CM ACF - Table of contents

Analyse et Conception Formelles

Lesson 1

–

Propositional logic
First order logic

Bibliography

- *Cours de logique pour l'informatique*, J-F. Raskin, `http://www.ulb.ac.be/di/ssd/jfr/info-148.html`
- *Logique et fondements de l'informatique* de Richard Lassaigne et Michel de Rougemont. Hermes 1993.

A selected bibliography on the Isabelle/HOL prover

- `http://people.irisa.fr/Thomas.Genet/ACF/BiblioIsabelle`

The web page of the course

- `http://people.irisa.fr/Thomas.Genet/ACF`

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

- `http://people.irisa.fr/Thomas.Genet/ACFSol`

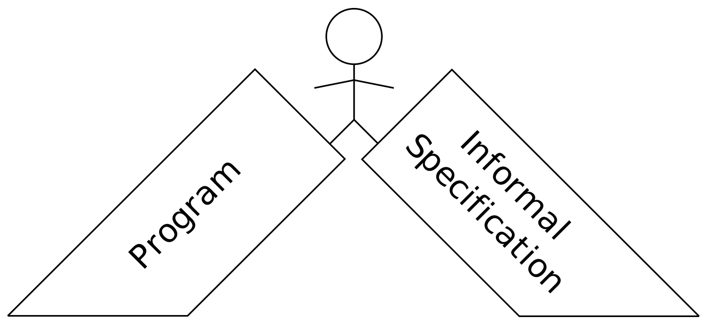
Acknowledgements

- Many thanks to T. Nipkow, J. Blanchette, L. Bulwahn and G. Riou for providing material, answering questions and for fruitful discussions.

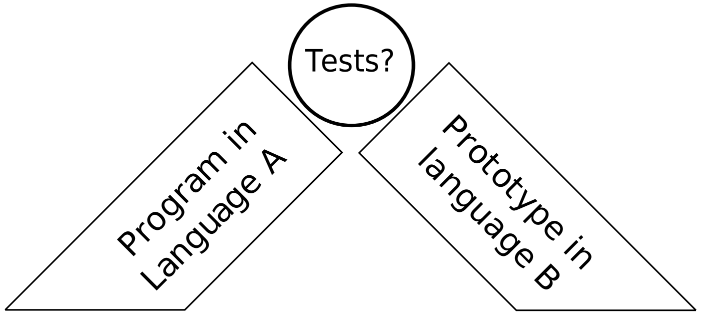
Outline

- 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

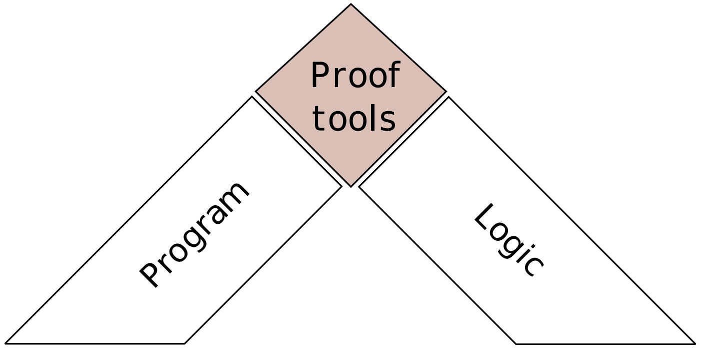
Why using logic for specifying/verifying programs?



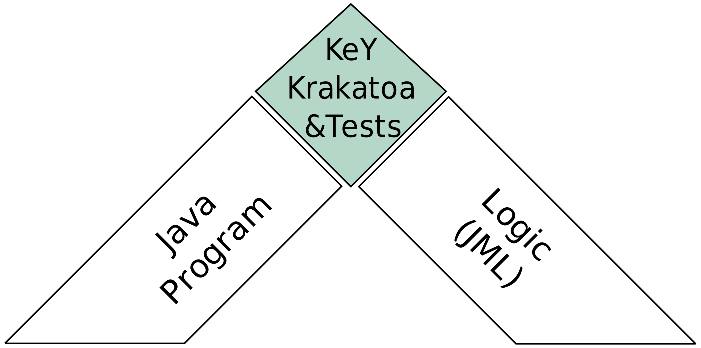
Why using logic for specifying/verifying programs?



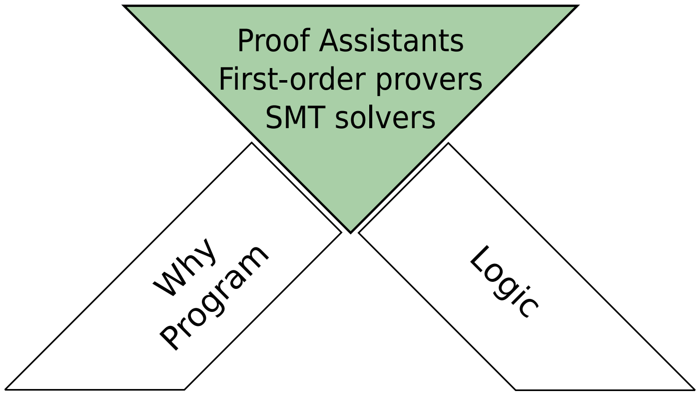
Why using logic for specifying/verifying programs?



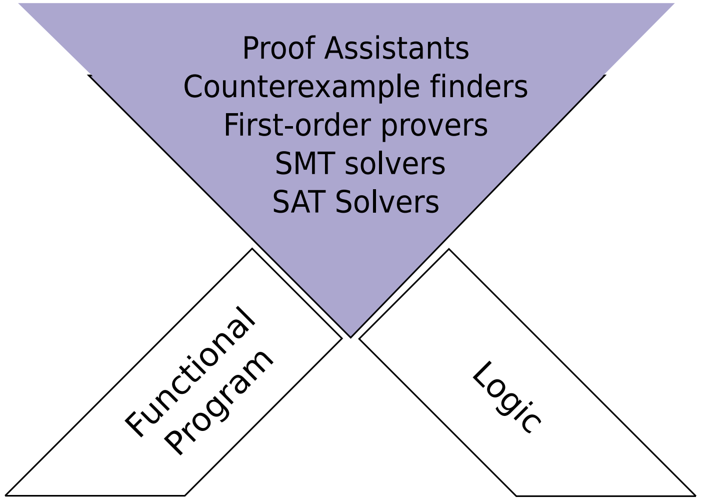
Why using functional paradigm to program?



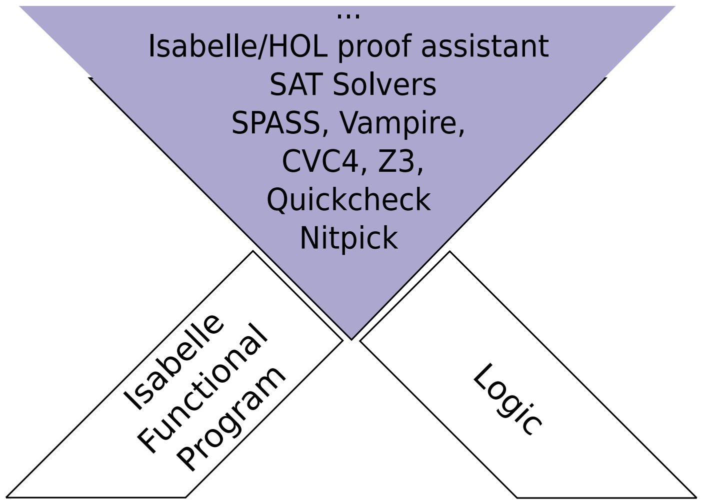
Why using functional paradigm to program?



Why using functional paradigm to program?



Why using functional paradigm to program?



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

$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \longrightarrow \phi_2$ where $p \in P$

Definition 2 (Propositional interpretation)

An *interpretation* I associates to variables of P a value in $\{\texttt{True}, \texttt{False}\}$.

Example 3

Let $\phi = (p_1 \wedge p_2) \longrightarrow p_3$. Let I be the interpretation such that $I[\![p_1]\!] = \texttt{True}$, $I[\![p_2]\!] = \texttt{True}$ and $I[\![p_3]\!] = \texttt{False}$.

