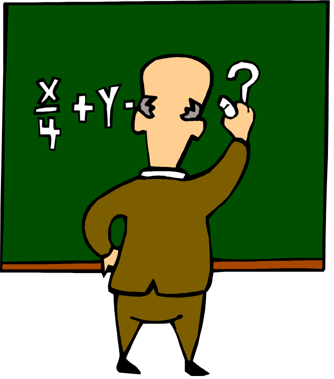
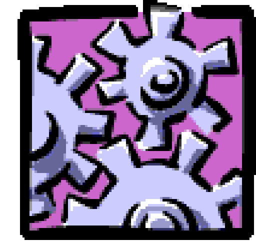
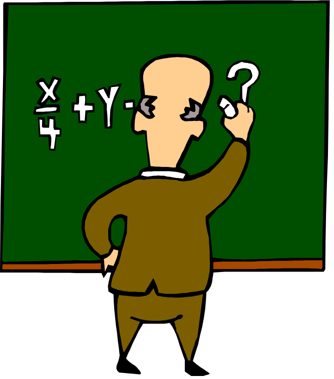




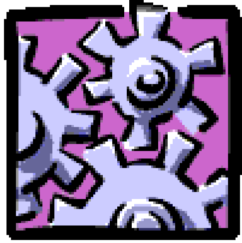
Test de Systèmes Informatiques

Partie III :
A Gentle Introduction to Isabelle/HOL

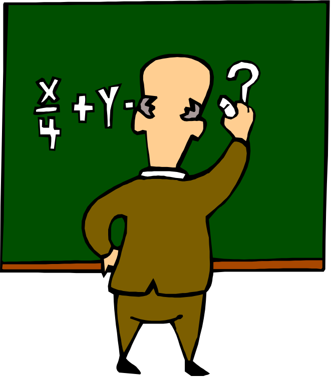
Burkhart Wolff,
Université Paris-Sud



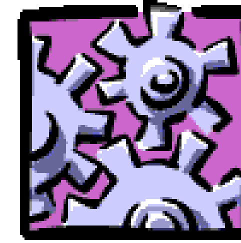
The History: Algebraic Specifications



- Abstract Data Types
- Description of required properties, independent of implementation
- Signature : sorts, opérations with profile
- + Axioms : equations, conditional equations (1st order formulas)
- (+ Constraints : hierarchy, finite generation)



A very basic example



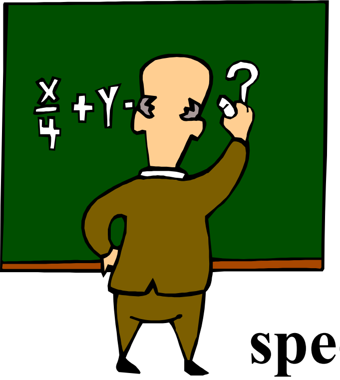
spec BOOL

free generated type *Bool* ::= true | false

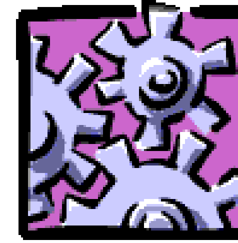
op *not* : *Bool* → *Bool*

- not(true) = false
- not(false) = true

end



A more sophisticated one



spec CONTAINER = NAT, BOOL

then

generated type *Container ::= [] | _::_(Nat ; Container)*

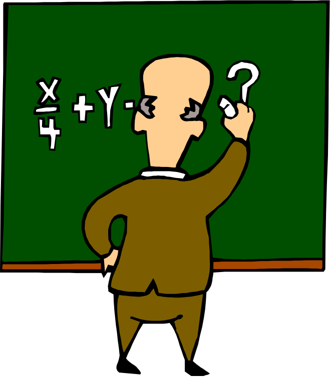
op *isin : Nat* $\times$ *Container* $\rightarrow$ *Bool*

