CM ACF - Table of contents

— CM1 : Propositional logic and First order logic
  — Why using logic for specifying/verifying programs ?
  — Propositional logic
    — Formula syntax
    — Interpretations and models (Interpretations, models, tautologies)
    — Isabelle/HOL commands (apply auto, nitpick)
  — First-order logic
    — Formula syntax
    — Interpretations and models (Interpretations, Valuations, Models, Tautologies)
    — Isabelle/HOL commands (apply auto, nitpick)

— CM2 : Types, terms and functions
  — Terms
    — Types, typed terms : type inference and type annotations (value)
    — λ-terms (syntax, semantics λ-calculus, curried functions, partial application, higher-order functions)
    — Isabelle/HOL commands (definition)
    — Constructor terms (Definition, Isabelle Theory Library)
  — Functions defined using equations
    — Logic everywhere! (Definition of total and partial functions with equations)
    — Function evaluation using term rewriting (substitutions and rewriting)
    — Partial functions
    — Isabelle/HOL command (export_code)

— CM3 : Recursive functions and Algebraic Data Types
  — Recursive functions
    — Definition
    — Termination proofs with measures
    — Difference between fun, function and primrec
  — (Recursive) Algebraic Data Types
    — Defining Algebraic Data Types using datatype
    — Building objects of Algebraic Data Types
    — Matching objects of Algebraic Data Types (with case and where)
    — Type abbreviations (with type_synonym)

— CM4 : Proofs with a proof assistant
  — Finding counterexamples
    — nitpick and models of finite domain
    — quickcheck and random test generation
  — Proving true formulas

— CM5 : Crash Course on Scala
  — Basics
    — Base types and type inference
    — Control : if and match - case
    — Loops : For
    — Structures : Arrays, Lists, Tuples, Maps
  — Functions
    — Basic functions
    — Anonymous, Higher order functions and Partial application
    — Object Model
    — Class definition and constructors
    — Method/operator/function definition, overriding and implicit definitions
    — Traits and polymorphism
    — Singleton Objects
    — Case classes and pattern-matching
    — Interoperability between Java and Scala
    — Isabelle/HOL export in Scala export_code

— CM6 : Certified Programming
  — Certified program production lines
    — Some examples of certified code production lines
    — What are the weak links ?
    — How to limit the trusted base ? How to certify a compiler ? How to certify a static analyzer of code ?
    — How to guarantee the correctness of proofs ? (Difference between some proof assistants)
    — Methodology for formally defining programs and properties
    — Simple programs have simple proofs
    — Generalize properties when possible
    — Look for the smallest trusted base

— CM7 : Program Verification Methods
  — Basics (Specification, Oracle, Domain of Definition)
  — Testing (random testing, white box testing)
  — Model-checking
  — Assisted proof
  — Static Analysis (Abstract domains, abstract interpretation, proving the correctness of a static analyzer)
  — A word about prototypes/models, accuracy, code generation
Outline

- Why using logic for specifying/verifying programs?
- Propositional logic
  - Formula syntax
  - Interpretations and models
  - Isabelle/HOL commands
- First-order logic
  - Formula syntax
  - Interpretations and models
  - Isabelle/HOL commands

Why using logic for specifying/verifying programs?

Bibliography


A selected bibliography on the Isabelle/HOL prover


The web page of the course


Solutions of Isabelle/HOL exercises (uploaded after each lecture)


Acknowledgements

Many thanks to T. Nipkow, J. Blanchette, L. Bulwahn and G. Riou for providing material, answering questions and for fruitful discussions.
Why using logic for specifying/verifying programs?

Why using logic for specifying/verifying programs?

Why using functional paradigm to program?

Why using functional paradigm to program?
Why using functional paradigm to program?

Proof Assistants
Counterexample finders
First-order provers
SMT solvers
SAT Solvers

Functional Program
Logic

Propositional logic: syntax and interpretations

Definition 1 (Propositional formula)
Let $P$ be a set of propositional variables. The set of propositional formula is defined by

$\phi ::= p \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \phi_1 \land \phi_2 \mid \phi_1 \rightarrow \phi_2$ where $p \in P$

Definition 2 (Propositional interpretation)
An interpretation $I$ associates to variables of $P$ a value in \{True, False\}.

Example 3
Let $\phi = (p_1 \land p_2) \rightarrow p_3$. Let $I$ be the interpretation such that $I[p_1] = \text{True}$, $I[p_2] = \text{True}$ and $I[p_3] = \text{False}$.

Propositional logic: syntax and interpretations (II)

We extend the domain of $I$ to formulas as follows:

$I[\neg \phi] = \begin{cases} \text{True} & \text{iff } I[\phi] = \text{False} \\ \text{False} & \text{iff } I[\phi] = \text{True} \end{cases}$

$I[\phi_1 \lor \phi_2] = \text{True} \text{ iff } I[\phi_1] = \text{True or } I[\phi_2] = \text{True}$

$I[\phi_1 \land \phi_2] = \text{True} \text{ iff } I[\phi_1] = \text{True and } I[\phi_2] = \text{True}$

$I[\phi_1 \rightarrow \phi_2] = \text{True} \text{ iff } \begin{cases} I[\phi_1] = \text{False or } I[\phi_2] = \text{True} \\ I[\phi_1] = \text{True and } I[\phi_2] = \text{True} \end{cases}$

Example 4
Let $\phi = (p_1 \land p_2) \rightarrow p_3$ and $I$ the interpretation such that $I[p_1] = \text{True}$, $I[p_2] = \text{True}$ and $I[p_3] = \text{False}$.

We have $I[p_1 \land p_2] = \text{True}$ and $I[(p_1 \land p_2) \rightarrow p_3] = \text{False}$. 
Propositional logic: syntax and interpretations (III)

The presentation using truth tables is generally preferred:

<table>
<thead>
<tr>
<th></th>
<th>¬a</th>
<th>a ∨ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>a ∧ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Propositional logic: models

Definition 5 (Propositional model)
I is a model of φ, denoted by I |= φ, if I[φ] = True.

Definition 6 (Valid formula/Tautology)
A formula φ is valid, denoted by |= φ, if for all I we have I |= φ.

Example 7
Let φ = (p1 ∧ p2) → p3 and φ' = (p1 ∧ p2) → p1. Let I be the interpretation such that I[p1] = True, I[p2] = True and I[p3] = False. We have I |= φ, I |= φ', and I |= φ'.

Propositional logic: decidability and tools in Isabelle/HOL

Property 1
In propositional logic, given φ, the following problems are decidable:
• Is |= φ?
• Is there an interpretation I such that I |= φ?
• Is there an interpretation I such that I |= ¬φ?

To automatically prove that |= φ .......................... apply auto (if the formula is not valid, there remains some unsolved goals)
To build I such that I |= ¬φ (or I |= ¬¬φ) .......................... nitpick (i.e. find a counterexample... may take some time on large formula)

Example 8 (Valid formula)
lemma "(p1 /\ p2) --> p1"
apply auto
done

Example 9 (Unprovable formula)
lemma "(p1 /\ p2) --> p3"
nitpick
oops

Writing and proving propositional formulas in Isabelle/HOL
Isabelle/HOL: ASCII notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>∧</td>
<td>\</td>
</tr>
<tr>
<td>∨</td>
<td>\</td>
</tr>
<tr>
<td>¬</td>
<td>¬</td>
</tr>
<tr>
<td>≠</td>
<td>≠</td>
</tr>
<tr>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>⇔</td>
<td>⇔</td>
</tr>
<tr>
<td>∀</td>
<td>ALL</td>
</tr>
<tr>
<td>∃</td>
<td>?</td>
</tr>
<tr>
<td>λ</td>
<td>%</td>
</tr>
</tbody>
</table>

Propositional logic: exercises in Isabelle/HOL

**Exercise 1**

Using Isabelle/HOL, for each formula, say if it is valid or give a counterexample interpretation, otherwise.

1. \( A \rightarrow B \)
2. \(((A \land B) \rightarrow \neg C) \lor (A \rightarrow B)) \rightarrow A \rightarrow C\)
3. If it rains, Robert takes his umbrella. Robert does not have his umbrella hence it does not rain.
4. \((A \rightarrow B) \iff (\neg A \lor B)\)

First-order logic (FOL) / Predicate logic

**Definition 10 (Terms)**

Let \( \mathcal{F} \) be a set of symbols and \( \text{ar} \) a function such that \( \text{ar} : \mathcal{F} \rightarrow \mathbb{N} \) associating each symbol of \( \mathcal{F} \) to its arity (the number of parameter). Let \( \mathcal{X} \) be a variable set.

The set \( \mathcal{T}(\mathcal{F}, \mathcal{X}) \), the set of terms built on \( \mathcal{F} \) and \( \mathcal{X} \), is defined by:

\[
\mathcal{T}(\mathcal{F}, \mathcal{X}) = \mathcal{X} \cup \{ f(t_1, \ldots, t_n) \mid \text{ar}(f) = n \text{ and } t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \}
\]

**Example 11**

Let \( \mathcal{F} = \{ f : 1, g : 2, a : 0, b : 0 \} \) and \( \mathcal{X} = \{ x, y, z \} \).

\( f(x), a, z, g(g(a, x), f(a)), g(x, x) \) are terms and belong to \( \mathcal{T}(\mathcal{F}, \mathcal{X}) \).

\( f, a(b), f(a, b), x(a), f(a, f(b)) \) do not belong to \( \mathcal{T}(\mathcal{F}, \mathcal{X}) \).
First-order logic: formula syntax

Definition 12 (Formulas)
Let $P$ be a set of predicate symbols all having an arity, i.e. $ar : P \Rightarrow \mathbb{N}$. The set of formulas defined on $\mathcal{F}$, $\mathcal{X}$ and $P$ is:

\[ \phi ::= \neg \phi \mid \phi_1 \lor \phi_2 \mid \phi_1 \land \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \forall x. \phi \mid \exists x. \phi \mid p(t_1, \ldots, t_n) \]

where $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $x \in \mathcal{X}$, $p \in P$ and $ar(p) = n$.

