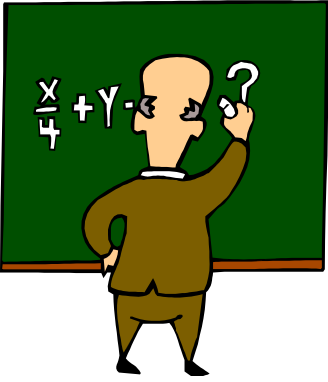


Test de Systèmes Informatiques

Partie III : Tests based on Data-Types

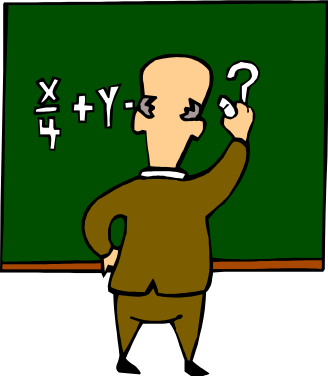
Burkhart Wolff, LRI
(basé sur Marie-Claude Gaudel)



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



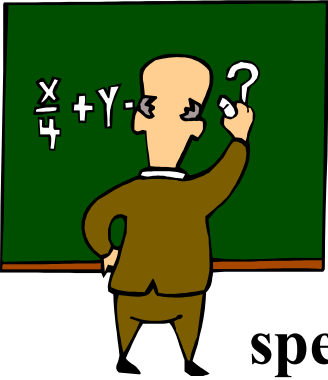
spec BOOL

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* ::= [] | $_::_(\text{Nat} ; \text{Container})$

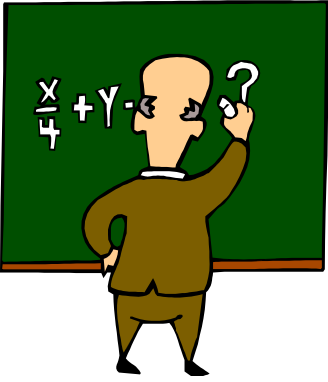
op *isin* : $\text{Nat} \times \text{Container} \rightarrow \text{Bool}$

op *remove* : $\text{Nat} \times \text{Container} \rightarrow \text{Container}$

$\forall x, y:\text{Nat}; c:\text{Container}$

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

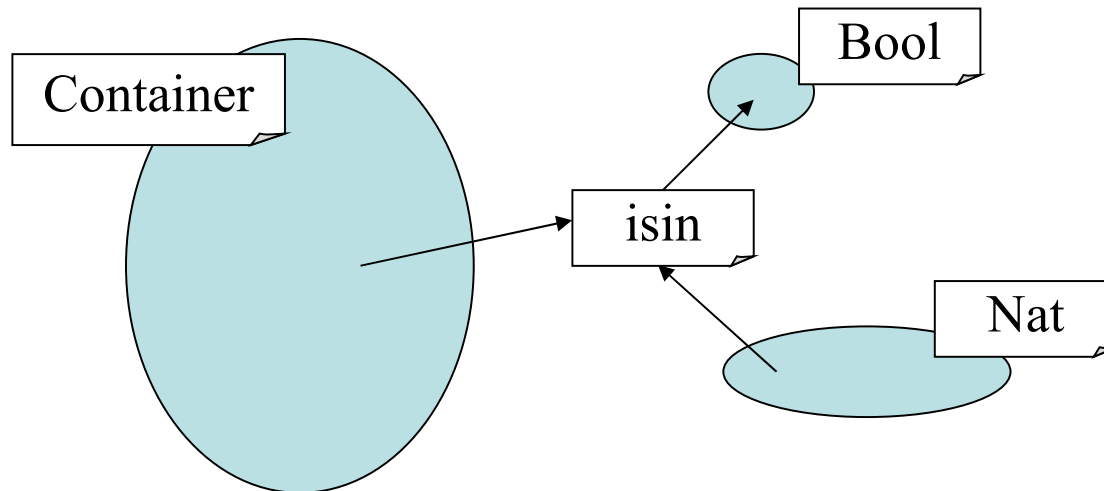
end



Formalities

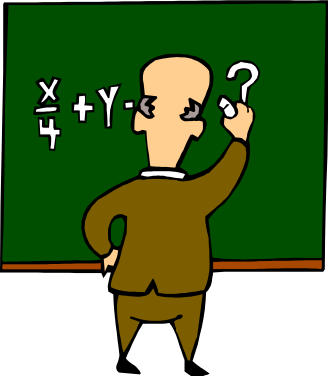
- **Semantics**

- Many-sorted algebras: sets of values and functions



- Question: which one?