op *remove: Nat* $\times$ *Container* $\rightarrow$ *Container*

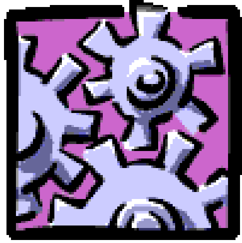
$\forall$ *x, y:Nat; c:Container*

- *isin(x, []) = false*
- *eq(x, y) = true* $\Rightarrow$ *isin(x, y::c) = true*
- *eq(x, y) = false* $\Rightarrow$ *isin(x, y::c) = isin(x,c)*
- *remove(x, []) = []*
- *eq(x, y) = true* $\Rightarrow$ *remove(x, y::c) = c*
- *eq(x, y) = false* $\Rightarrow$ *remove(x, y::c) = y::remove(x,c)*

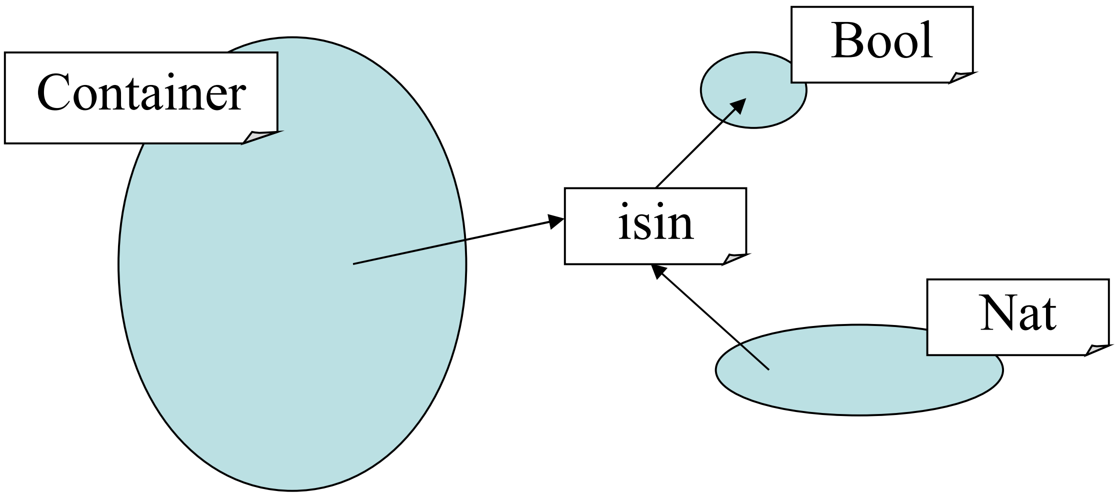
end



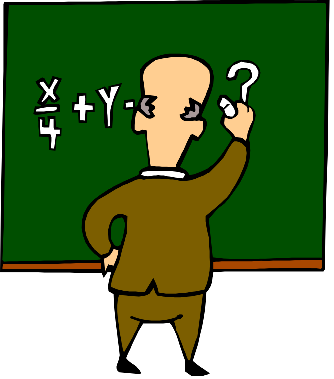
Formalities



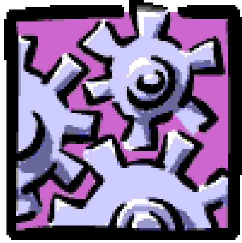
- **Semantics**
  - Many-sorted algebras: sets of values and functions



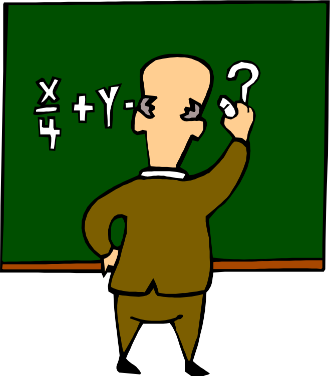
  - Question: which one?
    - initial semantics/ loose semantics
    - isomorphisms



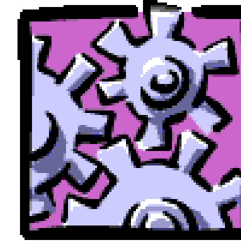
Higher-Order Logic



- … can do all that, too !!!