Example 13
Let $P = \{ p : 1, q : 2, \leq : 2 \}$, $\mathcal{F} = \{ f : 1, g : 2, a : 0 \}$ and $\mathcal{X} = \{ x, y, z \}$. The following expressions are all formulas:

- $p(f(a))$
- $\forall x. \exists y. y \leq x$
- $\forall x. \forall y. \forall z. x \leq y \land y \leq z \rightarrow x \leq z$

Interlude: a touch of lambda-calculus

We need to define anonymous functions

- Classical notation for functions
  \[ f : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} \quad \text{or, for short,} \quad f : \mathbb{N}^2 \Rightarrow \mathbb{N} \]
  \[ f(x, y) = x + y \quad f(x, y) = x + y \]

- Lambda-notation of functions
  \[ f : \mathbb{N}^2 \Rightarrow \mathbb{N} \]
  \[ f = \lambda x. y. x + y \]

\[ \lambda x. y. x + y \] is an anonymous function adding two naturals

This corresponds to

- `fun x y -> x+y` in OCaml/Why3
- `(x: Int, y:Int) => x + y` in Scala

Isabelle/HOL also use function update using (:=) as in:

- $(\lambda x. x)(0 := 1, 1 := 2)$ the identity function except for 0 that is mapped to 1 and 1 that is mapped to 2
- $(\lambda x. x)(a := b)$ a function taking one parameter and whose result is unspecified except for value $a$ that is mapped to $b$

Predicates in Isabelle/HOL

- A predicate is a function mapping values to $\{ \text{True, False} \}$
  
    For instance the predicate $p$ on $\{ a, b \}$
    
    \[ p = (\lambda x. a := \text{False}, b := \text{False}) \]

First-order formulas: interpretations and valuations

Definition 14 (First-order interpretation)
Let $\phi$ be a formula and $D$ a domain. An interpretation $I$ of $\phi$ on the domain $D$ associates:

- a function $f_I : D^n \Rightarrow D$ to each symbol $f \in \mathcal{F}$ such that $ar(f) = n$,
- a function $p_I : D^n \Rightarrow \{ \text{True, False} \}$ to each predicate symbol $p \in P$ such that $ar(p) = n$.

Example 15 (Some interpretations of $\phi = \forall x. \text{ev}(x) \rightarrow \text{od}(s(x))$)

- Let $I$ be the interpretation such that domain $D = \mathbb{N}$ and $s_I \equiv \lambda x. x + 1$ $\text{ev}_I \equiv \lambda x. ((x \mod 2) = 0)$ $\text{od}_I \equiv \lambda x. ((x \mod 2) = 1)$
- Let $I'$ be the interpretation such that domain $D = \{ a, b \}$ and $s_{I'} \equiv \lambda x. \text{if } x = a \text{ then } b \text{ else } a$ $\text{ev}_{I'} \equiv \lambda x. (x = a)$ $\text{od}_{I'} \equiv \lambda x. \text{False}$

Definition 16 (Valuation)
Let $D$ be a domain. A valuation $V$ is a function $V : \mathcal{X} \Rightarrow D$. 
First-order logic: interpretations and valuations (II)

Definition 17
The interpretation $I$ of a formula $\phi$ for a valuation $V$ is defined by:

- $(I, V)[x] = V(x)$ if $x \in \mathcal{X}$
- $(I, V)[f(t_1, \ldots, t_n)] = f((I, V)[t_1], \ldots, (I, V)[t_n])$ if $f \in \mathcal{F}$ and
  $ar(f) = n$
- $(I, V)[p(t_1, \ldots, t_n)] = p_1((I, V)[t_1], \ldots, (I, V)[t_n])$ if $p \in P$ and
  $ar(p) = n$
- $(I, V)[\phi_1 \vee \phi_2] = \text{True}$ iff $(I, V)[\phi_1] = \text{True}$ or $(I, V)[\phi_2] = \text{True}$
- etc...
- $(I, V)[\forall x. \phi] = \bigwedge_{d \in D} ((I, V + \{x \mapsto d\})[\phi])$
- $(I, V)[\exists x. \phi] = \bigvee_{d \in D} ((I, V + \{x \mapsto d\})[\phi])$

where $(V + \{x \mapsto d\})(x) = d$ and $(V + \{x \mapsto d\})(y) = V(y)$ if $x \neq y$.

Free variables are universally quantified (e.g. $P(x)$ equivalent to $\forall x. P(x)$)

First-order logic: satisfiability, models, tautologies

Definition 18 (Satisfiability)
$I$ and $V$ satisfy $\phi$ (denoted by $(I, V) \models \phi$) if $(I, V)[\phi] = \text{True}$.

Definition 19 (First-order Model)
An interpretation $I$ is a model of $\phi$, denoted by $I \models \phi$, if for all valuation $V$ we have $(I, V) \models \phi$.

Definition 20 (First-order Tautology)
A formula $\phi$ is a tautology if all its interpretations are models, i.e. $(I, V) \models \phi$ for all interpretations $I$ and all valuations $V$.

Remark 1
Free variables are universally quantified (e.g. $P(x)$ equivalent to $\forall x. P(x)$)

First-order logic: exercises in Isabelle/HOL

Exercise 2
Using Isabelle/HOL, for each formula, say if it is valid or give a counterexample interpretation and valuation otherwise.

- $\forall x. P(x) \longrightarrow \exists x. P(x)$
- $\exists x. P(x) \longrightarrow \forall x. P(x)$
- $\forall x. ev(x) \longrightarrow od(s(x))$
- $\forall x y. x \cdot y \longrightarrow x + 1 > y + 1$
- $x > y \longrightarrow x + 1 > y + 1$
- $\forall m n. (-(m < n) \land m < n + 1) \longrightarrow m = n$
- $\forall x. \exists y. x + y = 0$
- $\forall y. (\neg p(f(y))) \longleftrightarrow p(f(y))$
- $\forall y. (p(f(y)) \longrightarrow p(f(y + 1)))$
Isabelle/HOL notations: priority, associativity, shorthands

- Here are the logical operators in decreasing order of priority:
  - `=`, `⊤`, `∧`, `∨`, →, ∃
  - «a prioritary operator first chooses its operands»
- For instance
  - `¬¬P = P` means `¬¬(P = P)`!
  - `A ∧ B = B ∧ A` means `A ∧ (B = B) ∧ A`!
  - `P ∧ ∀x.Q(x)` will be parsed as `(P ∧ ∀x.Q(x))`!
  - Hence, write `P ∧ (∀x.Q(x))` instead!
- All binary operators are associative to the right, for instance `A ≠æ B ≠æ C` is equivalent to `A ≠æ (B ≠æ C)`
- Nested quantifications ∀x. ∀y. ∀z. P can be abbreviated into ∀x y z. P
- Free variables are universally quantified, i.e. `P(x)` is equiv. to ∀x. `P(x)`
  - All Isabelle/HOL tools will prefer `P(x)` to ∀x. `P(x)`

First-order logic: satisfiability and models

Definition 21 (Satisfiable formula)
A formula φ is satisfiable if there exists an interpretation I and a valuation V such that `(I, V) |= φ`.

Example 22
Let `φ = p(f(y))` with `F = {f : 1}`, `P = {p : 1}`, `X = {y}`.
The formula φ is satisfiable (there exists `(I, V)` such that `(I, V) |= φ`.
  - Let I be the interp. s.t. `D = {0, 1}`, `pI = λx.(x = 0)`, `fI = λx.x`
  - Let V be the valuation such that `V(y) = 0`
We have `(I, V) |= φ`. With `V'(y) = 1`, `(I, V') ⊭ φ`. Hence, I is not a model of φ.

Property 3
A formula φ is contradictory iff ¬φ is a tautology.

Example 24 (See in Isabelle cm1.thy file)
Let `φ = (∀y. ¬p(f(y))) ↔ (∀y. p(f(y)))`. The formula φ is contradictory and ¬φ is a tautology.

First-order logic: contradictions

Definition 23 (Contradiction)
A formula is contradictory (or unsatisfiable) if it cannot be satisfied, i.e. `(I, V) ⊭ φ` for all interpretation I and all valuation V.

Property 3
A formula φ is contradictory iff ¬φ is a tautology.

Example 24 (See in Isabelle cm1.thy file)
Let `φ = (∀y. ¬p(f(y))) ↔ (∀y. p(f(y)))`. The formula φ is contradictory and ¬φ is a tautology.
Outline

- Terms
  - Types
  - Typed terms
  - $\lambda$-terms
  - Constructor terms
- Functions defined using equations
  - Logic everywhere!
  - Function evaluation using term re-writing
  - Partial functions

Acknowledgements: some slides are borrowed from T. Nipkow’s lectures

Types: syntax

$$\tau ::= (\tau) \mid \text{bool} \mid \text{nat} \mid \text{char} \mid \ldots \quad \text{base types}$$
$$\mid 'a \mid 'b \mid \ldots \quad \text{type variables}$$
$$\mid \tau \Rightarrow \tau \quad \text{functions}$$
$$\mid \tau \times \ldots \times \tau \quad \text{tuples (ascii for $\times$: *)}$$
$$\mid \tau \text{ list} \quad \text{lists}$$
$$\mid \ldots \quad \text{user-defined types}$$

The operator $\Rightarrow$ is right-associative, for instance:

$\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ is equivalent to $\text{nat} \Rightarrow (\text{nat} \Rightarrow \text{bool})$

Typed terms: syntax

$$\text{term ::= (term)} \quad a \in \mathcal{F} \; \text{or} \; a \in \mathcal{X}$$
$$\mid \text{term term} \quad \text{function application}$$
$$\mid \lambda y. \text{term} \quad \text{function definition with} \; y \in \mathcal{X}$$
$$\mid (\text{term}, \ldots, \text{term}) \quad \text{tuples}$$
$$\mid [\text{term}, \ldots, \text{term}] \quad \text{lists}$$
$$\mid (\text{term} :: \tau) \quad \text{type annotation}$$
$$\mid \ldots \quad \text{a lot of syntactic sugar}$$

Function application is left-associative, for instance:

$fa \; b \; c$ is equivalent to $((fa) \; b) \; c$

Example 1 (Types of terms)

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>'a</td>
</tr>
<tr>
<td>$(t1, t2, t3)$</td>
<td>('a $\times$ 'b $\times$ 'c)</td>
</tr>
<tr>
<td>$\lambda y. ; y$</td>
<td>'a $\Rightarrow$ 'a</td>
</tr>
<tr>
<td>$t1$</td>
<td>[t1, t2, t3]</td>
</tr>
<tr>
<td>$\lambda y. ; y ; z. ; z$</td>
<td>'a $\Rightarrow$ 'b $\Rightarrow$ 'b</td>
</tr>
<tr>
<td>$'a$ list</td>
<td></td>
</tr>
</tbody>
</table>

Acknowledgements: some slides are borrowed from T. Nipkow’s lectures
Types and terms: evaluation in Isabelle/HOL

To evaluate a term \( t \) in Isabelle ...

<table>
<thead>
<tr>
<th>Term</th>
<th>Isabelle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>value &quot;True&quot;</td>
<td>True::bool</td>
</tr>
<tr>
<td>value &quot;2&quot;</td>
<td>Error (cannot infer result type)</td>
</tr>
<tr>
<td>value &quot;(2::nat)&quot;</td>
<td>2::nat</td>
</tr>
<tr>
<td>value &quot;[True,False]&quot;</td>
<td>[True,False]::bool list</td>
</tr>
<tr>
<td>value &quot;[(2::nat),6.10]&quot;</td>
<td>Error (cannot infer result type)</td>
</tr>
<tr>
<td>value &quot;[2,6.10]&quot;</td>
<td>[2,6.10]::nat list</td>
</tr>
</tbody>
</table>

Example 2

Terms and functions: semantics is the \( \lambda \)-calculus

Semantics of functional programming languages consists of one rule:

\[
(\lambda x. t) \ a \rightarrow_\beta \ t \{ x \mapsto a \} \quad (\beta\text{-reduction})
\]

where \( t \{ x \mapsto a \} \) is the term \( t \) where all occurrences of \( x \) are replaced by \( a \)

Example 3

Exercise 1 (In Isabelle/HOL)

Use append to concatenate 2 lists of bool, 2 lists of nat, and 3 lists of nat.