- initial semantics/ loose semantics
- isomorphisms



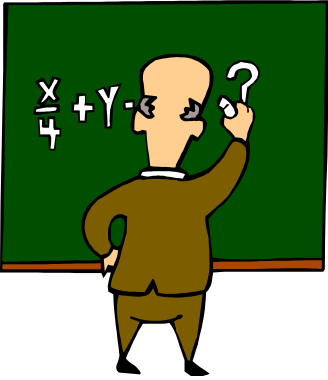
Specificities of testing based on AS



- It is not natural to test the operations with couples $\langle \text{input}, \text{output} \rangle$
 - Note that many outputs may be acceptable...
- What must be verified is that *the constructs which implement the operations satisfy the axioms*
- Exercises :

$$x + y = y + x$$

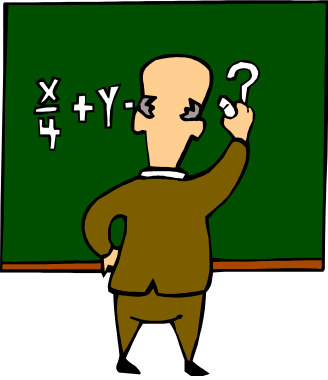
$$eq(x, y) = true \Rightarrow isin(x, y::c) = true$$



Link between the specification and the SUT

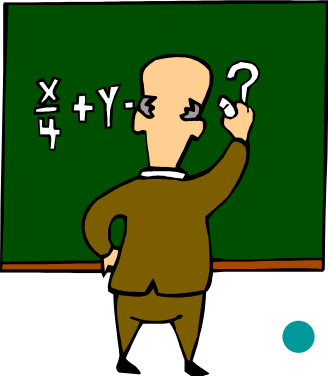


- The *SUT* provides some procedures, functions, methods, for executing the operations of the signature
 - (example : Java class, Ada package, ML structure...)
- Let note op_{SUT} the implementation of op
- Let t an expression without variable written with some operations of the signature,
- we note t_{SUT} the result of computation by *SUT*



What is a test?

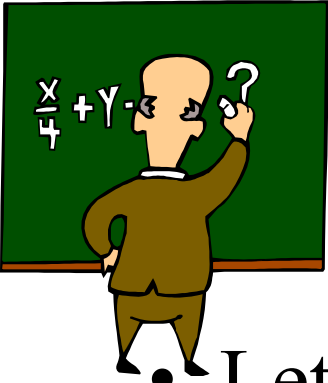
- Let ε some equation written with the operations of the signature (and, may be, some variables)
 - **test** of ε : any close instantiation $t = t'$ of ε
 - **test experiment** of SUT against $t = t'$: evaluations of t_{SUT} et t'_{SUT} and comparison of the resulting values
 - **NB** : oracle \Leftrightarrow test of equality
- Straightforward generalisation to conditional equations; pb with some 1st order formulas (\forall , \exists).



Examples of tests (simplified for conditional axioms)



- $isin(x, []) = false$
 - $isin(0, []) = false$
- $eq(x, y) = true \Rightarrow isin(x, y::c) = true$
 - $isin(1, 1::2:: []) = true$
- $eq(x, y) = false \Rightarrow isin(x, y::c) = isin(x, c)$
 - $isin(1, 0::3:: []) = isin(1, 3::[])$
- $remove(x, []) = []$
 - $remove(5, []) = []$
- $eq(x, y) = true \Rightarrow remove(x, y::c) = c$
 - $remove(0, 0::3::4:: []) = 3::4::[]$
- $eq(x, y) = false \Rightarrow remove(x, y::c) = y::remove(x, c)$
 - $remove(1, 7::5::3:: []) = 7::remove(1, 5::3:: [])$