Propositional logic: syntax and interpretations (II)

We extend the domain of I to formulas as follows:

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

$I[\![\phi_1 \vee \phi_2]\!] = \texttt{True}$ iff $I[\![\phi_1]\!] = \texttt{True}$ or $I[\![\phi_2]\!] = \texttt{True}$

$I[\![\phi_1 \wedge \phi_2]\!] = \texttt{True}$ iff $I[\![\phi_1]\!] = \texttt{True}$ and $I[\![\phi_2]\!] = \texttt{True}$

$$I[\![\phi_1 \longrightarrow \phi_2]\!] = \texttt{True iff } \begin{cases} I[\![\phi_1]\!] = \texttt{False or} \\ I[\![\phi_1]\!] = \texttt{True and } I[\![\phi_2]\!] = \texttt{True} \end{cases}$$

Example 4

Let $\phi = (p_1 \wedge p_2) \longrightarrow p_3$ and I the interpretation such that $I[\![p_1]\!] = \texttt{True}$, $I[\![p_2]\!] = \texttt{True}$ and $I[\![p_3]\!] = \texttt{False}$.

We have $I[\![p_1 \wedge p_2]\!] = \texttt{True}$ and $I[\![(p_1 \wedge p_2) \longrightarrow p_3]\!] = \texttt{False}$.

Propositional logic: syntax and interpretations (III)

The presentation using truth tables is generally preferred:

a	$\neg a$
False	True
True	False

a	b	$a \vee b$
False	False	False
True	False	True
False	True	True
True	True	True

a	b	$a \wedge b$
False	False	False
True	False	False
False	True	False
True	True	True

a	b	$a \longrightarrow b$
False	False	True
True	False	False
False	True	True
True	True	True

Propositional logic: models

Definition 5 (Propositional model)

I is a *model* of ϕ, denoted by $I \models \phi$, if $I[\![\phi]\!] =$ True.

Definition 6 (Valid formula/Tautology)

A formula ϕ is *valid*, denoted by $\models \phi$, if for all I we have $I \models \phi$.

Example 7

Let $\phi = (p_1 \wedge p_2) \longrightarrow p_3$ and $\phi' = (p_1 \wedge p_2) \longrightarrow p_1$. Let I be the interpretation such that $I[\![p_1]\!] =$ True, $I[\![p_2]\!] =$ True and $I[\![p_3]\!] =$ False. We have $I \not\models \phi$, $I \models \phi'$, and $\models \phi'$.

Propositional logic: decidability and tools in Isabelle/HOL

Property 1

In propositional logic, given ϕ, the following problems are decidable:

- *Is $\models \phi$?*
- *Is there an interpretation I such that $I \models \phi$?*
- *Is there an interpretation I such that $I \not\models \phi$?*

- To automatically prove that $\models \phi$ .... `apply auto` (if the formula is not valid, there remains some unsolved goals)
- To build I such that $I \not\models \phi$ (or $I \models \neg\phi$) .... `nitpick` (i.e. find a counterexample... may take some time on large formula)

Other useful commands

- To close the proof of a proven formula........................ `done`
- To abandon the proof of an unprovable formula .... `oops`
- To abandon the proof of (potentially) provable formula .... `sorry`

Writing and proving propositional formulas in Isabelle/HOL

Example 8 (Valid formula)

```
lemma "(p1 /\ p2) --> p1"
apply auto
done
```

Example 9 (Unprovable formula)

```
lemma "(p1 /\ p2) --> p3"
nitpick
oops
```

Isabelle/HOL: ASCII notations

Symbol	ASCII notation
`True`	`True`
`False`	`False`
$\wedge$	`/\`
$\vee$	`\/`
$\neg$	`~`
$\neq$	`~=`
$\longrightarrow$	`-->`
$\longleftrightarrow$	`=`
$\forall$	`ALL`
$\exists$	`?`
λ	`%`

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. $A \vee B$
2. $(((A \wedge B) \longrightarrow \neg C) \vee (A \longrightarrow B)) \longrightarrow A \longrightarrow C$
3. *If it rains, Robert takes his umbrella. Robert does not have his umbrella hence it does not rain.*
4. $(A \longrightarrow B) \longleftrightarrow (\neg A \vee B)$

First-order logic (FOL) / Predicate logic

1. Terms and Formulas
2. Interpretations
3. Models
4. Logic consequence and verification

First-order logic: terms

Definition 10 (Terms)

Let $\mathcal{F}$ be a set of symbols and *ar* a function such that $ar : \mathcal{F} \Rightarrow \mathbb{N}$ 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:
$\mathcal{T}(\mathcal{F},\mathcal{X}) = \mathcal{X} \cup \{f(t_1,\ldots,t_n) \mid ar(f) = n \text{ and } t_1,\ldots,t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})\}$

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:

$$\phi ::= \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \longrightarrow \phi_2 \mid \forall x.\phi \mid \exists x.\phi \mid p(t_1, \ldots, t_n)$$

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:

- $p(f(a))$
- $q(g(f(a), x), y)$
- $\forall x.\exists y.y \leq x$
- $\forall x.\forall y.\forall z.x \leq y \wedge y \leq z \longrightarrow x \leq z$

First-order logic syntax: the quiz

Quiz 1

Let $P = \{p : 1, q : 2, \leq : 2\}$, $\mathcal{F} = \{f : 1, g : 2, a : 0\}$ and $\mathcal{X} = \{x, y, z\}$.

- $f(g(a))$ *is a term* | V | True || R | False
- a *is a term* | V | True || R | False
- x *is a term* | V | True || R | False
- $\forall x.\ x$ *is a term* | V | True || R | False
- $\forall x.\ x$ *is a formula* | V | True || R | False
- $p(f(g(a, x)))$ *is a formula* | V | True || R | False
- $\forall x.\ p(x) \wedge x \leq y$ *is a formula* | V | True || R | False

Interlude: a touch of lambda-calculus

We need to define *anonymous* functions

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

Interlude: a touch of lambda-calculus (in Isabelle HOL)

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
- $(\lambda x.\_)(a := b)$ 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 {`True`, `False`}

  For instance the predicate p on $\{a, b\}$
  $p = (\lambda x.\_)(a := False, b := False)$

First-order formulas: interpretations and valuations

Definition 14 (First-order interpretation)

Let ϕ be a formula and D a domain. An *interpretation* I of ϕ 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 $p_I : D^n \Rightarrow \{\texttt{True}, \texttt{False}\}$ to each predicate symbol $p \in P$ such that $ar(p) = n$.

Example 15 (Some interpretations of $\phi = \forall x. ev(x) \longrightarrow od(s(x))$)

- Let I be the interpretation such that domain $D = \mathbb{N}$ and $s_I \equiv \lambda x.x + 1 \quad ev_I \equiv \lambda x.((x \bmod 2) = 0) \quad od_I \equiv \lambda x.((x \bmod 2) = 1)$
- Let I' be the interpretation such that domain $D = \{a, b\}$ and $s_{I'} \equiv \lambda x.if\ x = a\ then\ b\ else\ a \quad ev_{I'} \equiv \lambda x.(x = a) \quad od_{I'} \equiv \lambda x.False$

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 ϕ for a valuation V is defined by:

- $(I, V)[\![x]\!] = V(x)$ if $x \in \mathcal{X}$
- $(I, V)[\![f(t_1, \dots, t_n)]\!] = f_I((I, V)[\![t_1]\!], \dots, (I, V)[\![t_n]\!])$ if $f \in \mathcal{F}$ and $ar(f) = n$
- $(I, V)[\![p(t_1, \dots, t_n)]\!] = p_I((I, V)[\![t_1]\!], \dots, (I, V)[\![t_n]\!])$ if $p \in P$ and $ar(p) = n$
- $(I, V)[\![\phi_1 \vee \phi_2]\!] = \texttt{True}$ iff $(I, V)[\![\phi_1]\!] = \texttt{True}$ or $(I, V)[\![\phi_2]\!] = \texttt{True}$
- 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 $x \neq y$.

First-order logic: satisfiability, models, tautologies

Definition 18 (Satisfiability)

I and V *satisfy* ϕ (denoted by $(I, V) \models \phi$) if $(I, V)[\![\phi]\!] = \texttt{True}$.

Definition 19 (First-order Model)

An interpretation I is a *model* of ϕ, denoted by $I \models \phi$, if for all valuation V we have $(I, V) \models \phi$.

Definition 20 (First-order Tautology)

A formula ϕ 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: decidability and tools in Isabelle/HOL

Property 2

In first-order logic, given ϕ, the following problems are undecidable:

- *Is $\models \phi$?*
- *Is there an interpretation I (and valuation V) such that $(I, V) \models \phi$?*
- *Is there an interpretation I (and valuation V) such that $(I, V) \not\models \phi$?*

- Try to automatically prove $\models \phi$ .... `apply auto`
  Uses decision procedures (*e.g.* arithmetic) to try to prove the formula.
  If it does not succeed, it does not mean that the formula is unprovable!
- Try to build I and V such that $(I, V) \not\models \phi$ .... `nitpick`
  If it does not succeed, it does not mean that there is no counterexample!

