance, in the string rewrite "AA" $\rightsquigarrow$ "B", the critical pair ("BA", "AB") results from applying the one rule to "AAA" in two different ways.[5]

5. Rowland, Todd; Sakharov, Alex; and Weisstein, Eric W. "Critical Pair." From MathWorld–A Wolfram Web Resource.

Critical Pairs - Knuth-Bendix theorem(3)

Theorem (Knuth-Bendix)

Given a set of rewrite rules Rew, If $t \rightsquigarrow t_1$ *and* $t \rightsquigarrow t_2$ *then* $\exists t'$ *such that* $t_1 \rightsquigarrow^* t'$ *and* $t_2 \rightsquigarrow^* t'$ *or* $\exists (c_1, c_2)$ *a critical pair of Rew , a context* $C[]$ *and a substitution* σ *s.t.* $t_1 = C[c_1\sigma]$, $t_2 = C[c_2\sigma]$

Knuth Bendix completion

The Knuth-Bendix completion algorithm attempts to transform a finite set of identities into a finitely terminating, confluent term rewriting system whose reductions preserve identity. [6]

- Identities are equalities of two terms : $t_1 = t_2$. Two terms are equal for all values of variables occurring in them.
- A reduction order is another input to the completion algorithm. Every identity is viewed as two candidates for rewrite rules transforming the left-hand side into the right-hand side and vice versa.

6. This term rewriting system serves a decision procedure for validating identities.

Knuth Bendix completion

The output term rewriting system is used to determine whether $t_1 = t_2$ is an identity or not in the following manner.

- If two distinct terms t_1 and t_2 have the same normal form, then $t_1 = t_2$ is an identity.
- Otherwise, $t_1 = t_2$ is not an identity.

Term rewriting systems that are both finitely terminating and confluent have a unique normal forms for all expressions.

Knuth Bendix completion

Initially, this algorithm attempts to orient input identities according to the reduction order (if $t_1 < t_2$, then $t_1 \rightsquigarrow t_2$ becomes a rule). Then, it completes this initial set of rules with derived ones. The algorithm iteratively detects critical pairs, obtains their normal forms, and adds a new rule for every pair of the normal forms in accordance with the reduction order.
This algorithm may

1. Terminate with success and yield a finitely terminating, confluent set of rules,
2. Terminate with failure, or
3. Loop without terminating.

Exemple of completion

The axioms of group theory are :

- $0 + x = x$
- $x^{-1} + x = 0$
- $(x + y) + z = x + (y + z)$

If only left-right orientation is used then $x + x^{-1}$ is irreductible.
Using the completion, we have :

- $0 + x \rightsquigarrow x$, $x + 0 \rightsquigarrow x$
- $x^{-1} + x \rightsquigarrow 0$, $x + x^{-1} \rightsquigarrow 0$
- $(x + y) + z \rightsquigarrow x + (y + z)$, $0^{-1} \rightsquigarrow 0$
- $x^{-1} + (x + y) \rightsquigarrow y$, $x + (x^{-1} + y) \rightsquigarrow y$
- $(x^{-1})^{-1} \rightsquigarrow x$, $(x + y)^{-1} \rightsquigarrow x^{-1} + y^{-1}$

Summary

- Deductive and inductive theories have been presented
- The valid properties of the initial model are also valid in all models
- Rewriting is an operational model adapted to the proof of properties on closed terms
- Termination and confluence are desired properties of a rewrite system.
- The use of finitelly generated models wrt to constructors simplify inductive proofs.