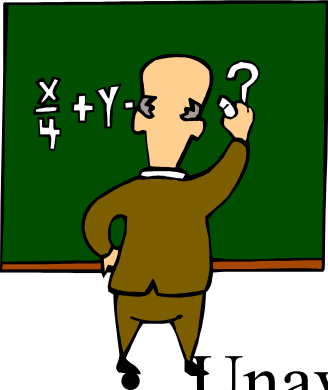


Exhaustive test set

- Let $SP = (\Sigma, Ax)$
- The exhaustive test set of SP , noted $Exhaust_{SP}$ is the set of all the closed well-sorted instances of all the axioms of SP :

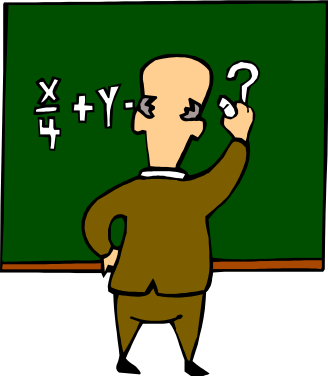
$$Exhaust_{SP} = \{ \Phi\sigma \mid \Phi \in Ax, \\ \sigma = \{ \sigma_s : var(\Phi)_s \rightarrow (T_\Sigma)_s \mid s \in S \} \}$$

- **NB1** : definition derived from the classical notion of axiom satisfaction
- **NB2** : some tests are inconclusive and can be removed (see notes)



Testability Hypotheses

- Unavoidable : when testing a system, it is impossible not to make assumptions on its behaviour and its environment
- **Remark** : $Exhaust_{SP}$ is exhaustive w.r.t. the specification, not always w.r.t. the implementation...
- Here : SUT is Σ -testable if :
 - The operations of Σ are implemented in a **deterministic way**
 - All the values are specified by Σ (**no junks**, Σ generation)
 - Notation : $H_{\min}(SUT)$ « minimal hypothesis »



Test and correctness

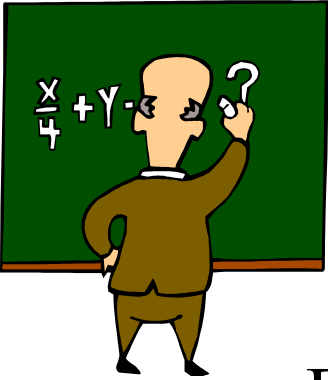
- Under the testability hypothesis, the success of the exhaustive test set guarantees that SUT satisfies the axioms

SUT testable \Rightarrow

(SUT passes Exhaust_{SP} \Leftrightarrow SUT sat SP)

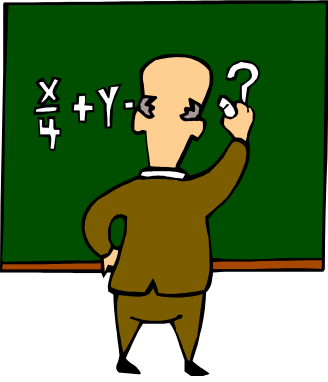
H_{min}(SUT) \Rightarrow

(SUT passes Exhaust_{SP} \Leftrightarrow SUT sat Ax)