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.

1. $\forall x.\ p(x) \longrightarrow \exists x. p(x)$
2. $\exists x.\ p(x) \longrightarrow \forall x. p(x)$
3. $\forall x.\ ev(x) \longrightarrow od(s(x))$
4. $\forall x\ y.\ x > y \longrightarrow x + 1 > y + 1$
5. $x > y \longrightarrow x + 1 > y + 1$
6. $\forall m\ n.\ (\neg(m < n) \wedge m < n + 1) \longrightarrow m = n$
7. $\forall x.\ \exists y.\ x + y = 0$
8. $\forall y.\ (\neg p(f(y))) \longleftrightarrow p(f(y))$
9. $\forall y.\ (p(f(y)) \longrightarrow p(f(y + 1)))$

Isabelle/HOL notations: priority, associativity, shorthands

- Here are the logical operators in decreasing order of priority:
  - $=, \neg, \wedge, \vee, \longrightarrow, \forall, \exists$
  - «a prioritary operator first chooses its operands»
- For instance
  - $\neg\neg P = P$ means $\neg\neg(P = P)$!
  - $A \wedge B = B \wedge A$ means $A \wedge (B = B) \wedge A$!
  - $P \wedge \forall x. Q(x)$ will be parsed as $(P \wedge \forall) x. Q(x)$!

    Hence, write $P \wedge (\forall x. Q(x))$ instead!
- All binary operators are associative to the right, for instance $A \longrightarrow B \longrightarrow C$ is equivalent to $A \longrightarrow (B \longrightarrow C)$
- Nested quantifications $\forall x. \forall y. \forall z.\ P$ can be abbreviated into $\forall x\ y\ z.\ P$
- Free variables are universally quantified, i.e. $P(x)$ is equiv. to $\forall x.\ P(x)$

  All Isabelle/HOL tools will prefer $P(x)$ to $\forall x.\ P(x)$

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 $(I, V) \models \phi$.

Example 22

Let $\phi = p(f(y))$ with $\mathcal{F} = \{f : 1\}$, $P = \{p : 1\}$, $\mathcal{X} = \{y\}$.
The formula ϕ is satisfiable (there exists (I, V) such that $(I, V) \models \phi$)

- Let I be the interp. s.t. $D = \{0, 1\}$, $p_I \equiv \lambda x.(x = 0)$, $f_I = \lambda x.x$
- Let V be the valuation such that $V(y) = 0$

We have $(I, V) \models \phi$. With $V'(y) = 1$, $(I, V') \not\models \phi$. Hence, I is not a model of ϕ.

- Let I' be the interp. s.t. $D = \{0, 1\}$, $p_{I'} \equiv \lambda x.(x = 0)$, $f_{I'} = \lambda x.0$

We have $(I', V) \models \phi$ for all valuation V, hence I' is a model of ϕ.

Satisfiability – the quiz

Quiz 2

Let $P = \{p : 1\}$, $\mathcal{F} = \{f : 1, a : 0, b : 0\}$ and $\mathcal{X} = \{x\}$.

- $f(a)$ *is satisfiable* — V True ‖ R False
- $p(f(a))$ *is satisfiable* — V True ‖ R False
- $p(f(x))$ *is satisfiable* — V True ‖ R False
- $p(f(x))$ *is a tautology* — V True ‖ R False
- $\neg p(f(x))$ *is satisfiable* — V True ‖ R False
- $\neg p(f(x)) \wedge p(f(x))$ *is satisfiable* — V True ‖ R False
- $p(f(a)) \longrightarrow p(f(b))$ *is satisfiable* — V True ‖ R False

First-order logic: contradictions

Definition 23 (Contradiction)

A formula is *contradictory* (or *unsatisfiable*) if it cannot be satisfied, i.e. $(I, V) \not\models \phi$ for all interpretation I and all valuation V.

Property 3

A formula ϕ is contradictory iff $\neg\phi$ is a tautology.

Example 24 (See in Isabelle `cm1.thy` file)

Let $\phi = (\forall y.\, \neg p(f(y))) \longleftrightarrow (\forall y.\, p(f(y)))$. The formula ϕ is contradictory and $\neg\phi$ is a tautology.

Analyse et Conception Formelles

Lesson 2

–

Types, terms and functions

Outline

1. Terms
   - Types
   - Typed terms
   - λ-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

Types: syntax

τ	::=	(τ)	
	|	*bool* | *nat* | *char* | ...	base types
	|	$'a$ | $'b$ | ...	type variables
	|	$\tau \Rightarrow \tau$	functions
	|	$\tau \times \ldots \times \tau$	tuples (ascii for $\times$: $*$)
	|	τ *list*	lists
	|	...	user-defined types

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

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

Typed terms: syntax

term	::=	(*term*)	
	|	*a*	$a \in \mathcal{F}$ or $a \in \mathcal{X}$
	|	*term term*	function application
	|	$\lambda y.$ *term*	function definition with $y \in \mathcal{X}$
	|	(*term*, ..., *term*)	tuples
	|	[*term*, ..., *term*]	lists
	|	(*term* :: τ)	type annotation
	|	...	a lot of syntactic sugar

Function application is left-associative, for instance:

$$f\ a\ b\ c \text{ is equivalent to } ((f\ a)\ b)\ c$$

Example 1 (Types of terms)

Term	Type	Term	Type
y	'a	t1	'a
(t1,t2,t3)	('a × 'b × 'c)	[t1,t2,t3]	'a list
λ y. y	'a ⇒ 'a	λ y z. z	'a ⇒ 'b ⇒ 'b

Types and terms: evaluation in Isabelle/HOL

To evaluate a term t in Isabelle .... value "t"

Example 2

Term	Isabelle's answer
value "True"	True::bool
value "2"	Error (cannot infer result type)
value "(2::nat)"	2::nat
value "[True,False]"	[True,False]::bool list
value "(True,True,False)"	(True,True,False)::bool * bool * bool
value "[2,6,10]"	Error (cannot infer result type)
value "[(2::nat),6,10]"	[2,6,10]::nat list

Terms and functions: semantics is the λ-calculus

Semantics of functional programming languages consists of one rule:

$$(\lambda\, x.\, t)\; a \;\twoheadrightarrow_\beta\; t\{x \mapsto a\} \qquad (\beta\text{-reduction})$$

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

Example 3

- $(\lambda\, x.\, x+1)\; 10 \;\twoheadrightarrow_\beta\; 10+1$
- $(\lambda\, x. \lambda\, y.\, x+y)\; 1\; 2 \;\twoheadrightarrow_\beta\; (\lambda\, y.\, 1+y)\; 2 \;\twoheadrightarrow_\beta\; 1+2$
- $(\lambda\, (x,y).\, y)\; (1,2) \;\twoheadrightarrow_\beta\; 2$

In Isabelle/HOL, to be β-reduced, terms have to be well-typed

Example 4

Previous examples can be reduced because:

- $(\lambda\, x.\, x+1) :: nat \Rightarrow nat$ and $10 :: nat$
- $(\lambda\, x. \lambda\, y.\, x+y) :: nat \Rightarrow nat \Rightarrow nat$ and $1 :: nat$ and $2 :: nat$
- $(\lambda\, (x,y).y) :: ('a \times 'b) \Rightarrow 'b$ and $(1,2) :: nat \times nat$

Lambda-calculus – the quiz

Quiz 1

- *Function $\lambda(x,y).\, x$ is a function with two parameters*

V	True	R	False

- *Type of function $\lambda(x,y).\, x$ is*

V	'a × 'b ⇒ 'a
R	'a ⇒ 'b ⇒ 'a

- *If* `f::nat ⇒ nat ⇒ nat` *how to call* `f` *on* `1` *and* `2`*?*

V	`f(1,2)`	R	`(f 1 2)`

- *If* `f::nat × nat ⇒ nat` *how to call* `f` *on* `1` *and* `2`*?*

V	`f(1,2)`	R	`(f 1 2)`

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 $(\lambda\, x. \lambda\, y.\, x+y) :: nat \Rightarrow nat \Rightarrow nat$ is curried
The function $(\lambda\, (x,y).\, x+y) :: nat \times nat \Rightarrow nat$ 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!

- $(\lambda\, x. \lambda\, y.\, x+y)\; 1\; 2 \;\twoheadrightarrow_\beta\; (\lambda\, y.\, 1+y)\; 2 \;\twoheadrightarrow_\beta\; (1+2) :: nat$
- $(\lambda\, x. \lambda\, y.\, x+y)\; 1 \;\twoheadrightarrow_\beta\; (\lambda\, y.\, 1+y) :: nat \Rightarrow nat$ which is a function!