Exercise 2 (In Isabelle/HOL)

1. Define the (non-curried) function addNc adding two naturals
2. Use addNc to add 5 to 6
3. Define the (curried) function add adding two naturals
4. Use add to add 5 to 6
5. Using add, define the incr function adding 1 to a natural
6. Apply incr to 5
7. Define a function app1 adding 1 at the beginning of any list of naturals, give an example of use

A word about curried functions and partial application

Definition 5 (Curried function)

A function is curried if it returns a function as result.

Example 6

The function \( (\lambda x. \lambda y. x + y) : \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \) is curried

The function \( (\lambda (x,y). x + y) : \text{nat} \times \text{nat} \Rightarrow \text{nat} \) is not curried

Example 7 (Curried function can be partially applied!)

The function \( (\lambda x. \lambda y. x + y) \) can be applied to 2 or 1 argument:

- \( (\lambda x. \lambda y. x + y) \ 1 \ 2 \rightarrow_\beta \ (\lambda y. 1 + y) \ 2 \rightarrow_\beta \ (1 + 2) : \text{nat} \)
- \( (\lambda x. \lambda y. x + y) \ 1 \rightarrow_\beta \ (\lambda y. 1 + y) : \text{nat} \Rightarrow \text{nat} \) which is a function!
A word about higher-order functions

Definition 8 (Higher-order function)

A higher-order function takes one or more functions as parameters.

Example 9 (Some higher-order functions and their evaluation)

\[
\begin{align*}
\lambda x. \lambda f. f x &:: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b \\
\lambda f. \lambda x. f x &:: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \\
\lambda f. \lambda x. f (x + 1) (x + 1) &:: (nat \Rightarrow nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat \\
\lambda f. \lambda x. f (x + 1) (x + 1) &:: \text{add} 20 \\
\Rightarrow_\beta &\ (\lambda x. \text{add} (x + 1) (x + 1)) \ 20 \\
\Rightarrow_\beta &\ \text{add} (20 + 1) (20 + 1) \\
= &\ (\lambda x.\lambda y. x + y) (20 + 1) (20 + 1) \\
\Rightarrow_\beta &\ (20 + 1) + (20 + 1) \\
= &\ 42
\end{align*}
\]

Interlude: a word about semantics and verification

- To verify programs, formal reasoning on their semantics is crucial!
- To prove a property \( \phi \) on a program \( P \) we need to precisely and exactly understand \( P \)'s behavior.

For many languages the semantics is given by the compiler (version)!

- C, Flash/ActionScript, JavaScript, Python, Ruby, ...

Some languages have a (written) formal semantics:

- Java*, subsets of C (hundreds of pages)
- Proofs are hard because of semantics complexity (e.g. KeY for Java)

Some have a small formal semantics:

- Functional languages: Haskell, subsets of (OCaml, F# and Scala)
- Proofs are easier since semantics essentially consists of a single rule

Exercise 3 (In Isabelle/HOL)

- Define a function \( \text{triple} \) which applies three times a given function to an argument
- Using \( \text{triple} \), apply three times the function \( \text{incr} \) on 0
- Using \( \text{triple} \), apply three times the function \( \text{app1} \) on \([2,3]\)
- Using \( \text{map} :: ('a \Rightarrow 'b) \Rightarrow 'a \ list \Rightarrow 'b \ list \)
  from the list \([1,2,3]\) build the list \([2,3,4]\)

Constructor terms

Isabelle distinguishes between constructor and function symbols

- A function symbol is associated to a function, e.g. \( \text{inc} \)
- A constructor symbol is not associated to any function

Definition 10 (Constructor term)

A term containing only constructor symbols is a constructor term

A constructor term does not contain function symbols
Constructor terms (II)

All data are built using constructor terms without variables ...even if the representation is generally hidden by Isabelle/HOL

Example 11
- Natural numbers of type nat are terms: 0, Suc(0), Suc(Suc(0)), ...
- Integer numbers of type int are couples of natural numbers: ...
- Lists are built using the operators
  - Nil: the empty list
  - Cons: the operator adding an element to the (head) of the list
The term Cons 0 (Cons (Suc 0) Nil) represents the list [0, 1]

Constructor terms: Isabelle/HOL

For most of constructor terms there exists shortcuts:
- Usual decimal representation for naturals, integers and rationals
  1, 2, -3, -45.67676, ...
- [] and # for lists, e.g. Cons 0 (Cons (Suc 0) Nil) = 0#(1#[]) = [0, 1]
- Strings using 2 quotes e.g. "'toto" (instead of "toto")

Exercise 4
1. Prove that 3 is equivalent to its constructor representation
2. Prove that [1, 1] is equivalent to its constructor representation
3. Prove that the first element of list [1, 2] is 1
4. Infer the constructor representation of rational numbers of type rat
5. Infer the constructor representation of strings

Isabelle Theory Library

Isabelle comes with a huge library of useful theories
- Numbers: Naturals, Integers, Rationals, Floats, Reals, Complex ...
- Data structures: Lists, Sets, Tuples, Records, Maps ...
- Mathematical tools: Probabilities, Lattices, Random numbers, ...
All those theories include types, functions and lemmas/theorems

Example 12
Let’s have a look to a simple one Lists.thy:
- Definition of the datatype (with shortcuts)
- Definitions of functions (e.g. append)
- Definitions and proofs of lemmas (e.g. length_append)
  - lemma "length (xs @ ys) = length xs + length ys"
- Exportation rules for SML, Haskell, Ocaml, Scala (code_printing)

Isabelle Theory Library: using functions on lists

Some functions of Lists.thy
- append :: "'a list ⇒ 'a list ⇒ 'a list"
- rev :: "'a list ⇒ 'a list"
- length :: "'a list ⇒ nat"
- map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list"

Exercise 5
1. Apply the rev function to list [1, 2, 3]
2. Prove that for all value x, reverse of the list [x] is equal to [x]
3. Prove that append is associative
4. Prove that append is not commutative
5. Using map, from the list [1, 2, 3] build the list [2, 4, 6]
6. Prove that map does not change the size of a list
Defining functions using equations

- Defining functions using λ-terms is hardly usable for programming
- Isabelle/HOL has a “fun” operator as other functional languages

**Definition 13 (fun operator for defining (recursive) functions)**

\[
\text{fun } f :: "\tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau"
\]

where

\[
" f t_1^1 \ldots t_n^1 = r^1 " \quad | \quad \text{where for all } i = 1 \ldots n \text{ and } k = 1 \ldots m
\]

\[
\ldots
\]

\[
" f t_1^m \ldots t_n^m = r^m " \quad | \quad (t_i^k :: \tau_i) \text{ are constructor terms possibly with variables, and } (r^k :: \tau)
\]

**Example 14 (The member function on lists (2 versions in cm2.thy))**

\[
\text{fun member} :: "'a => 'a list => bool"
\]

where

\[
"\text{member } e [] = \text{False} \mid \text{member } e (x#xs) = (\text{if } e=x \text{ then True else (member } e \text{ xs))}"
\]

---

Total and partial Isabelle/HOL functions

**Definition 15 (Total and partial functions)**

A function is total if it has a value (a result) for all elements of its domain. A function is partial if it is not total.

**Definition 16 (Complete Isabelle/HOL function definition)**

\[
\text{fun } f :: "\tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau"
\]

where

\[
" f t_1^1 \ldots t_n^1 = r^1 " \quad | \quad f \text{ is complete if any call } f t_1 \ldots t_n \text{ with } (t_i :: \tau_i), i = 1 \ldots n \text{ is covered by one case of the definition.}
\]

\[
\ldots
\]

\[
" f t_1^m \ldots t_n^m = r^m " \quad | \quad \text{case of the definition.}
\]

**Example 17 (Isabelle/HOL "Missing patterns" warning)**

When the definition of \( f \) is not complete, an uncovered call of \( f \) is shown.

---

**Theorem 18**

Complete and terminating Isabelle/HOL functions are total, otherwise they are partial.

**Question 1**

Why termination of \( f \) is necessary for \( f \) to be total?

**Remark 1**

All functions in Isabelle/HOL needs to be terminating!
Outline

1 Terms
   ▶ Types
   ▶ Typed terms
   ▶ \(\lambda\)-terms
   ▶ Constructor terms

2 Functions defined using equations
   ▶ Logic everywhere!
   ▶ Function evaluation using term rewriting
   ▶ Partial functions

Acknowledgements: some slides are borrowed from T. Nipkow’s lectures

Logic everywhere!
In the end, everything is defined using logic:
- data, data structures: constructor terms
- properties: lemmas (logical formulas)
- programs: functions (also logical formulas!)

Definition 19 (Equations (or simplification rules) defining a function)
A function \(f\) consists of a set of \(f\).simp\s of equations on terms.

To visualize a lemma/theorem/simplification rule \thm[thm]
For instance: \thm"length_append", \thm"append.simps"
To find the name of a lemma, etc \find[find]
For instance: \find"append" ", + , +"

Exercise 6
Use Isabelle/HOL to find the following formulas:
- definition of \(\text{member}\) (we just defined) and of \(\text{nth}\) (part of List.thy)
- find the lemma relating \(\text{rev}\) (part of List.thy) and \(\text{length}\)

Evaluation = Rewriting using equations

Recall that definition of the function \(\text{member}\) consists of the 2 equations:

(1) \(\text{member} \ e \ \text{[]} = \text{False}\)
(2) \(\text{member} \ e \ (x \ # \ xs) = \text{if } e=x \ \text{then True else } (\text{member} \ e \ xs)\)

How to use those equations to evaluate the term \(\text{member} \ 2 \ [1, 2, 3]\)?

Definition 20 (Substitution)
A substitution \(\sigma\) is a function replacing variables of \(\mathcal{X}\) by terms of \(\mathcal{T}(\mathcal{F}, \mathcal{X})\) in a term of \(\mathcal{T}(\mathcal{F}, \mathcal{X}')\).

Example 21
Let \(\mathcal{F} = \{f : 3, h : 1, g : 1, a : 0\}\) and \(\mathcal{X} = \{x, y, z\}\.
Let \(\sigma = \{x \mapsto g(a), y \mapsto h(z)\}\) and \(t = f(h(x), x, g(y))\). We have \(\sigma(t) = f(h(g(a)), g(a), g(h(z)))\).