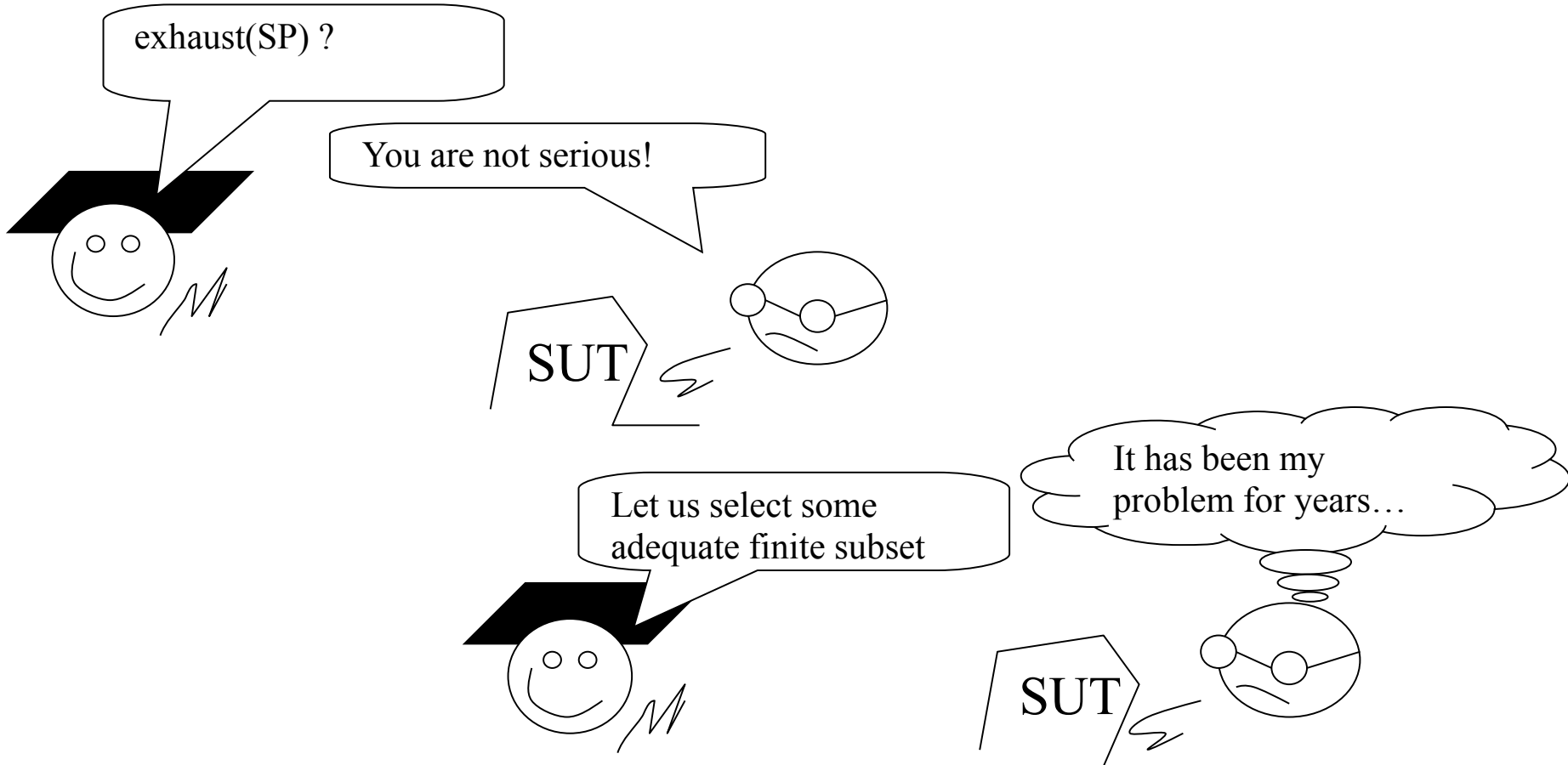


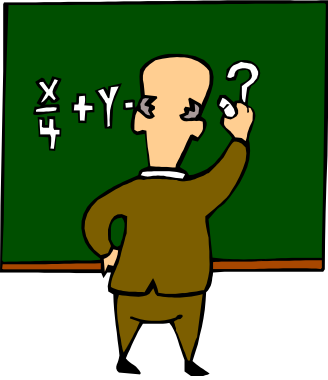
Another exhaustivity

- Based on a different (operational) semantics
 - $\{t = t\downarrow \mid T_\Sigma\}$
 - T_Σ is the sorted set of ground Σ -terms
 - $t\downarrow$ is the normal form of t , when using the axioms as conditional rewriting rules
- Restriction on the class of specifications
 - The axioms must define a convergent term rewriting system
- Weakening of the testability hypothesis
 - Finite generation is no more required