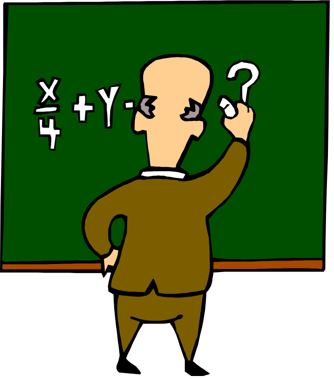
Higher-Order Logic



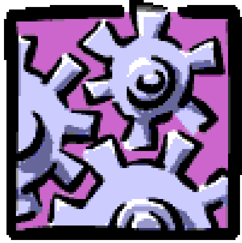
- … can do all that, too !!!

  … and more.

  it is a more modern specification language ...



Higher-Order Logic

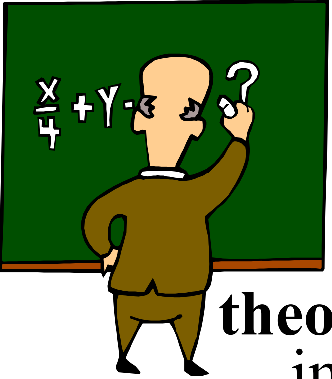


- … can do all that, too !!!

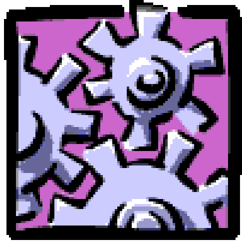
- one implementation: Isabelle/HOL

  (among other systems:
  Coq, HOL4, HOL-light, …)

- It is the implementation HOL-TestGen is built upon



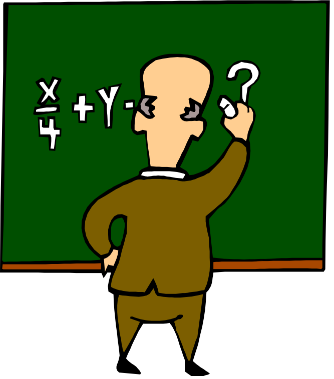
Recall Example: HOL



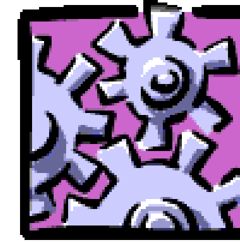
theory CONTAINER
  imports Main
begin
datatype *Container ::= [] | _::_(nat , Container)*
consts *isin :: [nat, Container]* $\Rightarrow$ *bool*
consts *remove: [nat, Container]* $\Rightarrow$ *Container*
axioms

- *isin x [] = False*
- $x=y \Rightarrow$ *isin x (y::c)*
- $x \neq y \Rightarrow$ *isin x (y::c) = isin x c*
- *remove x [] = []*
- $x = y \Rightarrow$ *remove x (y::c) = c*
- $x \neq y \Rightarrow$ *remove x (y::c) = y::(remove x c)*

end



Higher-Order Logic



- Isabelle/HOL has:
  - Signatures:

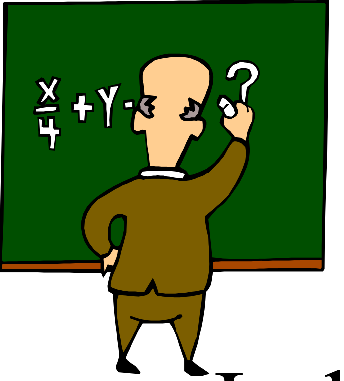
    ```
    arities nat
    consts plus :: [nat, nat] ⇒ nat
    ```

  - Special Syntax (Infix-Notations):

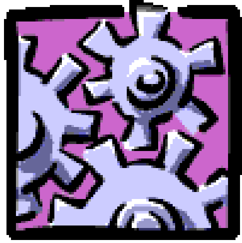
    ```
    consts plus :: [nat, nat]  ⇒ nat  (infixl + 55)
    ```

  - Axioms:

    ```
    axiom  add_commute: „(x::nat) + y = y + x"
    ```



Higher-Order Logic



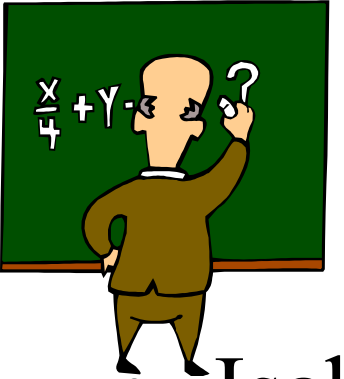
- Isabelle/HOL has additionally:
  - Higher-Order Syntax:

    `remove a C` instead `remove(a,C)`

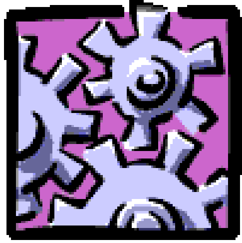
    as in functional programming languages
    (as SML, Haskell, Ocaml, …)

  - A type system with type classes as in Haskell:

    H :: [α::order,α] $\Rightarrow$ bool



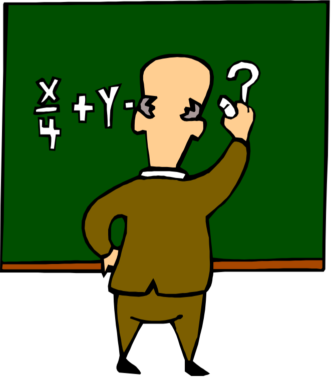
Higher-Order Logic



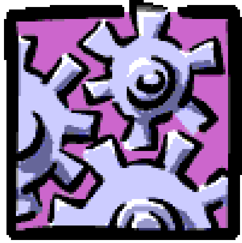
- Isabelle/HOL has additionally:
  - A built-in Equality : _ = _ :: [α,α] ⇒ bool with the rules:

$$\frac{}{s = s} \qquad \frac{s = t}{t = s} \qquad \frac{r = s \quad s = t}{r = t}$$

$$\frac{s = t \quad P\ s}{P\ t} \qquad \frac{\bigwedge x.\ s\ x = t\ x}{s = t}$$

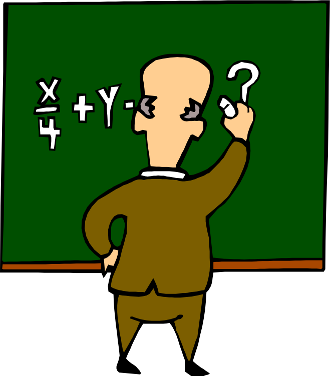


Higher-Order Logic

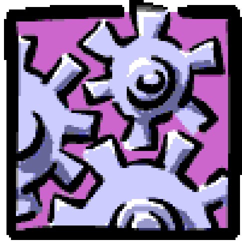


- Isabelle/HOL has additionally:
  - built-in expressions
    (as in functional programming languages)

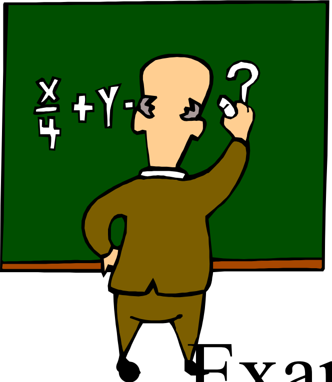
$$Suc = (\beta\ x\ .\ x + 1) :: nat \Rightarrow nat$$



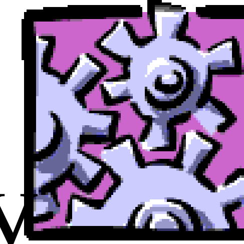
Higher-Order Logic



- Isabelle/HOL has additionally:
  - A particular « conservative » methodology avoiding (general and dangerous) axioms:
    - Logically safe and technically supported constructions for
      - Types
      - Definitions
      - Data-Types
      - Recursive Functions



Foundations: HOL / Library



- Examples:
  - type synonym (not even a type definition!)