Remark 2
Isabelle/HOL rewrites terms using equations in the order of the function definition and only from left to right.
Evaluation= Rewriting using equations (III)

(1) \text{member e} \emptyset = \text{False}
(2) \text{member e} \ (x \# xs) = (\text{if e=x then True else (member e xs)})

Evaluation of test: member 2 \ [1,2,3]

\[ 
\begin{align*}
\text{if 2=1 then True else (member 2 \ [2,3])} \\
\text{by equation (2), because [1,2,3] = 1\#[2,3]} \\
\text{if False then True else (member 2 \ [2,3])} \\
\text{by Isabelle equations defining equality on naturals} \\
\text{member 2 \ [2,3]} \\
\text{by Isabelle equation (if False then x else y = y)} \\
\text{if 2=2 then True else (member 2 \ [3])} \\
\text{by equation (2), because [2,3] = 2\#[3]} \\
\text{if True then True else (member 2 \ [3])} \\
\text{by Isabelle equations defining equality on naturals} \\
\text{True} \\
\text{by Isabelle equation (if True then x else y = x)}
\end{align*}
\]

Lemma simplification= Rewriting + Logical deduction (II)

(1) \text{member e} \emptyset = \text{False}
(2) \text{member e} \ (x \# xs) = (\text{if e=x then True else (member e xs)})

(3) append \emptyset x = x
(4) append (x \# xs) y = x \# (append xs y)

Exercise 7
Is it possible to prove the lemma \text{member u} \ (append \ [u] \ v) by simplification/rewriting?

Exercise 8
Is it possible to prove the lemma \text{member v} \ (append u \ [v]) by simplification/rewriting?

Evaluation of partial functions
Evaluation of partial functions using rewriting by equational definitions may not result in a constructor term

Exercise 9
Let index be the function defined by:

fun index :: "'a => 'a list => nat"

where

"index y (x#xs) = (if x=y then 0 else 1+(index y xs))"

- Define the function in Isabelle/HOL
- What does it computes?
- Why is index a partial function? (What does Isabelle/HOL says?)
- For index, give an example of a call whose result is:
  - a constructor term
  - a match failure
- Define the property relating functions index and List.nth
To export functions to Haskell, SML, Ocaml, Scala ....... export_code

For instance, to export the member and index functions to Scala:

```scala
object cm2 {
    def member[A : HOL.equal](e: A, x1: List[A]): Boolean =
        (e, x1) match {
            case (e, Nil) => false
            case (e, x :: xs) => (if (HOL.eq[A](e, x)) true
                                  else member[A](e, xs))
        }
    def index[A : HOL.equal](y: A, x1: List[A]): Nat =
        (y, x1) match {
            case (y, x :: xs) =>
                (if (HOL.eq[A](x, y)) Nat(0)
                 else Nat(1) + index[A](y, xs))
        }
}
```

T. Genet (ISTI/IRISA)
Recursion everywhere... and nothing else

«Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem»

- The «bad» news: in Isabelle/HOL, there is no while, no for, no mutable arrays and no pointers,...
- The good news: you don’t really need them to program!
- The second good news: programs are easier to prove without all that!

In Isabelle/HOL all complex types and functions are defined using recursion

- What theory says: expressive power of recursive-only languages and imperative languages is equivalent
- What OCaml programmers say: it is as it should always be
- What Java programmers say: may be tricky but you will always get by

Outline

1 Recursive functions
   - Definition
   - Termination proofs with measures
   - Difference between fun, function and primrec
2 (Recursive) Algebraic Data Types
   - Defining Algebraic Data Types using datatype
   - Building objects of Algebraic Data Types
   - Matching objects of Algebraic Data Types
   - Type abbreviations

Acknowledgements:
some material is borrowed from T. Nipkow and S. Blazy’s lectures
Terminating Recursive Functions
In Isabelle/HOL, all the recursive functions have to be terminating!

How to guarantee the termination of a recursive function? (practice)
- Needs at least one base case (non-recursive case)
- Every recursive case must go towards a base case
- ... or every recursive case «decreases» the size of one parameter

How to guarantee the termination of a recursive function? (theory)
- If $f : \tau_1 \to \ldots \to \tau_n \to \tau$ then define a measure function
  $g : \tau_1 \times \ldots \times \tau_n \Rightarrow \mathbb{N}$
- Prove that the measure of all recursive calls is decreasing
  To prove termination of $f : f(t_1) \Rightarrow f(t_2) \Rightarrow \ldots$ 
  Prove that $g(t_1) > g(t_2) > \ldots$
- The ordering $>$ is well founded on $\mathbb{N}$
  i.e. no infinite decreasing sequence of naturals $n_1 > n_2 > \ldots$

Example 1 (Proving termination using a measure)
"member e [] = False" |
"member e (x#xs) = (if e=x then True else (member e xs))"
- We define the measure $g = \lambda x. (\text{length } y)$
- We prove that $\forall e \in xs. (g (e (x\#xs))) > (g e xs)$

Terminating Recursive Functions (III)

How to guarantee the termination of a recursive function? (Isabelle/HOL)
- Define the recursive function using fun
- Isabelle/HOL automatically tries to build a measure\(^1\)
- If no measure is found the function is rejected
- If it is not terminating, make it terminating!
- Try to modify it so that its termination is easier to show

Otherwise
- Re-define the recursive function using function (sequential)
- Manually give a measure to achieve the termination proof

Example 2
A definition of the member function using function is the following:
function (sequential) member::"'a \Rightarrow 'a list \Rightarrow bool"
where
"member e [] = False" |
"member e (x\#xs) = (if e=x then True else (member e xs))"
apply pat_completeness
apply auto
done
Prove that the function is "complete"
 i.e. patterns cover the domain

termination member
apply (relation "measure ($\lambda (x,y). (\text{length } y)$)")
apply auto
done
Prove its termination using the measure proposed in Example 1

\(^1\)Actually, it tries to build a termination ordering but it has the same objective.
Terminating Recursive Functions (V)

Exercise 1
Define the following functions, see if they are terminating. If not, try to modify them so that they become terminating.

fun f::"nat => nat"
where
"f x=f (x - 1)"

fun f2::"int => int"
where
"f2 x = (if x=0 then 0 else f2 (x - 1))"

fun f3::"nat => nat => nat"
where
"f3 x y= (if x >= 10 then 0 else f3 (x + 1) (y + 1))"

Terminating Recursive Functions (VI)

Automatic termination proofs (fun definition) are generally enough
- Covers 90% of the functions commonly defined by programmers
- Otherwise, it is generally possible to adapt a function to fit this setting
Most of the functions are terminating by construction (primitive recursive)

Definition 3 (Primitive recursive functions: primrec)
Functions whose recursive calls «peels off» exactly one constructor

Example 4 (member can be defined using primrec instead of fun)
primrec member:: "'a => 'a list => bool"
where
"member e [] = False" |
"member e (x#xs) = (if e=x then True else (member e xs))"

For instance, in List.thy:
- 26 "fun", 34 "primrec" with automatic termination proofs
- 3 "function" needing measures and manual termination proofs.

Recursive functions, exercises

Exercise 2
Define the following recursive functions
- A function sumList computing the sum of the elements of a list of naturals
- A function sumNat computing the sum of the n first naturals
- A function makeList building the list of the n first naturals

State and verify a lemma relating sumList, sumNat and makeList

Outline

1. Recursive functions
   - Definition
   - Termination proofs with orderings
   - Termination proofs with measures
   - Difference between fun, function and primrec

2. (Recursive) Algebraic Data Types
   - Defining Algebraic Data Types using datatype
   - Building objects of Algebraic Data Types
   - Matching objects of Algebraic Data Types
   - Type abbreviations
(Recursive) Algebraic Data Types

Basic types and type constructors (list, \(\Rightarrow\), *) are not enough to:
- Define enumerated types
- Define unions of distinct types
- Build complex structured types

Like all functional languages, Isabelle/HOL solves those three problems using one type construction: Algebraic Data Types (sum-types in OCaml)

**Definition 5 (Isabelle/HOL Algebraic Data Type)**

To define type \(\tau\) parameterized by types \((\alpha_1, \ldots, \alpha_n)\):
\[
\text{datatype } (\alpha_1, \ldots, \alpha_n)\tau = C_1 \tau_1,1 \ldots \tau_1, m_1 | \ldots | C_k \tau_k, 1 \ldots \tau_k, n_k
\]
with \(C_1, \ldots, C_n\) capitalized identifiers

**Example 6 (The type of (polymorphic) lists, defined using datatype)**
\[
\text{datatype 'a list = Nil | Cons 'a 'a list}
\]
defines constructors \(Nil::'a list\) and \(Cons::'a \Rightarrow 'a list \Rightarrow 'a list\)
Hence,
- \(Cons (3::nat)\) is an object of type \(nat list\)
- \(Cons (3::nat)\) is an object of type \(nat list \Rightarrow nat list\)

Matching objects of Algebraic Data Types

Objects of Algebraic Data Types can be matched using case expressions:
- \((\text{case } l \text{ of } \text{Nil } \Rightarrow \ldots | (\text{Cons } x \ r) \Rightarrow \ldots)\)
- possibly with wildcards, i.e. "_"
- \((\text{case } i \text{ of } 0 \Rightarrow \ldots | (\text{Suc } _) \Rightarrow \ldots)\)
and nested patterns
- \((\text{case } l \text{ of } (\text{Cons } 0 \ \text{Nil}) \Rightarrow \ldots | (\text{Cons } (\text{Suc } x) \ \text{Nil}) \Rightarrow \ldots)\)
- possibly embedded in a function definition

**Algebraic Data Types, exercises**

**Exercise 3**
Define the following types and build an object of each type using value
- The enumerated type color with possible values: black, white and grey
- The type token union of types string and int
- The type of (polymorphic) binary trees whose elements are of type 'a

Define the following functions
- A function notBlack that answers true if a color object is not black
- A function sumToken that gives the sum of two integer tokens and 0 otherwise
- A function merge::color tree \(\Rightarrow\) color that merges all colors in a color tree (leaf is supposed to be black)
Type abbreviations

In Isabelle/HOL, it is possible to define abbreviations for complex types. To introduce a type abbreviation, use `type_synonym`.

For instance:

- `type_synonym name="(string * string)"
- `type_synonym ('a,'b) pair="('a * 'b)"

Using those abbreviations, objects can be explicitly typed:

- `value "('Leonard','Michalon')::name"
- `value "(1,'toto')::(nat,string)pair"

... though the type synonym name is ignored in Isabelle/HOL output.
Prove logic formulas ... to prove programs

fun nth:: "nat ⇒ 'a list ⇒ 'a"
where
"nth 0 (x#_)=x" |
"nth x (y#ys)= (nth (x - 1) ys)"

fun index:: "'a ⇒ 'a list ⇒ nat"
where
"index x (y#ys)= (if x=y then 1 else 1+(index x ys))"

lemma nth_index: "nth (index e l) l= e"

How to prove the lemma nth_index? (Recall that everything is logic!)

What we are going to prove is thus a formula of the form:

Theory of lists ∧ Equations for nth ∧ Equations for index → nth_index

Finding counterexamples

Why? because «90% of the theorems we write are false!»
- Because this is not what we want to prove!
- Because the formula is imprecise
- Because the function is false
- Because there are typos...

Before starting a proof, always first search for a counterexample!