Exercise 1 (In Isabelle/HOL)

Use `append::'a list ⇒ 'a list ⇒ 'a list` *to concatenate 2 lists of bool, 2 lists of nat, and 3 lists of nat.*

A word about curried functions and partial application (II)

- To associate the value of a term `t` to a name `n` ...... definition "n=t"

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 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)

- $\lambda x.\lambda f.\, f\ x :: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$
- $\lambda f.\lambda x.\, f\ x :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
- $\lambda f.\lambda x.\, f\ (x+1)\ (x+1) :: (nat \Rightarrow nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$

$(\lambda f.\lambda x.\, f\ (x+1)\ (x+1))$ `add` 20
$\twoheadrightarrow_\beta (\lambda x.$ `add` $(x+1)\ (x+1))\ 20$
$\twoheadrightarrow_\beta$ `add` $(20+1)\ (20+1)$
$= (\lambda x.\lambda y.\, x+y)\ (20+1)\ (20+1)$
$\twoheadrightarrow_\beta (20+1)+(20+1)$
$= 42$

A word about higher-order functions (II)

Exercise 3 (In Isabelle/HOL)

1. *Define a function* `triple` *which applies three times a given function to an argument*
2. *Using* `triple`*, apply three times the function* `incr` *on* 0
3. *Using* `triple`*, apply three times the function* `app1` *on* `[2,3]`
4. *Using* `map` $::$ *('a ⇒ 'b) ⇒ 'a list ⇒ 'b list from the list* $[1,2,3]$ *build the list* $[2,3,4]$

Interlude: a word about semantics and verification

- To verify programs, formal reasoning on their semantics is crucial!
- To prove a property ϕ 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 [a], subsets of C (hundreds of pages)
- Proofs are hard because of semantics complexity (*e.g.* KeY for Java)

[a] `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`

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

Constructor terms

Isabelle distinguishes between constructor and function symbols

- A function symbol is associated to a function, *e.g.* `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: $\ldots (0,2), (0,1), (0,0), (1,0), \ldots$

  where $(0,2) = (1,3) = (2,4) = \ldots$ all represent the *same* integer -2
- Lists are built using the operators
  - *Nil*: the empty list
  - *Cons*: the operator adding an element to the (head) of the list

    Be careful! the type of *Cons* is *Cons*`::'a ⇒ 'a list ⇒ 'a list`

The term *Cons* 0 (*Cons* (*Suc* 0) *Nil*) represents the list $[0,1]$

Constructor terms – the quiz

Quiz 2

- *Nil is a term?* — V True | R False
- *Nil is a constructor term?* — V True | R False
- *(Cons (Suc 0) Nil) is a constructor term?* — V True | R False
- *((Suc 0), Nil) is a constructor term?* — V True | R False
- *(inc (Suc 0)) is a constructor term?* — V True | R False
- *(Cons x Nil) is a constructor term?* — V True | R False
- *(inc x) is a constructor term?* — V True | R False

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*) $= 0\#(1\#[]) = [0,1]$ (similar to [] and :: of OCaml)
- 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* $[1,1,1]$ *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*

Outline

1. Terms
   - Types
   - Typed terms
   - λ-terms
   - Constructor terms
2. Functions defined using equations
   - Logic everywhere!
   - Function evaluation using term rewriting
   - Partial functions

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)

`fun` f :: "$\tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$"

`where`

" $f\ t_1^1 \ldots t_n^1 \quad = \quad r^1$ " |

...

" $f\ t_1^m \ldots t_n^m \quad = \quad r^m$ "

where for all $i = 1 \ldots n$ and $k = 1 \ldots m$ $(t_i^k :: \tau_i)$ are constructor terms **possibly with** variables, and $(r^k :: \tau)$

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

```
fun member:: "'a => 'a list => bool"
where
"member e []      = False" |
"member e (x#xs) = (if e=x then True else (member e xs))"
```

Function definition – the quiz

Quiz 3 (Is this function definition correct? V Yes | R No)

```
fun f:: "nat ⇒ nat ⇒ bool"
where
"f x y = (x + y)"
```

Quiz 4 (Is this function definition correct? V Yes | R No)

```
fun g:: "nat ⇒ nat ⇒ bool"
where
"g 0 y = False"
```

Quiz 5 (Is this function definition correct? V Yes | R No)

```
fun pos:: "nat ⇒ bool"
where
"pos 0 = False" |
"pos (Suc x) = True"
```


Function definition – the quiz (II)

Quiz 6 (Is this function definition correct? V Yes | R No)

```
fun pos2:: "nat ⇒ bool"
where
"pos2 0 = False" |
"pos2 (x + 1) = True"
```

Quiz 7 (Is this function definition correct? V Yes | R No)

```
fun isDivisor:: "nat ⇒
nat ⇒ bool"
where
"isDivisor x y = (∃ z. x * z = y)"
```


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)

fun f :: "$\tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$"
where
" $f\ t_1^1 \ldots t_n^1 \;=\; r^1$ " |
...
" $f\ t_1^m \ldots t_n^m \;=\; r^m$ "

f is *complete* if any call $f\ t_1 \ \ldots \ t_n$ with $(t_i :: \tau_i)$, $i = 1 \ldots n$ is covered by one 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.

Total and partial Isabelle/HOL functions (II)

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
   - λ-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.simps` of equations on terms.

To visualize a lemma/theorem/simplification rule .... `thm`
For instance: `thm "length_append"`, `thm "append.simps"`
To *find* the name of a lemma, etc. .... `find_theorems`
For instance: `find_theorems "append" "_ + _"`

Exercise 6

Use Isabelle/HOL to find the following formulas:

- *definition of* `member` *(we just defined) and of* `nth` *(part of List.thy)*
- *find the lemma relating* `rev` *(part of List.thy) and* `length`

Evaluating functions by rewriting terms using equations

The append function (aliased to `@`) is defined by the 2 equations:

```
(1) append    Nil  x = x                    (* recall that Nil=[] *)
(2) append (x#xs)  y = (x#(append xs y))
```

Replacement of equals by equals = Term rewriting

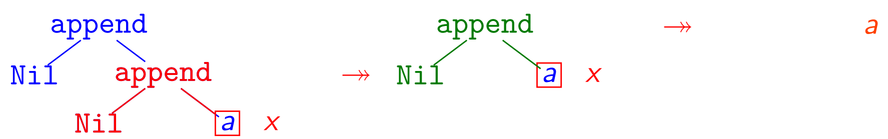
The first equation `(append Nil x) = x` means that

- *(concatenating the empty list with any list x)* is **equal** to *x*
- we can thus replace
  - any term of the form `(append Nil t)` by `t` (for any value `t`)
  - wherever and whenever we encounter such a term `append Nil t`

Term Rewriting in three slides

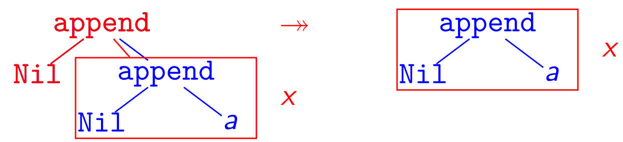
- Rewriting term `(append [] (append [] a))` using

```
(1) append    Nil   x  =  x
(2) append (x#xs)   y  =  (x#(append xs y))
```



- We note `(append Nil (append Nil a))` ↠ `(append Nil a)` if
  - there exists a position in the term where the rule matches
  - there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ for the rule to match.
    On the example $\sigma = \{x \mapsto a\}$
- We also have `(append Nil a)` ↠ `a`

and



Term Rewriting in three slides – Formal definitions

Definition 20 (Substitution)

A substitution σ 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 σ be the substitution $\sigma = \{x \mapsto g(a), y \mapsto h(z)\}$.

Let $t = f(h(x), x, g(y))$.

We have $\sigma(t) = f(h(g(a)), g(a), g(h(z)))$.

Term Rewriting in three slides – Formal definitions (II)

Definition 22 (Rewriting using an equation)

A term s can be *rewritten* into the term t (denoted by $s \twoheadrightarrow t$) using an Isabelle/HOL equation `l=r` if there exists a subterm u of s and a substitution σ such that $u = \sigma(\texttt{l})$. Then, t is the term s where subterm u has been replaced by $\sigma(\texttt{r})$.