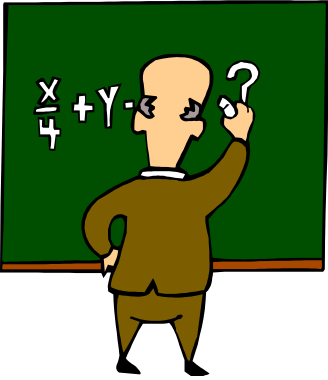
Exhaustivity is not practicable





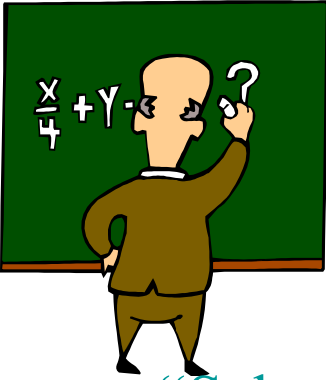
Selection

- How to select finite subsets of $Exhaust_{SP}$?
- *Test Set Selection* is based on the text of the specification (BBT!)
- Among the solutions: “partition testing”
 - Coverage of the sets of values by a finite number of subsets (sub-domains)
 - Choice of at least one values in each sub-domain



Selection Hypotheses

- STRONGER hypotheses on *SUT* (*selection hypotheses*)
- Example : *Uniformity Hypothesis*
 - $\Phi(X)$ formula, *SUT* system, *D* sub-domain
 - $(\forall t_0 \in D)(SUT \models \Phi(t_0) \Rightarrow (\forall t \in D)(SUT \models \Phi(t)))$
- Determination of sub-domains ?
guided by the specification, see later...
- Other example : *Regularity Hypothesis*
 - $((\forall t \in T_\Sigma)(t \leq k \Rightarrow P \models \Phi(t))) \Rightarrow (\forall t \in T_\Sigma)(P \models \Phi(t))$
 - Determination de $|t|$? *cf. specification*



Selection of finite test sets



- “Selection Hypotheses” H on SUT , and construction of practicable test sets T such that:

H holds for $SUT \Rightarrow$

*(SUT passes $T \Leftrightarrow$
 SUT sat SP)*

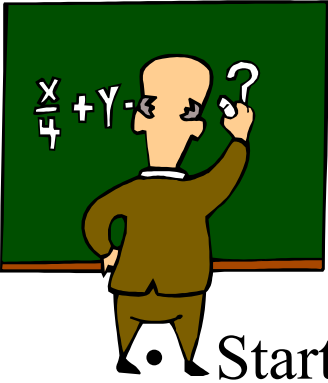
- $\langle H, T \rangle$ is a **valid and unbiased Test Context**
- or: T is **complete** w.r.t. H

$\langle SUT$ testable, exhaust(SP) \rangle

\langle Weak Hyp, Big Test Set \rangle

\langle Strong Hyp, Small TS \rangle

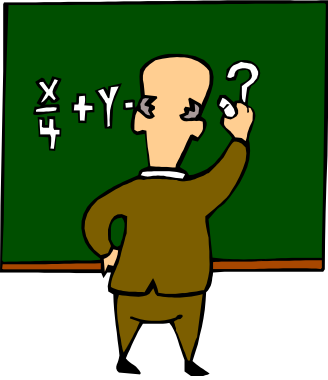
$\langle SUT$ correct, \emptyset \rangle