Isabelle/HOL offers two counterexample finders:
- nitpick: uses finite model enumeration
  - Works on any logic formula, any type and any function
  - Rapidly exhausted on large programs and properties
- quickcheck: uses random testing, exhaustive testing and narrowing
  - Does not covers all formula and all types
  - Scales well even on large programs and complex properties

Outline

1. Finding counterexamples
   - nitpick
   - quickcheck

2. Proving true formulas
   - Proof by cases: apply (case_tac x)
   - Proof by induction: apply (induct x)
   - Combination of decision procedures: apply auto and apply simp
   - Solving theorems in the Cloud: sledgehammer

Acknowledgements: some material is borrowed from T. Nipkow’s lectures and from Concrete Semantics by Nipkow and Klein, Springer Verlag, 2016.

More details (in french) about those proof techniques can be found in:
- The video https://youtu.be/qwlQIS46TLA
Nitpick

To build an interpretation \( I \) such that \( I \not\models \phi \) (or \( I \models \neg \phi \)) ....... nitpick

nitpick principle: build an interpretation \( I \models \neg \phi \) on a finite domain \( D \)
- Choose a cardinality \( k \)
- Enumerate all possible domains \( D \) of size \( k \) for all types \( \tau \) in \( \neg \phi \)
- Build all possible interpretations of functions in \( \neg \phi \) on all \( D \)
- Check if one interpretation satisfy \( \neg \phi \) (this is a counterexample for \( \phi \))
- If not, there is no counterexample on a domain of size \( k \) for \( \phi \)

nitpick algorithm:
- Search for a counterexample to \( \phi \) with cardinalities 1 upto \( n \)
- Stops when \( I \models \neg \phi \) is found (counterex. to \( \phi \)), or
- Stops when maximal cardinality \( n \) is reached (10 by default), or
- Stops after 30 seconds (default timeout)

Exercise 2
- Explain the counterexample found for \( \text{rev } 1 = 1 \)
- Is there a counterexample to the lemma \( \text{nth\_index} \) ?
- Correct the lemma and definitions of \( \text{nth\_index} \) and \( \text{nth} \)
- Is the lemma \( \text{append\_commut} \) true? really?

Quickcheck

To build an interpretation \( I \) such that \( I \not\models \phi \) (or \( I \models \neg \phi \)) ....... quickcheck

quickcheck principle: test \( \phi \) with automatically generated values of size \( k \)
Either with a generator
- Random: values are generated randomly (Haskell’s QuickCheck)
- Exhaustive: (almost) all values of size \( k \) are generated
- Narrowing: like exhaustive but taking advantage of symbolic values

No exhaustiveness guarantee!! with any of them

quickcheck algorithm:
- Export Haskell code for functions and lemmas
- Generate test values of size 1 upto \( n \) and, test \( \phi \) using Haskell code
- Stops when a counterexample is found, or
- Stops when max. size of test values has been reached (default 5), or
- Stops after 30 seconds (default timeout)
Quickcheck (II)

quickcheck options:
- `timeout=t`, set the timeout to `t` seconds
- `expect=s`, specifies the expected outcome where `s` can be `no_counterexample`, `counterexample` or `no_expectation`
- `tester=tool`, specifies generator to use where `tool` can be `random`, `exhaustive` or `narrowing`
- `size=i`, specifies the maximal size of testing values

For instance: `quickcheck [tester=narrowing,size=6]

Exercise 3 (Using quickcheck)

- *find a counterexample on TP0 (solTP0.thy, CM4_TP0)*
- *find a counterexample for length_slice*

Remark 2

Quickcheck first generates values and then does the tests. As a result, it may not run the tests if you choose bad values for size and timeout.

What to do next?

When no counterexample is found what can we do?
- Increase the timeout and size values for nitpick and quickcheck?
- ... go for a proof!

Any proof is faster than an infinite time nitpick or quickcheck.
Any proof is more reliable than an infinite time nitpick or quickcheck (They make approximations or assumptions on infinite types)

The five proof tools that we will focus on:
- `apply case_tac`
- `apply induct`
- `apply auto`
- `apply simp`
- `sledgehammer`

Proof by cases

... possible when the proof can be split into a finite number of cases

Proof by cases on a formula `F`

Do a proof by cases on a formula `F` ........ apply (case_tac "F")
Splits the current goal in two: one with assumption `F` and one with `¬ F`

Example 2 (Proof by case on a formula)

With apply (case_tac "F::bool")

goal (1 subgoal): becomes goal (2 subgoals):

1. `F ==> A ==> B`
2. `¬ F ==> A ==> B`

Exercise 4

Prove that for any natural number `x`, if `x < 4` then `x * x < 10`.

How do proofs look like?

A formula of the form `A_1 ∧ ... ∧ A_n` is represented by the proof goal:

```
goal (n subgoals):
1. A_1
...n. A_n
```

Where each subgoal to prove is either a formula of the form

```
∧ x_1 ... x_n, B    meaning prove B, or
∧ x_1 ... x_n, B → C meaning prove B  →  C, or
∧ x_1 ... x_n, B_1 ... B_n → C meaning prove B_1 ∧ ... ∧ B_n → C
```

and `∧ x_1 ... x_n` means that those variables are local to this subgoal.

Example 1 (Proof goal)

```
goal (2 subgoals):
1. member [] e → nth (index e []) [] = e
2. ∀ a l. e ≠ a → member (a # l) e →
   ¬ member l e → nth (index e l) l = e
```
Proof by induction (II)

Proof by induction on a variable x of an enumerated type of size n
Do a proof by cases on a variable x ............ apply (case_tac "x")
Splits the current goal into n goals, one for each case of x.

Example 3 (Proof by case on a variable of an enumerated type)
In Course 3, we defined datatype color = Black | White | Grey
With apply (case_tac "x")
goal (1 subgoal):
1. P (x::color)

Example 4
Recall the definition of the function notBlack x
On the color enumerated type or course 3, show that for all color x if the
notBlack x is true then x is either white or grey.

Example 5 (Choice of the induction variable)
(1) append [] l = l
(2) append (x#xs) l = x#(append xs l)
To prove \( \forall l \in \text{'a list}. (append l []) = l \) by induction on l, we prove:
\( \forall e \in \text{'a}. \forall l \in \text{'a list}. (append l l) = e#(append xs l) \)

Example 6 (Choice of the induction variable)
(1) append [] l = l
(2) append (x#xs) l = x#(append xs l)
To prove \( \forall l_1 l_2 \in \text{'a list}. (length append l_1 l_2) \geq (length l_2) \)
An induction proof on \( l_1 \), instead of \( l_2 \), is more likely to succeed:
• an induction on \( l_1 \) will require to prove:
  \( (length append (e#l_1) l_2) \geq (length l_2) \)
• an induction on \( l_2 \) will require to prove:
  \( (length append l_1 (e#l_2)) \geq (length (e#l_2)) \)
Proof by induction: apply \((\text{induct } x)\) (II)

Exercise 6
Recall the datatype of binary trees we defined in lecture 3. Define and prove the following properties:
1. If \(\text{member } x \ t\), then there is at least one node in the tree \(t\).
2. Relate the fact that \(x\) is a sub-tree of \(y\) and their number of nodes.

Exercise 7
Recall the functions \(\text{sumList}\), \(\text{sumNat}\) and \(\text{makeList}\) of lecture 3. Try to state and prove the following properties:
1. Relate the length of list produced by \(\text{makeList } i\) and \(i\)
2. Relate the value of \(\text{sumNat } i\) and \(i\)
3. Give and try to prove the property relating those three functions

Proof by induction: generalize the goals
By default \(\text{apply induct}\) may produce too weak induction hypothesis

Example 7
When doing an \(\text{apply (induct } x)\) on the goal \(P (x::nat) (y::nat)\) goal (2 subgoals):
1. \(P \ 0 \ y\)
2. \(\forall x. \ P \ x \ y \Rightarrow P \ (\text{Suc } x) \ y\)

Example 8
With \(\text{apply (induct } x \ \text{arbitrary}\ y\) on the same goal
goal (2 subgoals):
1. \(\forall y. \ P \ 0 \ y\)
2. \(\forall x \ y. \ P \ x \ y \Rightarrow P \ (\text{Suc } x) \ y\)

Exercise 8
Prove the sym lemma on the leq function.

Proof by induction: induction principles
Recall the basic induction principle on naturals:
\[
P(0) \land \forall x \in \mathbb{N}. (P(x) \rightarrow P(x+1)) \rightarrow \forall x \in \mathbb{N}. P(x)
\]
In fact, there are infinitely many other induction principles
- \(P(0) \land P(1) \land \forall x \in \mathbb{N}. ((x > 0 \land P(x)) \rightarrow P(x+1)) \rightarrow \forall x \in \mathbb{N}. P(x)\)
- ...
- Strong induction on naturals
  \(\forall x, y \in \mathbb{N}. ((y < x \land P(y)) \rightarrow P(x)) \rightarrow \forall x \in \mathbb{N}. P(x)\)
- Well-founded induction on any type having a well-founded order \(<<\
  \forall x, y. ((y << x \land P(y)) \rightarrow P(x)) \rightarrow \forall x. P(x)\)

Exercise 9
Prove the lemma on function \(\text{div2}\).
Combination of decision procedures auto and simp

Automatically solve or simplify all subgoals .......... apply auto
apply auto does the following:
- Rewrites using equations (function definitions, etc)
- Applies a bit of arithmetic, logic reasoning and set reasoning
- On all subgoals
  - Solves them all or stops when stuck and shows the remaining subgoals

Automatically simplify the first subgoal .......... apply simp
apply simp does the following:
- Rewrites using equations (function definitions, etc)
- Applies a bit of arithmetic
- on the first subgoal
  - Solves it or stops when stuck and shows the simplified subgoal

Want to know what those tactics do?
- Add the command using [[simp_trace=true]] in the proof script
- Look in the output buffer

Example 9
Switch on tracing and try to prove the lemma:
lemma "(index (1::nat) [3,4,1,3]) = 2"
using [[simp_trace=true]]
apply auto

Sledgehammer

«Sledgehammers are often used in destruction work...»

Sledgehammer

«Solve theorems in the Cloud»

Architecture:

<table>
<thead>
<tr>
<th>Isabelle/HOL</th>
<th>Formula to prove + relevant definitions and lemmas</th>
<th>External ATPs</th>
<th>Proof (click on it)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Local or in the Cloud</td>
<td></td>
</tr>
</tbody>
</table>

Prove the first subgoal using state-of-the-art ATPs ...... sledgehammer
- Call to local or distant ATPs: SPASS, E, Vampire, CVC4, Z3, etc.
- Succeeds or stops on timeout (can be extended, e.g. [timeout=120])
- Provers can be explicitly selected (e.g. [provers= z3 spass])
- A proof consists of applications of lemmas or definition using the Isabelle/HOL tactics: metis, smt, simp, fast, etc.