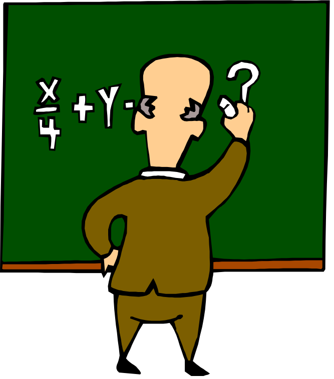
    types α set = "α ⇒ bool"

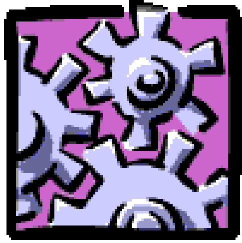
  - constant definitions

    ```
    definition   Collect   :: "(α ⇒ bool) ⇒ α set"
      where      "Collect S   ≡ S"
      definition member    :: "α⇒ α set ⇒ bool"
      where      "member s S ≡ S x"
    ```

  - syntactic paraphasing (not shown here!):

    Collect(β x. A) ≙ {x . A}, member s S ≙ s ∈ S





Foundations: HOL / Library

- conservative theory extension

  constant definition: "$c \equiv E$"

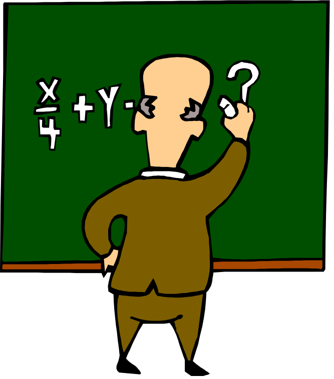
  (syntax: definition `c :: T` where `c ≡ E`)

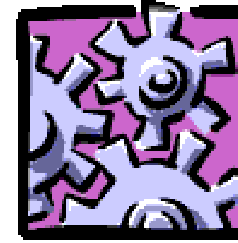
  side-conditions:

  – where constant symbol c is fresh

  – where E is closed and does not contain c

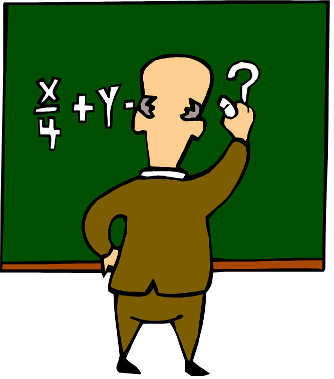
  – where no free type-variables occur in E that do not occur in the type of c

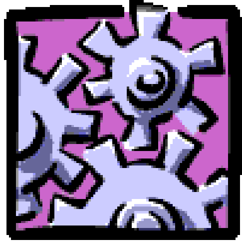




Foundations: HOL / Library

- conservative theory extension *type definition* *for* $\mathsf{T}' = \mathsf{T}(\alpha_1 .. \alpha_k)$ *from* $E :: \mathsf{T}(\alpha_1 .. \alpha_k) \Rightarrow \text{bool}$ (syntax: typedef $(\alpha_1 .. \alpha_k)\mathsf{T} = E$)
  - introduces constants Abs_T , Rep_T for type T
  - where type constructor T_k, Abs_T , Rep_T are fresh, and
  - where added axioms state an isomorphism between E and T'