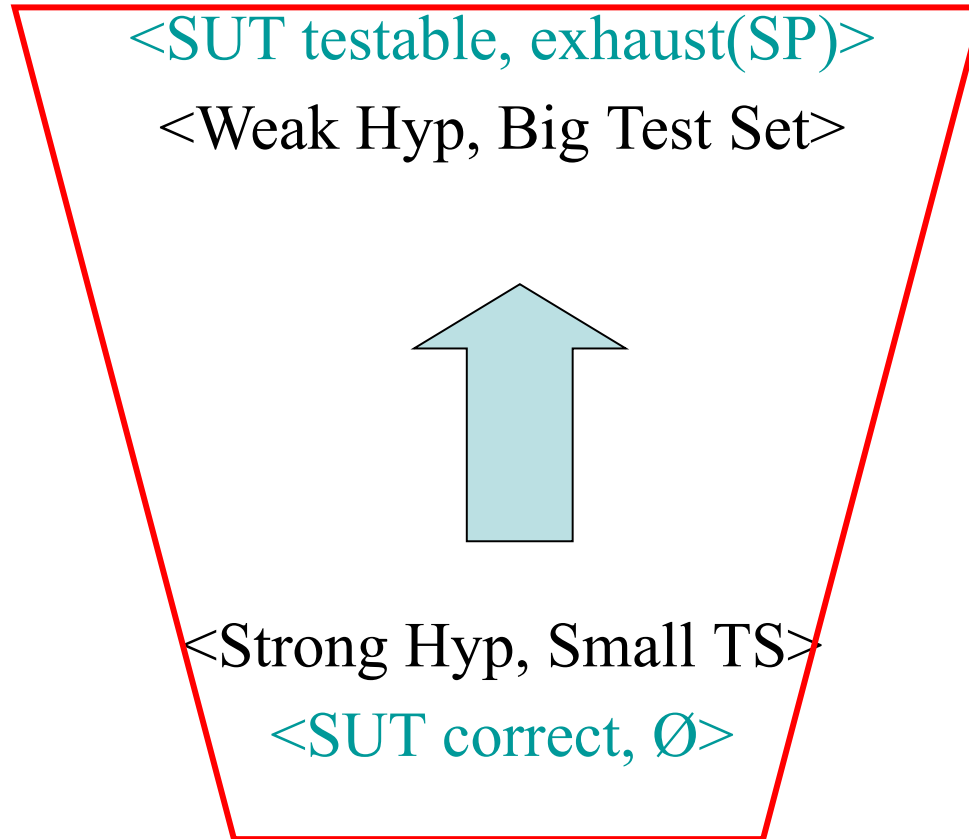
A Method

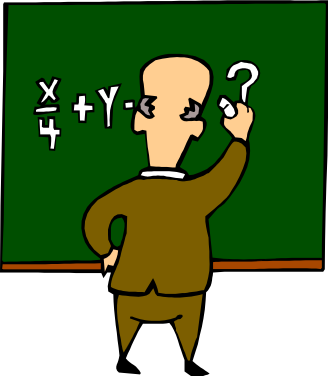


- Starting point : axioms coverage (one test by axiom)
 - \Rightarrow Strong uniformity hypotheses on the sorts of the variables or on the validity domain of the premisses
 - Example : 6 tests for **CONTAINER**
 - `isin (0, []) = false`
 - `isin(1, 1::2:: []) = true`
 - `isin(1, 0::3:: []) = isin(1, 3::[])`
 - `remove(1, []) = []`,
 - `remove(0, 0::3:: []) = 3:: []`
 - `remove(1, 3:: []) = 3:: remove(1, [])`
 - Uniformity on *Nat*, on pairs of *Nat* such that $eq(x,y) = true$, on pairs of *Nat* such as $eq(x, y) = false$
 - Uniformity on *Container*



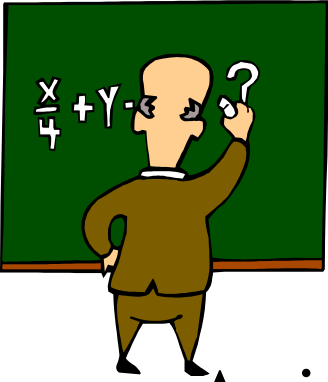
The principle of this method





Weakening of hypotheses

- Successive weakening using the axioms of the specification
- A natural way for discovering sub-domains is to perform *some case analysis of the specification*
- Example : the *isin* function is defined by 3 axioms
 - $isin(x, []) = false$
 - $eq(x, y) = true \Rightarrow isin(x, y::c) = true$
 - $eq(x, y) = false \Rightarrow isin(x, y::c) = isin(x, c)$
- \Rightarrow 3 tests. But one may want to go further
 - Occurrences of *isin*(,) can be decomposed into these 3 subcases



2 main techniques for weakening uniformity hypotheses



- Axioms Composition

- For instance, given the axioms:

$$\text{eq}(x,y) = \text{true} \Rightarrow \text{le}(x, y) = \text{true}$$

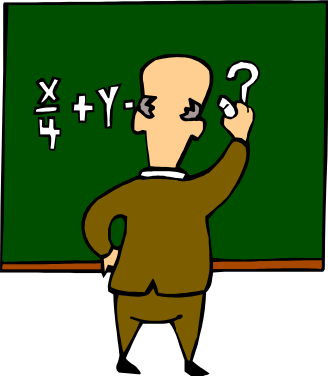
$$\text{lt}(x,y) = \text{true} \Rightarrow \text{le}(x, y) = \text{true}$$

$$\text{lt}(x,y) = \text{false} \wedge \text{eq}(x, y) = \text{false} \Rightarrow \text{le}(x, y) = \text{false}$$