1Automatic Theorem Provers
2See http://www.tptp.org/CASC/.
Remark 3
By default, sledgehammer does not use all available provers. But, you can remedy this by defining, once for all, the set of provers to be used:
```
sledgehammer_params [provers=cvc4 spass z3 e vampire]
```

Exercise 10
Finish the proof of the property relating \texttt{nth} and \texttt{index}

Exercise 11
Recall the functions \texttt{sumList}, \texttt{sumNat} and \texttt{makeList} of lecture 3. Try to state and prove the following properties:

1. Prove that there is no repeated occurrence of elements in the list produced by \texttt{makeList}
2. Finish the proof of the property relating those three functions

Hints for building proofs in Isabelle/HOL

When stuck in the proof of \texttt{prop1}, add relevant intermediate lemmas:

- In the file, define a lemma \textbf{before} the property \texttt{prop1}
- \textbf{Name} the lemma (say \texttt{lem1}) (to be used by sledgehammer)
- Try to find a counterexample to \texttt{lem1}
- If no counterexample is found, close the proof of \texttt{lem1} by \texttt{sorry}
- Go back to the proof of \texttt{prop1} and check that \texttt{lem1} helps
- If it helps then prove \texttt{lem1}. If not try to guess another lemma

To build correct theories, do not confuse \texttt{oops} and \texttt{sorry}:

- Always close an \textbf{unprovable} property by \texttt{oops}
- Always close an unfinished proof of a \textbf{provable} property by \texttt{sorry}

Example 10 (Everything is provable using contradictory lemmas)
We can prove that $1 + 1 = 0$ using a false lemma.
Scala in a nutshell

- "Scalable language": small scripts to architecture of systems
- Designed by Martin Odersky at EPFL
  - Programming language expert
  - One of the designers of the Java compiler
- Pure object model: only objects and method calls (≠ Java)
- With functional programming: higher-order, pattern-matching, ...
- Fully interoperable with Java (in both directions)
- Concise smart syntax (≠ Java)
- A compiler and a read-eval-print loop integrated into the IDE

Scala in a nutshell

Outline

1 Basics
   - Base types and type inference
   - Control: if and match - case
   - Loops (for) and structures: Lists, Tuples, Maps
2 Functions
   - Basic functions
   - Anonymous, Higher order functions and Partial application
3 Object Model
   - Class definition and constructors
   - Method/operator/function definition, overriding and implicit defs
   - Traits and polymorphism
   - Singleton Objects
   - Case classes and pattern-matching
4 Interactions with Java
   - Interoperability between Java and Scala
5 Isabelle/HOL export in Scala

Bibliography

- An Overview of the Scala Programming Language, M. Odersky & al. 

Acknowledgements

Many thanks to J. Noyé and J. Richard-Foy for providing material, answering questions and for fruitful discussions.
Outline

1 Basics
   - Base types and type inference
   - Control: if and match - case
   - Loops (for) and structures: Lists, Tuples, Maps

2 Functions
   - Basic functions
   - Anonymous, Higher order functions and Partial application

3 Object Model
   - Class definition and constructors
   - Method/operator/function definition, overriding and implicit def
   - Traits and polymorphism
   - Singleton Objects
   - Case classes and pattern-matching

4 Interactions with Java
   - interoperability between Java and Scala

5 Isabelle/HOL export in Scala

Base types and type annotations

- 1: Int, "toto": String, 'a': Char, (): Unit

- Every data is an object, including base types!
  e.g. 1 is an object and Int is its class

- Every access/operation on an object is a method call!
  e.g. 1 + 2 executes: 1.+(2) (o.x(y) is equivalent to o x y)

Exercise 1
Use the max(Int) method of class Int to compute the maximum of 1+2 and 4.
val and var
- `val` associates an object to an identifier and cannot be reassigned
- `var` associates an object to an identifier and can be reassigned
- Scala philosophy is to use `val` instead of `var` whenever possible
- Types are (generally) automatically inferred

```
scala> val x=1  // or val x:Int = 1
x: Int = 1

scala> x=2
<console>:8: error: reassignment to val
     x=2
     ^

scala> var y=1
y: Int = 1

scala> y=2
y: Int = 2
```
if expressions

- Syntax is similar to Java if statements ...
  but that they are not statements but typed expressions
- if ( condition ) e1 else e2
  Remark: the type of this expression is the supertype of e1 and e2
- if ( condition ) e1 // else ()
  Remark: the type of this expression is the supertype of e1 and Unit

Exercise 2

What are the results and types of the corresponding if expressions:

- if (1==2) 1 else 2
- if (1==2) 1 else "toto"
- if (1==1) 1
- if (1==1) println(1)

match - case expressions

- Replaces (and extends) the usual switch - case construction
- The syntax is the following:
  e match {
    case pattern1 => r1 //patterns can be constants
    case pattern2 => r2 //or terms with variables
    ... //or terms with holes: '_'
    case _ => rn
  }
- Remark: the type of this expression is the supertype of r1, r2, ...rn

Example 1 (Match-case expressions)

x match {
  case "bonjour" => "hello"
  case "au revoir" => "goodbye"
  case _ => "don't know"
}

(Immutable) Lists: List[A]

- List definition (with type inference)
  val l = List(1,2,3,4,5)
- Adding an element to the head of a list
  val l1 = 0::l
- Adding an element to the queue of a list
  val l2 = l1:+6
- Concatenating lists
  val l3 = l1++l2
- Getting the element at a given position
  val x = l2(2)
- Doing pattern-matching over lists
  12 match {
    case Nil => 0
    case e::Nil => e
  }

for loops

- for (ident <- s) e
  Remark: s has to be a subtype of Traversable
  (Arrays, Collections, Tables, Lists, Sets, Ranges, ...)
- Usual for-loops can be built using .to(...)
  "(1).to(5)" "1 to 5" results in Range(1, 2, 3, 4, 5)

Exercise 3

Given val lb=List(1,2,3,4,5) and using for, build the list of squares of lb.

Exercise 4

Using for and println build a usual 10 \times 10 multiplication table.
(Immutable) Tuples : (A,B,C,...)

- Tuple definition (with type inference)
  scala> val t= (1,"toto",18.3)
  t: (Int, String, Double) = (1,toto,18.3)
- Tuple getters: t._1, t._2, etc.
- ... or with match - case:
  t match { case (2,"toto",_) => "found!" 
    case (_,x,_) => x 
  }

The above expression evaluates in "toto"

(Immutable) maps : Map[A,B]

- Map definition (with type inference)
  val m= Map('C' -> "Carbon","H' -> "Hydrogen")
  Remark: inferred type of m is Map[Char,String]
- Finding the element associated to a key in a map, with default value
  m.getOrElse('K","Unknown")
- Adding an association in a map
  val m1= m+('O' -> "Oxygen")
- A Map[A,B] can be traversed (using for) as a Collection of pairs
  of type Tuple[A,B], e.g. for((k,v) <- m){ ... }

Exercise 5
Print all the keys of map m1

Outline

1 Basics
   - Base types and type inference
   - Control : if and match - case
   - Loops (for) and structures: Lists, Tuples, Maps

2 Functions
   - Basic functions
   - Anonymous, Higher order functions and Partial application

Basic functions

- def f ( arg1: Type1, ... , argn:Typen ): Typef = { e }
  Remark 1: type of e (the type of the last expression of e) is Typef
  Remark 2: Typef can be inferred for non recursive functions
  Remark 3: The type of f is : (Type1,...,Typen) Typef

Example 2

def plus(x:Int,y:Int):Int={
  println("Sum of "+x+" and "+y+" is equal to "+(x+y))
  x+y // no return keyword 
} // the result of the function is the last expression

Exercise 6
Using a map, define a phone book and the functions
addName(name:String,tel:String), getTel(name:String):String,
getUserList:List[String] and getTelList:List[String].
Anonymous functions and Higher-order functions

- The anonymous Scala function adding one to x is:
  \[(x:\text{Int}) \Rightarrow x + 1\]
  Remark: it is written \((\lambda x \cdot x + 1)\) in Isabelle/HOL.
- A higher order function takes a function as a parameter
  e.g. method/function `map` called on a List[A] takes a function \((A \Rightarrow B)\) and results in a List[B]

```scala
scala> val l=List(1,2,3)
l: List[Int] = List(1, 2, 3)

scala> l.map ((x:Int) => x+1)
res1: List[Int] = List(2, 3, 4)
```

Exercise 7

**Using** `map` **and the** `capitalize` **method of the class String**, define the `capUserList` function returning the list of capitalized user names.

Partial application

- The `'` symbol permits to partially apply a function
  e.g. `getTel(.)` returns the function associated to `getTel`

**Example 3** (Other examples of partial application)

\[(_:\text{String}).size \:_:\text{Int} + (_:\text{Int} \:_:\text{String}) == "toto"\]

Exercise 8

Using `map` and partial application on `capitalize`, redefine the function `capUserList`.

Exercise 9

Using the higher order function `filter` on Lists, define a function `above(n:String):List(String)` returning the list of users having a capitalized name greater to name `n`.

Class definition and constructors

- **class** `C` \((v1: \text{type1}, \ldots, vn: \text{typen})\) \{ \ldots \}
  the primary constructor
  e.g. **class** `Rational(n: \text{Int}, d: \text{Int})`\{
    val num=n       // can use var instead
    val den=d       // to have mutable objects
    def isNull:Boolean=(num==0)
  \}
  
  Objects instances can be created using **new**:
  ```
  val r1= new Rational(3,2)
  ```

Exercise 10

Complete the `Rational` class with an `add(r: Rational): Rational` function.
Overriding, operator definitions and implicit conversions

- Overriding is explicit: `override def f(...)`

Exercise 11
Redefine the `toString` method of the `Rational` class.

- All operators '+', '*', '==', '>', ... can be used as function names
e.g. `def +(x:Int):Int= ...`
Remark: when using the operator recall that `x.+(y) = x + y`

Exercise 12
Define the `+' and '*' operators for the class `Rational`.

- It is possible to define implicit (automatic) conversions between types
e.g. `implicit def bool2int(b:Boolean):Int= if b 1 else 0`

Exercise 13
Add an implicit conversion from `Int` to `Rational`.