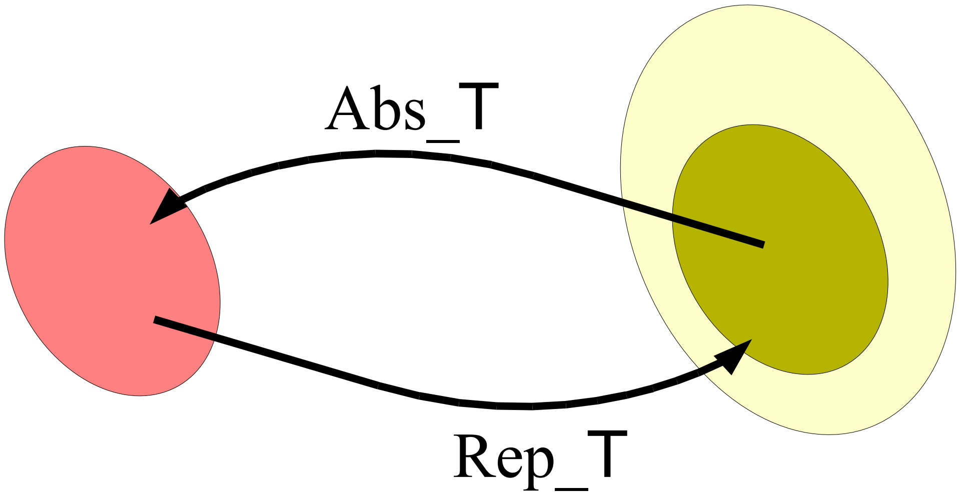


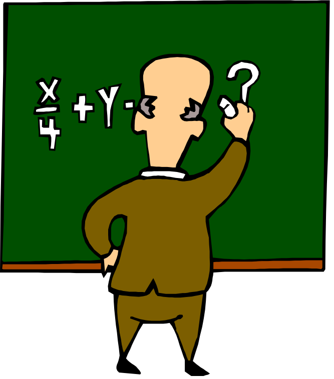


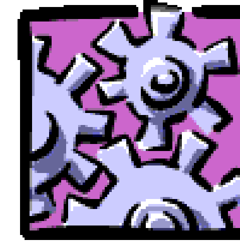
Foundations: HOL / Library

- conservative theory extension *type definition*

for $T' = T(\alpha_1 .. \alpha_k)$ *from set* $E :: T(\alpha_1 .. \alpha_k) \Rightarrow \text{bool}$







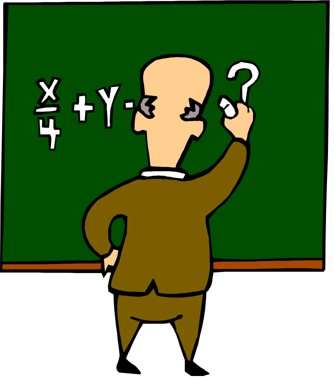
Foundations: HOL / Library

- conservative theory extension *type definition* *for* $\tau' = \tau(\beta_1 .. \alpha_k)$ *from set* $E :: \tau(\alpha_1 .. \alpha_k) \Rightarrow \text{bool}$

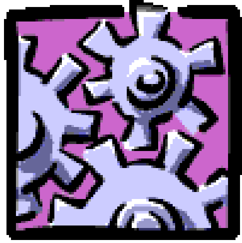
  – A: $\exists\, x.\ E\ x$ -- type consistency

  – B: $\text{Abs\_}\tau\ (\text{Rep\_}\tau\ x) = x$

  – C: $E\ x \Longrightarrow \text{Rep\_}\tau(\text{Abs\_}\tau\ x) = x$

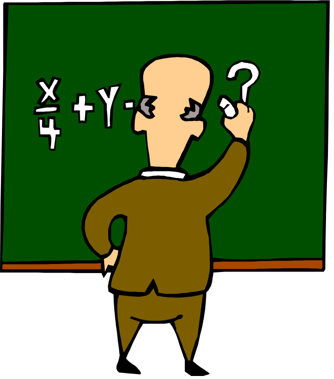


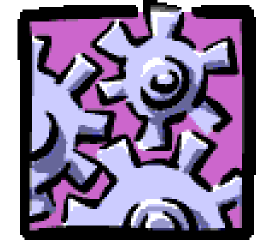
Foundations: HOL / Library



- Examples:
  - type definitions (syntax simplified)

```
typedef  ('a, 'b) "_×_"
         = "{f. ∃ a::'a b::'b. F = β x y. x = a ∧ y = b}"
```





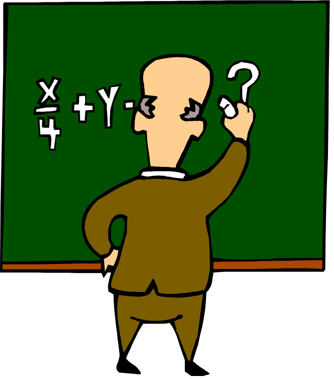
Foundations: HOL / Library

- Conservative datatype statements
  - data type definitions (automatically compiled to type definitions)