- any occurrence of $\text{le}(,)$ in an axiom can be decomposed into 3 sub-cases (or 2, or 1... depending on its context)

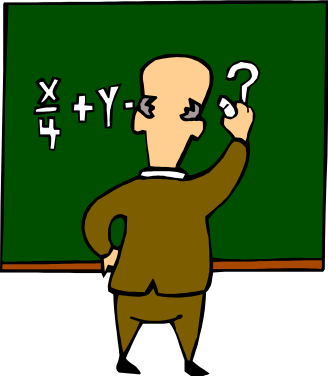
- Unfolding of recursive occurrences,

- see next slide



Unfolding

- Unfolding is a classical technique for transforming (and understanding) recursive definitions
- It is just replacement of $f(op(x))$ by the definition(s) of f , with adequate renaming of variables
 - $\text{fact}(n) =_{\text{def}} \text{if } n=0 \text{ then } 1 \text{ else } n * \text{fact}(n-1)$ becomes:
 - $\text{fact}(n) =_{\text{def}} \text{if } n=0 \text{ then } 1 \text{ else if } (n-1)=0 \text{ then } n * 1 \text{ else } n * (n-1) * \text{fact}(n-2)$
 - i.e. $\text{fact}(n) =_{\text{def}} \text{if } n=0 \text{ then } 1 \text{ else if } n=1 \text{ then } 1$
else $n * (n-1) * \text{fact}(n-2)$
 - etc
 - Going on, the definition of the *fact* function is replaced by its graph, i.e. its *exhaustive test set*...

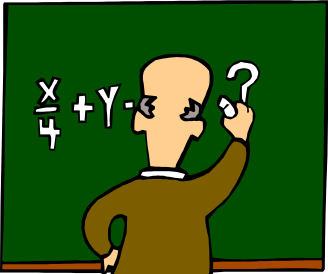


Unfolding *isin*



The definition of *isin* is:

- $isin(x, []) = false$
- $eq(x, y) = true \Rightarrow isin(x, y::c) = true$
- $eq(x, y) = false \Rightarrow isin(x, y::c) = isin(x, c)$
- Thus any term $isin(t1, t2)$ may correspond to three subcases
 - $t2=[]$: $isin(t1, t2)$ can be replaced by *false*
 - $t2= y::c$, and $eq(t1, y)=true$: it can be replaced by *true*
 - $t2= y::c$, and $eq(t1, y)=false$: it can be replaced by $isin(t1, c)$



Infolding the 3rd (red) axiom

$$eq(x, y) = false \Rightarrow isin(x, y::c) = isin(x, c)$$

- $c=[]$:

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

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

- $c=y'::c'$, and $eq(x, y')=true$

- $eq(x, y) = false \wedge c=y'::c' \wedge eq(x, y')=true \Rightarrow isin(x, y::y'::c') = isin(x, y'::c')$

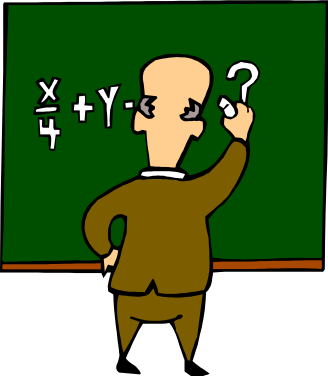
- $eq(x, y) = false \wedge eq(x, y')=true \Rightarrow isin(x, y::y'::c') = true$

- $c=y'::c'$, and $eq(x, y')=false$:

- $eq(x, y) = false \wedge c=y'::c' \wedge eq(x, y')=false \Rightarrow isin(x, y::y'::c') = isin(x, y'::c')$

- $eq(x, y) = false \wedge eq(x, y')=true \Rightarrow isin(x, y::y'::c') = false$