Example 23

Let $s = f(g(a), c)$ and `g(x) = h(g(x),b)` the Isabelle/HOL equation.

we have	$f($ $g(a)$ $, c)$	$\twoheadrightarrow$	$f($ $h(g(a), b)$ $, c)$	
because	`g(x)`	=	`h(g(x),b)`	and $\sigma = \{\texttt{x} \mapsto a\}$

On the opposite $t = f(a, c)$ cannot be rewritten by `g(x) = h(g(x),b)`.

Remark 2

Isabelle/HOL rewrites terms using equations in the order of the function definition and only from left to right.

Term rewriting – the quiz

Quiz 8

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

- *Rewriting the term $f(g(g(a)))$ with equation $g(x) = x$ is*
  V Possible ‖ R Impossible
- *To rewrite the term $f(g(g(a)))$ with $g(x) = x$ the substitution σ is*
  V $\{x \mapsto a\}$ ‖ R $\{x \mapsto g(a)\}$
- *Rewriting the term $f(g(g(y)))$ with equation $g(x) = x$ is*
  V Possible ‖ R Impossible
- *Rewriting the term $f(g(g(y)))$ with equation $g(f(x)) = x$ is*
  V Possible ‖ R Impossible

Isabelle evaluation = rewriting terms using equations

```
(1) append    Nil  x  =  x
(2) append (x#xs)  y  =  (x#(append xs y))
```

Rewriting the term: `append [1,2] [3,4]` with (1) then (2) (Rmk 2)

First, recall that `[1,2] = (1#(2#Nil))` and `[3,4] = (3#(4#Nil))`!

`append (1#(2#Nil)) (3#(4#Nil))` `(1# (append (2#Nil) (3#(4#Nil))))}` with $\sigma = \{x \mapsto 1, xs \mapsto (2\#Nil), y \mapsto (3\#(4\#Nil))\}$	$\not\twoheadrightarrow_{(1)}$ $\twoheadrightarrow_{(2)}$
`(1# (append (2#Nil) (3#(4#Nil))))` `(1# (2#(append Nil (3#(4#Nil)))))` with $\sigma = \{x \mapsto 2, xs \mapsto Nil, y \mapsto (3\#(4\#Nil))\}$	$\twoheadrightarrow_{(2)}$
`(1#(2# (append Nil (3#(4#Nil)))))` `(1#(2# (3#(4#Nil)) )) = [1,2,3,4] !` with $\sigma = \{x \mapsto (3\#(4\#Nil))\}$	$\twoheadrightarrow_{(1)}$

Example 24

See demo of step by step rewriting in Isabelle/HOL!

Isabelle evaluation = rewriting terms using equations (II)

```
(1) member e []         = False
(2) member e (x # xs)= (if e=x then True else (member e xs))
```

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

↠ `if 2=1 then True else (member 2 [2,3])`
  by equation (2), because `[1,2,3] = 1#[2,3]`

↠ `if False then True else (member 2 [2,3])`
  by Isabelle equations defining equality on naturals

↠ `member 2 [2,3]`
  by Isabelle equation `(if False then x else y = y)`

↠ `if 2=2 then True else (member 2 [3])`
  by equation (2), because `[2,3] = 2#[3]`

↠ `if True then True else (member 2 [3])`
  by Isabelle equations defining equality on naturals

↠ `True`
  by Isabelle equation `(if True then x else y = x)`

Lemma simplification= Rewriting + Logical deduction

```
(1) member e []         = False
(2) member e (x # xs)= (if e=x then True else (member e xs))
```

Proving the lemma: `member y [z,y,v]`

↠ `if y=z then True else (member y [y,v])`
  by equation (2), because `[z,y,v] = z#[y,v]`

↠ `if y=z then True else (if y=y then True else (member y [v]))`
  by equation (2), because `[y,v] = y#[v]`

↠ `if y=z then True else (if True then True else (member y [v]))`
  because `y=y` is trivially True

↠ `if y=z then True else True`
  by Isabelle equation `(if True then x else y = x)`

↠ `True`
  by logical deduction `(if b then True else True)` ⟷ `True`

Lemma simplification= Rewriting + Logical deduction (II)

```
(1) member e []         = False
(2) member e (x # xs)   = (if e=x then True else (member e xs))

(3) append [] x         = x
(4) append (x # xs) y   = x # (append xs y)
```

Exercise 7

Is it possible to prove the lemma `member u (append [u] v)` *by simplification/rewriting?*

Exercise 8

Is it possible to prove the lemma `member v (append u [v])` *by simplification/rewriting?*

Demo of rewriting in Isabelle/HOL!

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`

Scala export + Demo

To export functions to Haskell, SML, Ocaml, Scala .... `export_code`

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

```
export_code member index in Scala
```

test.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))
  }
}
```

Analyse et Conception Formelles

Lesson 3

–

Recursive Functions and Algebraic Data Types

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

Recursive Functions

- A function is recursive if it is defined using itself.
- Recursion can be direct

```
fun member:: "'a => 'a list => bool"
where
"member e []      = False" |
"member e (x#xs) = (e=x \/ (member e xs))"
```

- ... or indirect. In this case, functions are said to be mutually recursive.

```
fun even:: "nat => bool"
and odd::  "nat => bool"
where
  "even 0       = True"  |
  "even (Suc x) = odd x" |
  "odd 0        = False" |
  "odd (Suc x)  = even x"
```

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 $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ 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

$$\frac{\text{To prove termination of } f \quad f(t_1) \twoheadrightarrow f(t_2) \twoheadrightarrow \ldots}{\text{Prove that} \quad g(t_1) > g(t_2) > \ldots}$$

- The ordering $>$ is well founded on $\mathbb{N}$
  i.e. no infinite decreasing sequence of naturals $n_1 > n_2 > \ldots$

Terminating Recursive Functions (II)

How to guarantee the termination of a recursive function? (theory)

- If $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ 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

$$\frac{\text{To prove termination of } f \quad f(t_1) \twoheadrightarrow f(t_2) \twoheadrightarrow \ldots}{\text{Prove that} \quad g(t_1) > g(t_2) > \ldots}$$

Example 1 (Proving termination using a measure)

```
"member e []      = False" |
"member e (x#xs) = (if e=x then True else (member e xs))"
```

1. We define the measure $g = \lambda(x, y).\ (length\ y)$
2. We prove that $\forall$e x xs. g(e, (x#xs)) $>$ g(e, xs)

Proving termination with measure – the quiz

Quiz 1

- *Proving termination of a function f ensures that the evaluations of (f t) will terminate for* | V | *some t* | R | *all possible t* |
- *For a function* `f::'a list ⇒ 'a list` *a measure function should be of type* | V | `'a list ⇒ 'a list` | R | `'a list ⇒ nat` |
- *For the function* `f::nat list ⇒ nat list`
  ```
  "f [] = []" |
  "f (x#xs) = (if x=1 then [x] else xs)"
  ```

V	*We do not need a measure function*
R	*The only possible measure is* $\lambda x.\ (length\ x)$

- *For function* `f::nat list ⇒ nat list`
  ```
  "f [] = []" |
  "f (x#xs)= (if x=1 then (f(x#xs)) else (f xs))"
  ```

V	*There is no measure function*
R	*The only possible measure is* $\lambda x.\ (length\ x)$

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

[1] Actually, it tries to build a termination ordering but it has the same objective.

Terminating Recursive Functions (IV)

Example 2

A definition of the member function using `function` is the following:

```
function (sequential) member::"'a ⇒ 'a list ⇒ bool"
where
"member e []      = False" |
"member e (x#xs) = (if e=x then True else (member e xs))"

apply pat_completeness
apply auto
done

termination member
apply (relation "measure (λ(x,y). (length y))")
apply auto
done
```

Prove that the function is "complete" *i.e.* patterns cover the domain

Prove its termination using the measure proposed in Example 1

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, ⇒, *) 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 τ parameterized by types $(\alpha_1, \ldots, \alpha_n)$:

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \;=\; C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \;\mid\; \ldots \;\mid\; C_k\ \tau_{1,k} \ldots \tau_{1,n_k}$$

with $C_1, \ldots, C_n$ capitalized identifiers

Example 6 (The type of (polymorphic) lists, defined using `datatype`)

```
datatype 'a list = Nil
                 | Cons 'a  "'a list"
```


Building objects of Algebraic Data Types

Any definition of the form

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \;=\; C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \;\mid\; \ldots \;\mid\; C_k\ \tau_{1,k} \ldots \tau_{1,n_k}$$

also defines constructors $C_1, \ldots, C_k$ for objects of type $(\alpha_1, \ldots, \alpha_n)\tau$

The type of constructor C_i is $\tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$

Example 7