Traits

- Traits stands for interfaces (as in Java)
  ```scala
  trait IntQueue {
    def get:Int
    def put(x:Int):Unit
  }
  ```
- The keyword `extends` defines trait implementation
  ```scala
  class MyIntQueue extends IntQueue{
    private var b= List[Int]()
    def get= {val h=b(0); b=b.drop(1); h}
    def put(x:Int)= {b=b:+x}
  }
  ```

Singleton objects

- Singleton objects are defined using the keyword `object`
  ```scala
  trait IntQueue {
    def get:Int
    def put(x:Int):Unit
  }
  ```
  ```scala
  object InfiniteQueueOfOne extends IntQueue{
    def get=1
    def put(x:Int)={}
  }
  ```
- A singleton object does not need to be “created” by `new`
  ```scala
  InfiniteQueueOfOne.put(10)
  InfiniteQueueOfOne.put(15)
  val x=InfiniteQueueOfOne.get
  ```

Type abstraction and Polymorphism
Parameterized function/class/trait can be defined using type parameters

```scala
trait Queue[T]{
  // more generic than IntQueue
  def get:T
  def push(x:T):Unit
}
```

```scala
class MyQueue[T] extends Queue[T]
{
  protected var b= List[T]()
  def get= {val h=b(0); b=b.drop(1); h}
  def put(x:T)= {b=b:+x}
}
```
Case classes

- Case classes provide a natural way to encode Algebraic Data Types e.g. binary expressions built over rationals: $\frac{18}{27} + -\frac{1}{2}$

```scala
trait Expr
case class BinExpr(o:String,l:Expr,r:Expr) extends Expr
case class Constant(r:Rational) extends Expr
case class Inv(e:Expr) extends Expr
```

- Instances of case classes are built without `new` e.g. the object corresponding to $\frac{18}{27} + -\frac{1}{2}$ is built using:

  ```scala
BinExpr("+",Constant(new Rational(18,27)), Inv(Constant(new Rational(1,2))))
  ```

Case classes and pattern-matching

```scala
def getOperator(e:Expr):String= {
  e match {
    case BinExpr(_,_,_) => o
    case _ => "No operator"
  }
}
```

Exercise 14

Define an `eval(e:Expr):Rational` function computing the value of any expression.

Interoperability between Java and Scala

- In Scala, it is possible to build objects from Java classes e.g. `val txt:JTextArea=new JTextArea("")`

- And to define scala classes/objects implementing Java interfaces e.g. `object Window extends JFrame`

- There exists conversions between Java and Scala data structures

  ```scala
  import scala.collection.JavaConverters._
  val li:java.util.List[Int]= new java.util.ArrayList[Int](){
    li.add(1); li.add(2); li.add(3) // li: java.util.List[Int]
  };
  val sb1= li.asScala.toList // sl1: List[Int]
  val sl1= sb1.asJava // sl1: java.util.List[Int]
  ```

- Remark: it is also possible to use Scala classes and Object into Java
Outline

1 Basics
   - Base types and type inference
   - Control: if and match - case
   - Loops (for) and structures: Lists, Tuples, Maps

2 Functions
   - Basic functions
   - Anonymous, Higher order functions and Partial application

3 Object Model
   - Class definition and constructors
   - Method/operator/function definition, overriding and implicit defs
   - Traits and polymorphism
   - Singleton Objects
   - Case classes and pattern-matching

4 Interactions with Java
   - Interoperability between Java and Scala

5 Isabelle/HOL export in Scala

---

Isabelle/HOL exports Scala case classes and functions...

```scala
theory tp
[
  ...
  datatype 'a tree = Leaf | Node "'a * 'a tree * 'a tree"
  fun member:: "'a => 'a tree => bool"
  where
    "member _ Leaf = False" |
    "member x (Node(y,1,r)) = (if x=y then True else ((member x l)"
    
----------to Scala----------

object tp {
  abstract sealed class tree[+A] // similar to traits
  case object Leaf extends tree[Nothing]
  case class Node[+A](a: (A, (tree[A], tree[A]))) extends tree[A]
  def member[A : HOL.equal](uu: A, x1: tree[A]): Boolean =
    (uu, x1) match {
      case (uu, Leaf) => false
      case (x, Node((y, (l, r)))) => (if (HOL.eq[A](x, y)
        else member[A](x, l) || member[A](x, r))

----------

... and some more cryptic code for Isabelle/HOL equality

```scala
object HOL {
  trait equal[A] {
    val 'HOL.equal': (A, A) => Boolean
  }

  def equal[A](a: A, b: A)(implicit A: equal[A]): Boolean =
    A.'HOL.equal'(a, b)

  def eq[A : equal](a: A, b: A): Boolean = equal[A](a, b)
}

To link Isabelle/HOL code and Scala code, it can be necessary to add:
(Demo tp.thy + CMS5Integ)

```
implicit def equal_t[T]: HOL.equal[T] = new HOL.equal[T] {
  val 'HOL.equal' = (a: T, b: T) => a==b
}

Which defines HOL.equal[T] for all types T as the Scala equality ==
```
Analyse et Conception Formelles

Lesson 6

Certified Programming

Outline

- Certified program production lines
  - Some examples of certified code production lines
  - What are the weak links?
  - How to certify a compiler?
  - How to certify a static analyzer of code?
  - How to guarantee the correctness of proofs?

- Methodology for formally defining programs and properties
  - Simple programs have simple proofs
  - Generalize properties when possible
  - Look for the smallest trusted base

B code production line

- The first certified code production line used in the industry
- For security critical code
- Used for onboard automatic train control of metro 14 (RATP)
- Several industrial users: RATP, Alstom, Siemens, Gemalto

Scade/Astree/CompCert code production line

- The (next) Airbus code production line
- For security critical code (e.g., flight control)
- Scade uses model-checking to verify programs or find counterexamples
- Astree is a static analyzer of C programs proving the absence of
  - division by zero, out of bound array indexing
  - arithmetic overflows
- Frama-C is a proof tool for C programs based on Why, automated provers like Alt-Ergo, CVC4, Z3, etc. and the Coq proof assistant
- CompCert is a certified C compiler (X. Leroy & S. Blazy, etc.)
Isabelle to Scala line

- Used for specification and verification of industrial size softwares e.g. Operating system kernel seL4 (C code)
- Code generation not yet used at an industrial level
- More general purpose line than previous ones
- All proofs performed in Isabelle are checked by a trusted kernel e.g. some research efforts for certifying a JVM

What are the weak links of such lines?

- The initial choice of algorithms and properties
- The verification tools (analyzers and proof assistants)
- Code generators/compilers

⇒ we need some guaranties on each link!
- Certification of compilers
- Certification of static analyzers
- Verification of proofs in proof assistant
- Methodology for formally defining algorithms and properties

⇒ we need to limit the trusted base!

How to limit the trusted base?

1. Compiler checks
2. Static Analyzer checks
3. Proven Theory checks

The trusted base

1. Compiler
2. Static Analyzer
3. Proven Theory
How to certify a static analyzer (SAn)? (II)

Isabelle file cm6.thy

Exercise 1

Define a static analyzer san for such programs:

\[ \text{san:: program} \Rightarrow \text{bool} \]

Exercise 2

Define the BAD predicate on program states:

\[ \text{BAD:: pgState} \Rightarrow \text{bool} \]

Exercise 3

Define the correctness lemma for the static analyzer san.
In the end, we managed to do this...

![Diagram showing the trusted base of a proof assistant]

How to guarantee correctness of proofs in proof assistants?

How to be convinced by the proofs done by a proof assistant?
- Relies on complex algorithms
- Relies on complex logic theories
- Relies on complex decision procedures

⇒ there may be bugs everywhere!

Weak points of proof assistants

A proof in a proof assistant is a tree whose leaves are axioms

Difference with a proof on paper:
- Far more detailed
- A lot of axioms
- Shortcuts: External decision procedures

Axioms ⇒ fewer details
Decision Proc. ⇒ automatization

Axioms and decision procedures are the main weaknesses of proof assistants

Proof handling: differences between proof assistants

<table>
<thead>
<tr>
<th></th>
<th>Coq</th>
<th>PVS</th>
<th>Isabelle</th>
<th>ACL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axioms</td>
<td>minimum and fixed</td>
<td>free</td>
<td>minimum and fixed</td>
<td>free</td>
</tr>
<tr>
<td>Decision procedures</td>
<td>proofs checked by Coq</td>
<td>trusted (no check)</td>
<td>proofs checked by Isabelle</td>
<td>trusted (no check)</td>
</tr>
<tr>
<td>Proof terms</td>
<td>built-in</td>
<td>no</td>
<td>additional</td>
<td>no</td>
</tr>
<tr>
<td>System automation</td>
<td>basic</td>
<td>in between</td>
<td>in between</td>
<td>good</td>
</tr>
<tr>
<td>Counterexample generator</td>
<td>none</td>
<td>basic</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
Proof checking: how is it done in Isabelle/HOL?

Isabelle/HOL have a well defined and «small » trusted base
- A kernel deduction engine (with Higher-order rewriting)
- Few axioms for each theory (see HOL.thy, HOL/Nat.thy)
- Other properties are lemmas, i.e. demonstrated using the axioms

All proofs are carried out using this trusted base:
- Proofs directly done in Isabelle (auto/simp/induct/...)
- All proofs done outside (sledgehammer) are re-interpreted in Isabelle using metis or smt that construct an Isabelle proof

Example 1
Prove the lemma \((x + 4) \ast (y + 5) \geq x \ast y\) using sledgehammer.
1. Interpret the found proof using metis
2. Switch on tracing: add
   - using [[simp_trace=true,simp_trace_depth_limit=5]] before the apply command
3. Re-interpret the proof

Simple programs have simple proofs: Simple is beautiful

Example 2 (The intersection function of TP2/3)
An «optimized» version of \(\text{intersection}\) is harder to prove.
1. Program function \(f(x)\) as simply as possible... no optimization yet!
   - Use simple data structures for \(x\) and the result of \(f(x)\)
   - Use simple computation methods in \(f\)
2. Prove all the properties \(\text{lem1}, \text{lem2}, \ldots\) needed on \(f\)
3. (If necessary) program \(\text{fopt}(x)\) an optimized version of \(f\)
   - Optimize computation of \(\text{fopt}\)
   - Use optimized data structure if necessary
4. Prove that \(\forall x. f(x)=\text{fopt}(x)\)
5. Using the previous lemma, prove again \(\text{lem1}, \text{lem2}, \ldots\) on \(\text{fopt}\)

Outline

- Certified program production lines
  - Some examples of certified code production lines
  - What are the weak links?
  - How to certify a compiler?
  - How to certify a static analyzer of code?
  - How to guarantee the correctness of proofs?

- Methodology for formally defining programs and properties
  - Simple programs have simple proofs
  - Generalize properties when possible
  - Look for the smallest trusted base

Simple programs have simple proofs (II)

Exercise 4
The function \(\text{fastReverse}\) is a tail-recursive version of \(\text{reverse}\). Prove the classical lemmas on \(\text{fastReverse}\) using the same properties of \(\text{reverse}\):
- \(\text{fastReverse} (\text{fastReverse} l)=l\)
- \(\text{fastReverse} (l1@l2)=(\text{fastReverse} l2)@(\text{fastReverse} l1)\)

Exercise 5
Prove that the fast exponentiation function \(\text{fastPower}\) enjoys the classical properties of exponentiation:
- \(x^y \ast x^z = x^{y+z}\)
- \((x \ast y)^z = x^z \ast y^z\)
- \(x^{y+z} = x(y+z)\)
Generalize properties when possible

Exercise 6

Recall functions member and intersection of TP2/3.
- Prove that
  $((\text{member } e \ l1) \land (\text{member } e \ l2)) \rightarrow (\text{member } e \ (\text{intersection} \ l1 \ l2))$
- How to generalize this property?
- What is the problem with the given function intersection?

Exercise 7

Recall function clean of TP2/3.
- Prove that clean $[x,y,x]=[y,x]$
- How to generalize this property of clean?
- What is the problem with the given definition of function clean?