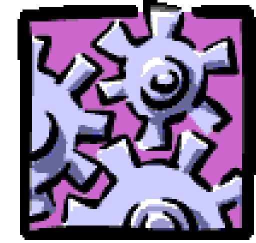
    datatype β option = None | Some β
    datatype β list = Nil | Cons β "β list"

  - syntax:

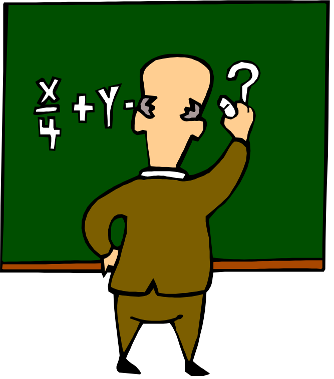
    Nil ≙ [], Cons a l ≙ a # l

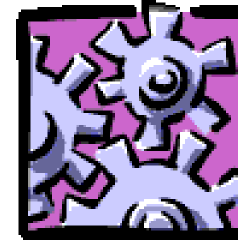


Foundations: HOL / Library



- Datatype statements generate a *data type theory* (i.e. a collection of definitions and theorems). Among others, we have:





Foundations: HOL / Library

- Datatype statements generate a *data type theory* (i.e. a collection of definitions and theorems). Among others, we have:
  - simplification rules (including case and size), for example for list:

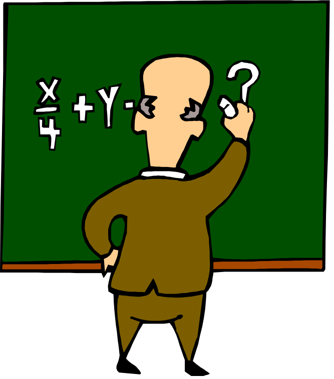
    list.list.cases(1): list_case f1 f2 [] = f1

    list.list.cases(2): list_case f1 f2 (a # list) = f2 a list

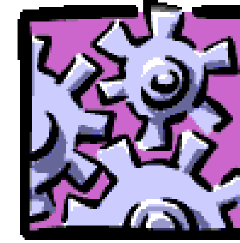
    list.simps: [] ≠ a # list, a # list ≠ []

    list.size(1): list_size fa [] = 0

    list.size(2): list_size f (a#l)=fa a + list_size fa l + 1



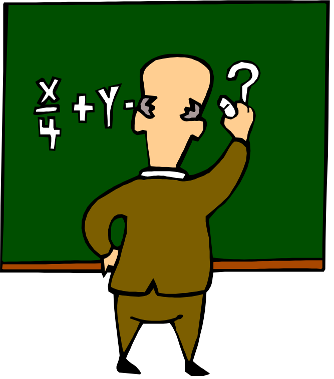
Foundations: HOL / Library



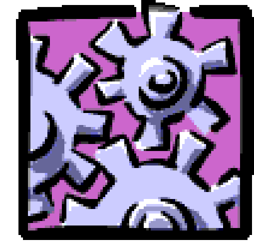
- Datatype statements generate a *data type theory* (i.e. a collection of definitions and theorems). Among others, we have:
  - injectivity rules, for example:

    list.inject: $(a \# l = a' \# l') = (a = a' \wedge l = l')$
  - exhaustion rules:

    list.exhaust: $[\![y = [] \Longrightarrow P;\ \bigwedge a\ l.\ y = a \# l \Longrightarrow P]\!] \Longrightarrow P$