```
datatype 'a list = Nil
                 | Cons 'a  "'a list"
```

defines constructors

`Nil::'a list` and `Cons::'a` ⇒ `'a list` ⇒ `'a list`

Hence,

- `Cons (3::nat) (Cons 4 Nil)` is an object of type `nat list`
- `Cons (3::nat)` is an object of type `nat list` ⇒ `nat list`

Matching objects of Algebraic Data Types

Objects of Algebraic Data Types can be matched using case expressions:

```
(case l of Nil => ... | (Cons x r) => ...)
```

possibly with wildcards, i.e. "_"

```
(case i of 0 => ... | (Suc _) => ...)
```

and nested patterns

```
(case l of (Cons 0 Nil) => ... | (Cons (Suc x) Nil) => ...)
```

possibly embedded in a function definition

```
fun first::"'a list =>'a list"
 where
"first Nil = Nil" |
"first (Cons x _) = (Cons x Nil)"
```

```
fun first::"'a list =>'a list"
 where
"first [] = []" |
"first (x#_) = [x]"
```

Building objects of Algebraic Data Types – the quiz

Quiz 2 (we define `datatype abstInt= Any | Mint int`)

- *How to build an object of type* `abstInt` *from integer 13?*

V	13	R	(Mint 13)

- *How to build the object* `Any` *of type* `abstInt`*?*

V	Any	R	(Mint Any)

- *To check if a variable* `x::abstInt` *contains an integer how to do?*

V	`(case x of (Mint _) => True	Any => False)`
R	`x= (Mint _)`

- *Let* `f` *be defined by*

```
f::abstInt ⇒ abstInt ⇒ abstInt
"f (Mint x) (Mint y) = (Mint x+y)" |
"f _ _ = Any"
```

What is the value of:

`(f (Mint 1) (Mint 2))`		`(f Any (Mint 2))`	
V	`Any`	V	`Any`
R	`Mint 3`	R	*Undefined*

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 ⇒ 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 .... `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 ☹

Analyse et Conception Formelles

Lesson 4

–

Proofs with a proof assistant

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 | $\wedge$ | Equations for `nth` | $\wedge$ | Equations for `index` | $\longrightarrow$ `nth_index` |

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:

- `http://people.irisa.fr/Thomas.Genet/ACF/TPs/pc.thy`
- CM4 video and "Principes de preuve avancés" video

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

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_τ of size k for all types τ in $\neg\phi$
- Build all possible interpretations of functions in $\neg\phi$ on all D_τ
- Check if one interpretation satisfy $\neg\phi$ (this is a counterexample for ϕ)
- If not, there is no counterexample on a domain of size k for ϕ

`nitpick` algorithm:

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

Nitpick (II)

Exercise 1

By hand, iteratively check if there is a counterexample of cardinality 1, 2, 3 *for the formula* ϕ, *where* ϕ *is* `length la <= 1`.

Remark 1

- *The types occurring in* ϕ *are* `'a` *and* `'a list`
- **One** *possible domain* $D_{'a}$ *of cardinality* 1: $\{a_1\}$
- **One** *possible domain* $D_{'a\ list}$ *of cardinality* 1: $\{[\,]\}$ ~~$\{[a_1]\}$~~
  Domains have to be **subterm-closed**, *thus* $\{[a_1]\}$ *is not valid*
- **One** *possible domain* $D_{'a}$ *of cardinality* 2: $\{a_1, a_2\}$
- **Two** *possible domains* $D_{'a\ list}$ *of cardinality* 2: $\{[\,], [a_1]\}$ *and* $\{[\,], [a_2]\}$
- **One** *possible domain* $D_{'a}$ *of cardinality* 3: $\{a_1, a_2, a_3\}$
- **Twelve** *possible domains* $D_{'a\ list}$ *of cardinality* 3: $\{[\,], [a_1], [a_1, a_1]\}$, $\{[\,], [a_1], [a_2]\}$, $\{[\,], [a_1], [a_3, a_1]\}$, ... ~~$\{[\,], [a_1], [a_3, a_2]\}$~~ *(Demo!)*

Nitpick (III)

`nitpick` options:

- `timeout=t`, set the timeout to `t` seconds (`timeout=none` possible)
- `show_all`, displays the domains and interpretations for the counterex.
- `expect=s`, specifies the expected outcome where `s` can be `none` (no counterexample) or `genuine` (a counterexample exists)
- `card=i-j`, specifies the cardinalities to explore

For instance:

```
nitpick [timeout=120, show_all, card=3-5]
```

Exercise 2

- *Explain the counterexample found for* `rev l = l`
- *Is there a counterexample to the lemma* `nth_index`*?*
- *Correct the lemma and definitions of* `index` *and* `nth`
- *Is the lemma* `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 ϕ 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 (TP4bis)
- Narrowing: like exhaustive but taking advantage of symbolic values

No exhautiveness guarantee!! with any of them

`quickcheck` algorithm:

- Export Haskell code for functions and lemmas
- Generate test values of size 1 upto n and, test ϕ 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`.

Counter-example finders – the quiz

Quiz 1 (On (N)itpick and (Q)uickcheck counter-example finders)

- *If Q/N finds a counter-example on ϕ*

V	ϕ is contradictory
R	ϕ is not valid

- *If Q/N do not find a cex on ϕ*

V	ϕ is valid
R	We do not know anything

- *Which of Q/N is the most powerful?*

V	Q
R	N

Quiz 2 (If Isabelle/HOL accepts `lemma` ϕ closed by `done`)

- *Then*

V	ϕ is valid
R	ϕ is satisfiable

- *There may remain some counter-example*

V	True
R	False

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:

1. `apply case_tac`
2. `apply induct`
3. `apply auto`
4. `apply simp`
5. `sledgehammer`

How do proofs look like?

A formula of the form $A_1 \wedge \ldots \wedge A_n$ 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

$\bigwedge x_1 \ldots x_n.\ B$	meaning	prove B, or
$\bigwedge x_1 \ldots x_n.\ B \Longrightarrow C$	meaning	prove $B \longrightarrow C$, or
$\bigwedge x_1 \ldots x_n.\ B_1 \Longrightarrow \ldots B_n \Longrightarrow C$	meaning	prove $B_1 \wedge \ldots \wedge B_n \longrightarrow C$

and $\bigwedge x_1 \ldots x_n$ 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 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):                 goal (2 subgoals):
 1. A ⟹ B           becomes        1. F ⟹ A ⟹ B
                                    2. ¬ F ⟹ A ⟹ B
```

Exercise 4

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

Proof by cases (II)

Proof by cases 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 (3 subgoals):
goal (1 subgoal):                  1. x = Black ⟹ P x
 1. P (x::color)       becomes     2. x = White ⟹ P x
                                   3. x = Grey ⟹ P x
```

Exercise 5

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.*

Proof by induction

«Properties on recursive functions need proofs by induction»

Recall the basic induction principle on naturals:

$$P(0) \wedge \forall x \in \mathbb{N}.\,(P(x) \longrightarrow P(x+1)) \quad \longrightarrow \quad \forall x \in \mathbb{N}.\,P(x)$$

All recursive datatype have a similar induction principle, *e.g.* `'a lists`:

$$P([\,]) \wedge \forall e \in \texttt{'a}.\ \forall l \in \texttt{'a list}.(P(l) \longrightarrow P(e\#l)) \quad \longrightarrow \quad \forall l \in \texttt{'a list}.P(l)$$

Etc…

Example 4

`datatype 'a binTree= Leaf | Node 'a "'a binTree" "'a binTree"`

$$P(\text{Leaf}) \wedge \forall e \in \texttt{'a}.\ \forall t1\ t2 \in \texttt{'a binTree}.\ (P(t1) \wedge P(t2) \longrightarrow P(Node\ e\ t1\ t2)) \quad \longrightarrow \quad \forall t \in \texttt{'a binTree}.P(t)$$

Proof by induction (II)

$$P([\,]) \wedge \forall e \in \texttt{'a}.\ \forall l \in \texttt{'a list}.(P(l) \longrightarrow P(e\#l)) \quad \longrightarrow \quad \forall l \in \texttt{'a list}.P(l)$$

Example 5 (Proof by induction on lists)

Recall the definition of the function `append`:

```
    (1) append [] l   =  l
  (2) append (x#xs) l  =  x#(append xs l)
```

To prove $\forall l \in \texttt{'a list}.\,(append\ l\,[\,]) = l$ by induction on l, we prove:

1. $append\,[\,]\,[\,] = [\,]$, proven by the first equation of `append`
2. $\forall e \in \texttt{'a}.\ \forall l \in \texttt{'a list}.$
   $(append\ l\,[\,]) = l \ \longrightarrow\ (append\,(e\#l)\,[\,]) = (e\#l)$
   using the second equation of `append`, it becomes
   $(append\ l\,[\,]) = l \ \longrightarrow\ e\#(append\ l\,[\,]) = (e\#l)$
   using the (induction) hypothesis, it becomes
   $(append\ l\,[\,]) = l \ \longrightarrow\ e\#l = (e\#l)$

Proof by induction: `apply (induct x)`

To apply induction principle on variable `x` .... `apply (induct x)`

Conditions on the variable chosen for induction (induction variable):

- The variable `x` has to be of an inductive type (`nat`, `datatypes`, ...)
  Otherwise `apply (induct x)` fails
- The terms built by induction cases should easily be reducible!

Example 6 (Choice of the induction variable)

```
   (1) append [ ] l   =  l
(2) append (x#xs) l   =  x#(append xs l)
```

To prove $\forall l_1\ l_2 \in$ `'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 (induct x)` (II)

Exercise 6

Recall the datatype of binary trees we defined in lecture 3. Define and prove the following properties:

1. *If* `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 `sumList`*,* `sumNat` *and* `makeList` *of lecture 3. Try to state and prove the following properties:*

1. *Relate the length of list produced by* `makeList i` *and* `i`
2. *Relate the value of* `sumNat i` *and* `i`
3. *Give and try to prove the property relating those three functions*

Proof by induction: generalize the goals

By defaut `apply induct` may produce too weak induction hypothesis

Example 7

When doing an `apply (induct x)` on the goal `P (x::nat) (y::nat)`

```
goal (2 subgoals):
 1.  P 0 y
 2. ⋀x. P x y ⟹ P (Suc x) y
```

In the subgoals, `y` is fixed/constant!

Example 8

With `apply (induct x arbitrary:y)` on the same goal