Limit the trusted base in your Isabelle theories

Trusted base= functions that you cannot prove and have to trust
- Basic functions on which lemmas are difficult to state
- Interdependent functions... choose the simplest one as a trusted base

Example 3 (Prove a parser and a prettyPrinter on programs)

- parser:: string $\Rightarrow$ prog
- prettyPrinter:: prog $\Rightarrow$ string

The property to prove is: $\forall \ p. \ parser(\text{prettyPrinter} \ p) = p$
prettyPrinter is more likely to be the trusted base since it is simpler

- To verify a function $f$, define lemmas using $f$ and:
  - functions of the trusted base
  - other proven functions

Example 4

In TP2/3, which functions can be a good trusted base?
Outline

- Testing
- Model-checking
- Assisted proof
- Static Analysis
- A word about prototypes/models, accuracy, code generation

Disclaimer

Theorem 1 (Rice, 1953)
Any nontrivial property about the language recognized by a Turing machine is undecidable.

"The more you prove the less automation you have"

The basics

Definition 2 (Specification)
A complete description of the behavior of a software.

Definition 3 (Oracle)
An oracle is a mechanism determining whether a test has passed or failed, w.r.t. a specification.

Definition 4 (Domain (of Definition))
The set of all possible inputs of a program, as defined by the specification.
**Notations**

- **Spec** the specification
- **Mod** a formal model or formal prototype of the software
- **Source** the source code of the software
- **EXE** the binary executable code of the software
- **D** the domain of definition of the software
- **Oracle** an oracle
  - **D#** an abstract definition domain
- **Source#** an abstract source code
- **Oracle#** an abstract oracle

**Testing principles (random generators)**

This is what Isabelle/HOL quickcheck does.

**Definition 5 (Code coverage)**

The degree to which the source code of a program has been tested, e.g. a statement coverage of 70% means that 70% of all the statements of the software have been tested at least once.
Demo of white box testing in Evosuite

Objective: cover 100% of code (and raised exceptions)

Example 6 (Program to test with Evosuite)
```java
public static int Puzzle(int[] v, int i){
    if (v[i]>1) {
        if (v[i+2]==v[i]+v[i+1]) {
            if (v[i+3]==v[i]+18)
                throw new Error("hidden bug!");
            else return 1;
        }
    }
    else return 3;
}
```

Testing, to sum up

Strong and weak points

+ Done on the code —— Finds real bugs!
+ Simple tests are easy to guess
  - Good tests are not so easy to guess! (Recall TP0?)
+ Random and white box testing automate this task. May need an oracle: a formula or a reference implementation.
  - Finds bugs but cannot prove a property
+ Test coverage provides (at least) a metric on software quality

Some tool names

SAGE (Microsoft), PathCrawler (CEA), Evosuite, many others ...

One killer result

SAGE (running on 200 PCs/year) found 1/3 of security bugs in Windows 7

Demo of white box testing in Evosuite

Generates tests for all branches (1, 2, 3, null array, hidden bug, etc)

One of the generated JUnit test cases:
```java
@Test(timeout = 4000)
public void test5() throws Throwable {
    int[] intArray0 = new int[18];
    intArray0[1] = 3;
    intArray0[3] = 3;
    intArray0[4] = 21; // an array raising hidden bug!
    try {
        Main.Puzzle(intArray0, 1);
        fail("Expecting exception: Error");
    } catch(Error e) {
        verifyException("temp.Main", e);
    }
}
```

Model-checking principles

Where \( \models \) is the usual logical consequence. This property is not shown by doing a logical proof but by checking (by computation) that ...
Model-checking principles (II)

Model-checking principle explained in Isabelle/HOL

Exercise 1
Define the lemma stating that whatever the initial state, typing A,B,C leads execution to Final state.

Exercise 2
Define the lemma stating that the only possibility for arriving in Final state is to have typed a sequence of letters ending by A,B,C.

Assisted proof principles

Some tool names
SPIN, SMV, (bug finders) CBMC, SLAM, ESC-Java, Java path finder, . . .

One killer result
INTEL processors are commonly model-checked

Where D, Mod and Oracle are finite.
Assisted proof, to sum-up

**Strong and weak points**

- Can do the proof or find bugs (with counterexample finders)
- Proofs can be **certified**
- Needs assistance
- For models/prototypes only (not on source nor on EXE)
- Proof holds on the source code if it is generated from the prototype

**Some tool names**

B, Coq, Isabelle/HOL, ACL2, PVS, ... Why, Frama-C, ...

**One killer result**

CompCert certified C compiler

---

**Static Analysis principles**

Where abstraction $\cdots$ is a **correct** abstraction

---

**Static Analysis principles (II)**

Where abstraction $\cdots$ is a **correct** abstraction
Static Analysis principles – Abstract Interpretation (III)

The abstraction ‘~’ is based on the abstraction function \( \text{abs} :: D \Rightarrow D^\# \)

Depending on the verification objective, precision of \( \text{abs} \) can be adapted

Example 7 (Some abstractions of program variables for \( D=\text{int} \))

1. \( \text{abs} :: \text{int} \Rightarrow \{\bot, T\} \) where \( \bot \equiv \text{“undefined”} \) and \( T \equiv \text{“any int”} \)
2. \( \text{abs} :: \text{int} \Rightarrow \{\bot, \text{Neg}, \text{Pos}, \text{Zero}, \text{NegOrZero}, \text{PosOrZero}, T\} \)
3. \( \text{abs} :: \text{int} \Rightarrow \{\bot\} \cup \text{Intervals on } \mathbb{Z} \)

Example 8 (Program abstraction with \( \text{abs}\) (1), (2) and (3))

\[
\begin{array}{c|c|c|c}
\text{x:= y+1; } & x=\bot & x=\bot & x=\bot \\
\text{read(x); } & x=T & x=T & x=\{+\infty;+\infty\} \\
\text{y:= x+10 } & y=T & y=T & y=\{-\infty;+\infty\} \\
\text{u:= 15; } & u=T & u=\text{Pos} & u=[15;15] \\
\text{x:= |x| } & x=T & x=\text{PosOrZero} & x=\{0;+\infty\} \\
\text{u:= x+u; } & u=T & u=\text{Pos} & u=[15;+\infty] \\
\end{array}
\]

Static Analysis principle explained in Isabelle/HOL

To abstract \( \text{int} \), we define \( \text{absInt} \) as the abstract domain (\( D^\# \)):

```
datatype absInt= Neg|Zero|Pos|Undef|Any
```

Exercise 3

Define the function \( \text{abs} :: \text{int} \Rightarrow \text{absInt} \)

Exercise 4

Define the function \( \text{absPlus} :: \text{absInt} \Rightarrow \text{absInt} \Rightarrow \text{absInt} \) (noted \( +\# \))

Exercise 5 (Prove that \(+\#\) is a correct abstraction of \(+\))

Prove that \( \forall x, y \in \mathbb{Z} : \text{abs}(x) +\# \text{abs}(y) \) is a correct abstraction of \( x + y \).
To sum-up on all presented techniques

- Some properties are too complex to be verified using a static analyzer
- Testing can only be used to check finite properties
- Model-checking deals only with finite models (to be built by hand)
- Static analysis is always fully automatic

Testing works on EXE, Static analysis on source code, others on models/prototypes
- Model-checking, assisted proof and static analysis have a similar guarantee level except that assisted proofs can be certified

A word about models/prototypes

Program verification using “formal methods” relies on:

- Program has a Property
- Abstraction
- Prototype is equivalent to Logic Formula

This is the case for model-checking and assisted proof.

Testing prototypes is a common practice in engineering

It is crucial for early detection of problems! Do you know Tacoma bridge?
Testing prototypes is an engineering common practice (II)
More and more, prototypes are mathematical/numerical models

If the prototype is accurate: any detected problem is a **real** problem!

Problem on the prototype $\rightarrow$ Problem on the real system

But in general, we do not have the opposite:

No problem on the prototype $\not\rightarrow$ No problem on the real system

---

Why code exportation is a great plus?
Code exportation produces the program from the model itself!

Thus, we here have a **great bonus**:

[TP5bis, TP67, CompCert]

No problem on the prototype $\rightarrow$ No problem on the real system

If the exported program is not efficient enough it can, at least, be used as a reference implementation (an oracle) for testing the optimized one.

---

About "Property $\rightarrow$ Logic formula"
This is the only remaining difficulty, and this step is **necessary**!

Back to TP0, it is very difficult for **two reasons**:

1. The "what to do" is not as simple as it seems
   - Many tests to write and what exactly to test?
   - How to be sure that no test was missing?
   - Lack of a **concise and precise** way to state the property
     Defining the property with a french text is too ambiguous!

2. The "how to do" was not that easy

Logic Formula = factorization of tests
- guessing 1 formula is harder than guessing 1 test
- guessing 1 formula is harder than guessing 10 tests
- guessing 1 formula is **not harder** than guessing 100 tests
- guessing 1 formula is **faster** than writing 100 tests (TP0 in Isabelle)
- proving 1 formula is **stronger** than writing **infinitely** many tests

---

About formal methods and security

You **have to use formal methods** to secure your software
... because hackers will use them to find new attacks!

Be serious, do hackers read scientific papers?
or use academic stuff?

**Yes, they do!**
compliant smart cards in circulation worldwide [5]. In EMV,
Europay, MasterCard, and Visa), has been deployed through-
for point-of-sale and ATM transactions in many countries.

The standard also defines exchanges between cards and automatic
teller machines (ATMs). The protocol vulnerability described in [7]
is based on the fact that the card does not condition
transaction request cryptogram (ARQC) based on
an amount, currency, date, terminal ID, fresh randomness, etc.). The card replies with an authoriza-
tion request cryptogram (ARQC). The issuer receives this ARQC,
performs a signature calculation, and returns the response cryptogram (RC).
The transaction is then completed by the terminal.

During cardholder verification, the PoS queries the PIN from the user and transmits it to the
issuing bank. The bank verifies the PIN using its own database and
returns a response to the PoS. If the PIN is valid, the transaction is
authorized; otherwise, it is rejected.

The forensic analysis relied on X-ray chip imaging, side-channel analysis, protocol analysis, and
traffic analysis to identify the weaknesses in the EMV protocol.

The protocol failure described in [7] is based on the fact that the card does not
condition the transaction request cryptogram (ARQC) based on
an amount, currency, date, terminal ID, fresh randomness, etc.). The card replies with an authoriza-
tion request cryptogram (ARQC). The issuer receives this ARQC,
performs a signature calculation, and returns the response cryptogram (RC).
The transaction is then completed by the terminal.

Criminals used the attack of Murdoch & al. but not:
About formal methods and security

You have to use formal methods to secure your software
... because hackers will use them to find new attacks!

(1 formula) + (counter-example generator) $\longrightarrow$ attack!

- Fuzzing of implementations using model-checking
- Finding bugs (to exploit) using white-box testing
- Finding errors in protocols using counter-example gen. (e.g. TP89)

$\Longrightarrow$ You will have to formally prove security of your software!