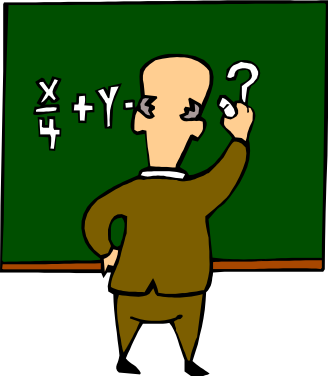
3 new test cases



When and how to stop



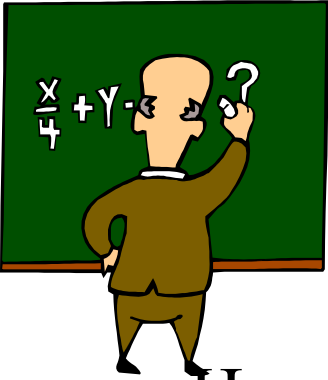
- Depending on the context (risk, cost, schedule, ...), one chooses for each specification:
 - What boolean functions or predicates to decompose (le, or, and, ...)
 - What operations to unfold and how many times (rarely more than once, but there is a counter example at the end of the course 😊)
- Some good standard strategy : composition of all pairs of sub-cases
 - NB : There may be unfeasible compositions



The oracle problem



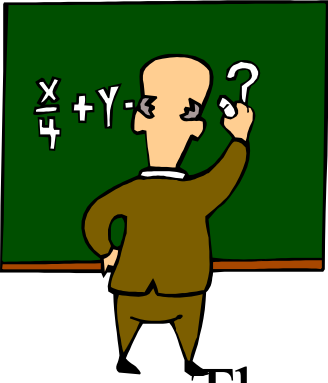
- Decision that t_{SUT} and t'_{SUT} are “equal”
- The simple case :
 - the sort s of t and t' corresponds to some type of the programming language with a built-in equality (observable sort)
- “Weak oracle hypothesis”: the built-in equality on the types of the programming language, and the booleans, are correctly implemented



The other cases



- How to test that
$$eq(x, y) = false \Rightarrow remove(x, y::c) = y::remove(x, c) ?$$
- Suppose that containers are represented by hash-tables, or ordered trees, or ...
- Solution 1 : *observable contexts*
 - Test that all the possible “observations” on the two results are equal
 - **Observation** : (minimal) composition of operations of the signature that yields an observable results



Observable contexts

- The CONTAINER example

- $isin(n, _)$, for all $n: \text{Nat}$

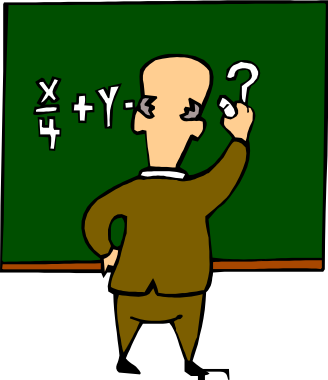
- $eq(x, y) = false \Rightarrow isin(n, remove(x, y::c)) = isin(n, y::remove(x, c))$

As in this case, there is often an infinity of observable contexts ☹

- Need for selection strategies

- Either among the observable contexts \Rightarrow partial oracle

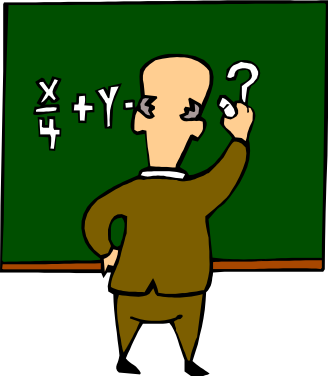
- Either among a new observable exhaustive test set (see [30])



Solution 2 (Machado)



- Every non-observable equations in positive positions are observed by means of **observable contexts** while those in negative positions are observed by using the **concrete equality** in the implementation.
- In that sense, Machado's approach is not a pure black-box approach deriving test cases and oracles from specifications but an approach *mixing black-box and white-box* where
 - test cases are derived from the specifications and
 - the oracle procedure is built from both the specification and some knowledge of the SUT.



Some applications of testing based on AS



- Onboard part of the driving system of an automatic subway (line D, Lyon)
- pieces of software written in C, parts of a nuclear safety shutdown system.
- EPFL library of Ada components
- *Validation* of a transit node algebraic specification
- test of the data types in an implementation of the Two-Phase-Commit protocol
- Application to other specification methods embedding abstract data types specification