Foundations: HOL / Library

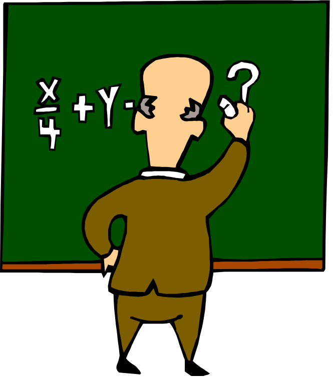


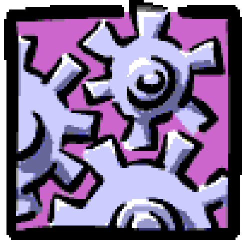
- Datatype statements generate a *data type theory* (i.e. a collection of definitions and theorems). Among others, we have:
  - induction rules:

list.induct: ⟦P []; ⋀a l. P l ⟹ P (a # l)⟧ ⟹ P (x::β list)

or in textbook-notation:

$$\frac{P\ [] \qquad \begin{array}{c}[P\ l]_{a,l}\\ \vdots \\ P(a\#l)\end{array}}{P(x :: \alpha\ list)}$$

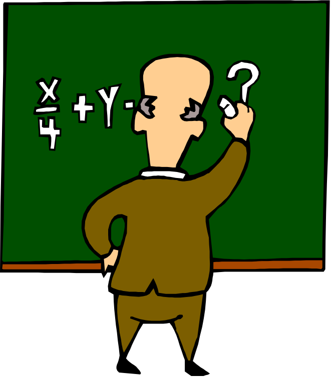




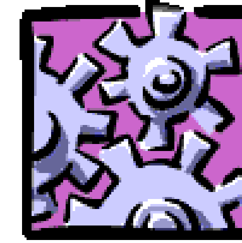
Foundations: HOL / Library

- Examples:
  - primitive recursions
    (automatically compiled to constant definitions)

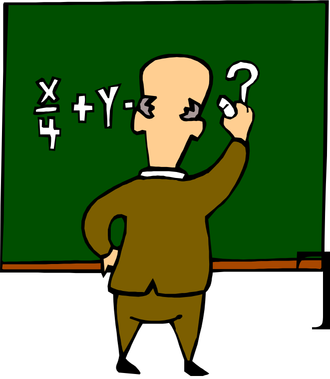
```
consts ins :: “[β::linorder, List β] ⇒  List β”
primrec
    ins x [] = [x]
    ins x (y#ys)= if x<y then x#(ins y ys) else y#(ins x  ys)

consts sort ::“List(a::linorder)  ⇒ List a”
primrec
    sort [] = []
    sort (x#xs) =ins x (sort xs)
```



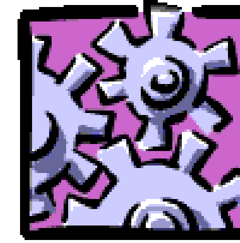
Foundations: HOL / Library



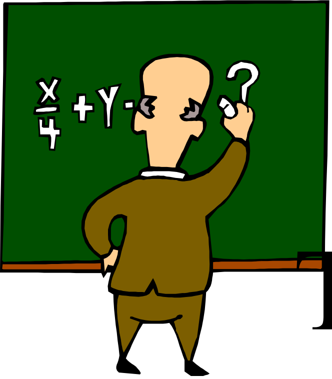
- With these two kinds of conservative extensions, the entire Library of Isabelle/HOL is built including:
  - set theory, inductive sets
  - wellfounded orders, well-founded recursion
  - arithmetic (nat,int,real,hyperreal, IEE754 floats)
  - data types, option, list, tree, ...
  - partial maps, updates, ...
  - programming language semantics (IMP, JAVA, JVM, ...)

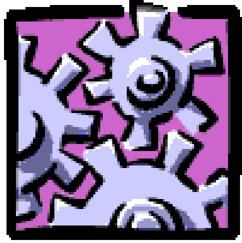


Testing : Conclusions Partie - III



- HOL is a universal foundation for:
  - on specifications (a model what a program should do)
  - on programs (a model on how a program does behave)
  - on symbolic computations of both
- More on HOL and its Theory: see MAN www.lri.fr/~wolff/teach-material/2009-10/MAN/





Testing : Conclusions Partie - III

- Thus, it is a good foundation for Model-based Testing and Tools like HOL-TestGen
  http://www.brucker.ch/projects/hol-testgen/

  Download of Version 1.6 from the TSI page, please !

  www.lri.fr/~wolff/teach-material/2009-10/M2R-TSI/