```
goal (2 subgoals):
 1.  ⋀y. P 0 y
 2. ⋀x y. P x y ⟹ P (Suc x) y
```

The subgoals range over any `y`

Exercise 8

Prove the `sym` *lemma on the* `leq` *function.*

Proof by induction: : induction principles

Recall the basic induction principle on naturals:

$$P(0) \wedge \forall x \in \mathbb{N}.\ (P(x) \longrightarrow P(x+1)) \quad \longrightarrow \quad \forall x \in \mathbb{N}.\ P(x)$$

In fact, there are infinitely many other induction principles

- $P(0) \wedge P(1) \wedge \forall x \in \mathbb{N}.\ ((x > 0 \wedge P(x)) \longrightarrow P(x+1)) \quad \longrightarrow \quad \forall x \in \mathbb{N}.\ P(x)$
- ...
- Strong induction on naturals
  $\forall x, y \in \mathbb{N}.\ ((y < x \wedge P(y)) \longrightarrow P(x)) \quad \longrightarrow \quad \forall x \in \mathbb{N}.\ P(x)$
- Well-founded induction on any type having a well-founded order $<<$
  $\forall x, y.\ ((y << x \wedge P(y)) \longrightarrow P(x)) \quad \longrightarrow \quad \forall x.\ P(x)$

Proof by induction: : induction principles (II)

Apply an induction principle adapted to the function call (f x y z)
.... apply (induct x y z rule:f.induct)

Apply strong induction on variable x of type nat

.... apply (induct x rule:nat_less_induct)

Apply well-founded induction on a variable x
.... apply (induct x rule:wf_induct)

Exercise 9

Prove the lemma on function divBy2.

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

Combination of decision procedures auto and simp (II)

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:

Isabelle/HOL	Formula to prove + relevant definitions and lemmas ⟶ ⟵ Proof (click on it)	External **ATPs**[1] Local or in the Cloud

Prove the first subgoal using state-of-the-art[2] 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 explicitely 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.

[1] Automatic Theorem Provers

[2] See `http://www.tptp.org/CASC/`.

Sledgehammer (II)

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 `nth` *and* `index`

Exercise 11

Recall the functions `sumList`, `sumNat` *and* `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* `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 `prop1`, add relevant intermediate lemmas:

1. In the file, define a lemma **before** the property `prop1`
2. **Name** the lemma (say `lem1`) (to be used by sledgehammer)
3. Try to find a counterexample to `lem1`
4. If no counterexample is found, close the proof of `lem1` by `sorry`
5. Go back to the proof of `prop1` and check that `lem1` helps
6. If it helps then prove `lem1`. If not try to guess another lemma

To build correct theories, do not confuse `oops` and `sorry`:

- Always close an unprovable property by `oops`
- Always close an unfinished proof of a provable property by `sorry`

Example 10 (Everything is provable using contradictory lemmas)

We can prove that $1 + 1 = 0$ using a false lemma.

Analyse et Conception Formelles

Lesson 5

–

Crash Course on Scala

Bibliography

- *Programming in Scala*, M. Odersky, L. Spoon, B. Venners. Artima. `http://www.artima.com/pins1ed/index.html`
- *An Overview of the Scala Programming Language*, M. Odersky & al. `http://www.scala-lang.org/docu/files/ScalaOverview.pdf`
- *Scala web site*. `http://www.scala-lang.org`

Acknowledgements

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

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* ($\neq$ Java)
- With functional programming: higher-order, pattern-matching, ...
- Fully interoperable with Java (in both directions)
- Concise smart syntax ($\neq$ Java)
- A compiler and a read-eval-print loop integrated into the IDE

Scala worksheets!!

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

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

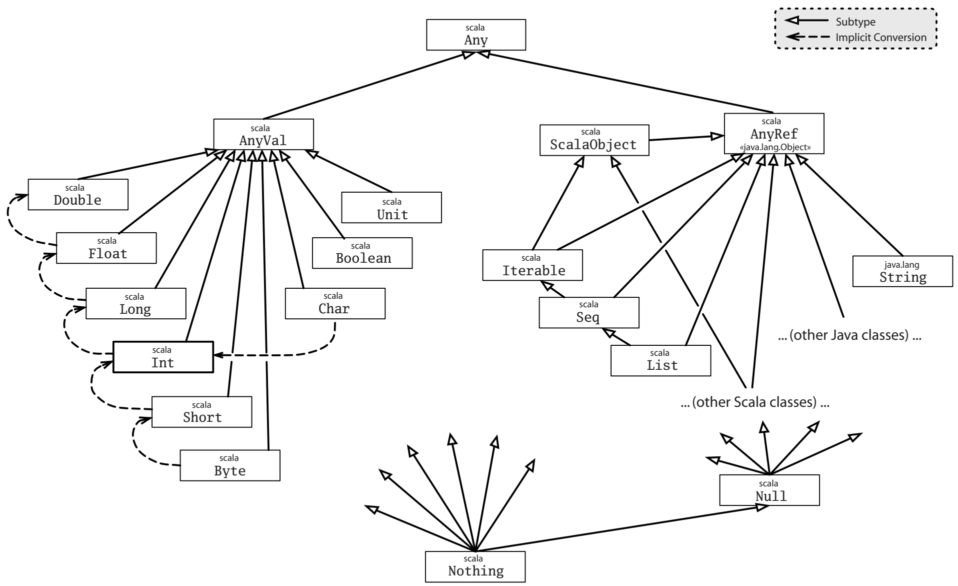
Base types and type annotations

- `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`)

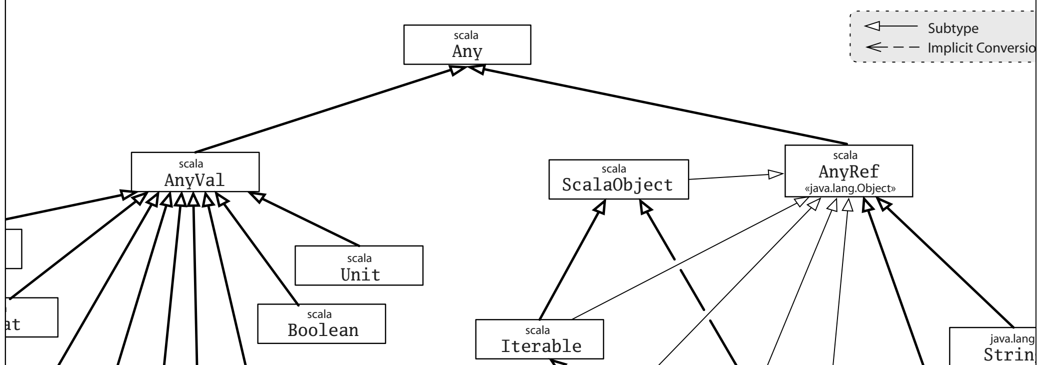
Exercise 1

Use the `max(Int)` *method of class* `Int` *to compute the maximum of* `1+2` *and* `4`*.*

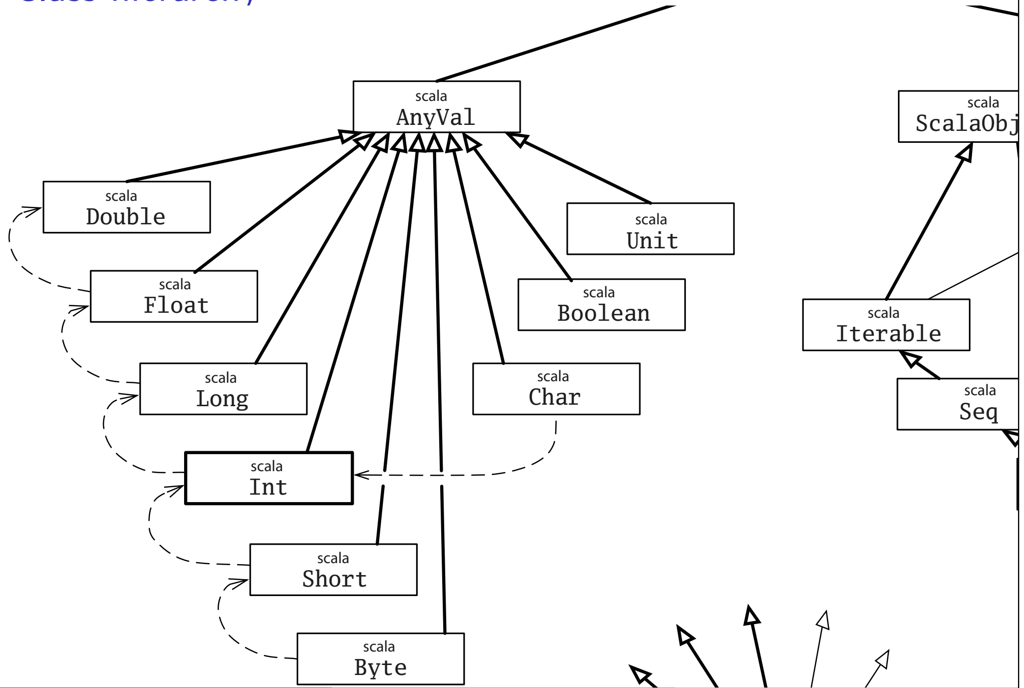
Class hierarchy



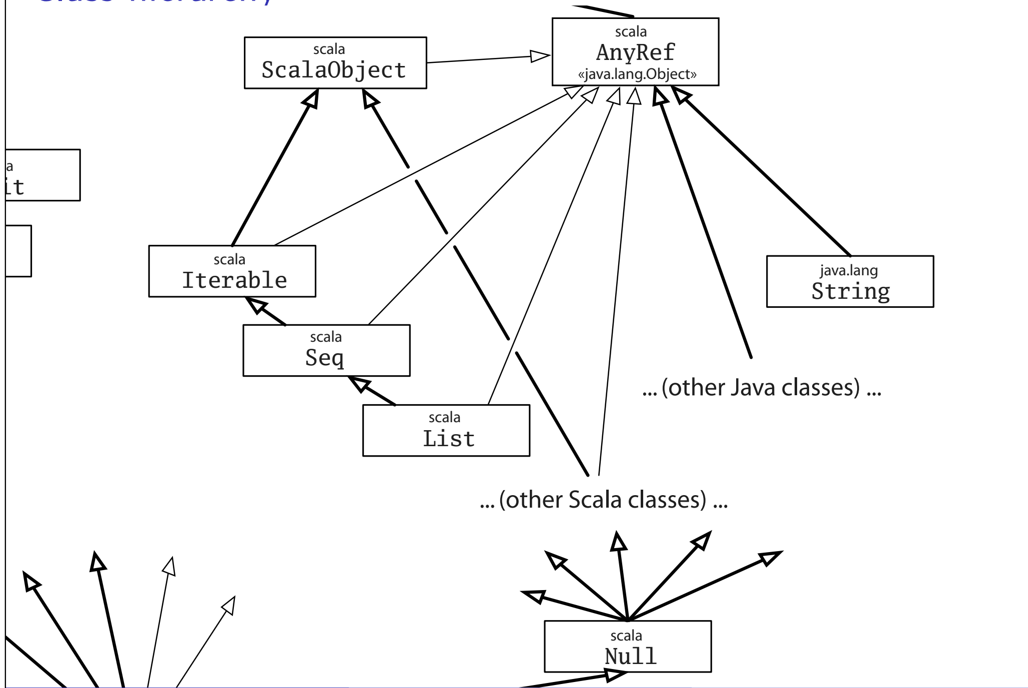
Class hierarchy



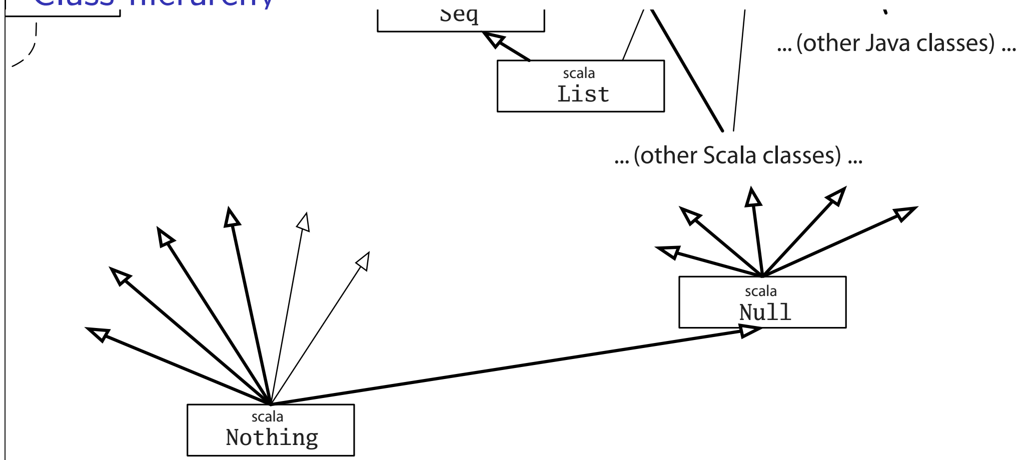
Class hierarchy



Class hierarchy



Class hierarchy



Subtyping and class hierarchy – the quiz

Quiz 1

❶	`12` *is of type* `Int`.	V	*True*	R	*False*
❷	`Int` *is a subtype of* `Any`.	V	*True*	R	*False*
❸	`12` *is of type* `Any`.	V	*True*	R	*False*
❹	`Int` *is a subtype of* `Double`.	V	*True*	R	*False*
❺	`12` *of type* `Double`.	V	*True*	R	*False*
❻	`null` *of type* `List`.	V	*True*	R	*False*
❼	`12` *of type* `Nothing`.	V	*True*	R	*False*
❽	`"toto"` *of type* `Any`.	V	*True*	R	*False*

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

Quiz 2 (What is the smallest type for the following expressions)

1	`if (1==2) 1 else 2`	V	Int	R	Any
2	`if (1==2) 1 else "toto"`	V	Int	R	Any
3	`if (1==2) 1`	V	AnyVal	R	Int
4	`if (1==1) println(1)`	V	Any	R	Unit

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

Match-case – the quiz

Quiz 3 (What is the value of the following expression?)

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

V	"hello"
R	"don't know"

Quiz 4 (What is the value of the following expression?)

```
val x= "bonj"
x match {
  case "au revoir" => "goodbye"
  case "bonjour" => "hello"
}
```

V	*Undefined*
R	"hello"

(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

```
l2 match {
    case Nil => 0
    case e::_ => e
}
```


Immutable lists – the quiz

Quiz 5 (Is this program valid?)

```
val li= List("zero","un","deux")
li(1)="one"
```

V Yes || R No

Quiz 6 (Is this program valid?)

```
var li= List("zero","un","deux")
li(1)="one"
```

V Yes || R No

Quiz 7 (Is this program valid?)

```
val li= List(1,"toto",2)
val l2= li ++ List(3,4)
```

V Yes || R No

Immutable lists – the quiz

Quiz 8 (Is this program valid?)

```
var li= List(1,2,3)
li= li ++ List(5,6)
```

V Yes || R No

Quiz 9 (What is the result printed by this program?)

```
val t1= Array(4,5,6)
val t2= t1
t2(1)= -4
println(t1(1))
```

V -4 || R 5

Quiz 10 (What is the result printed by this program?)

```
var li= List(1,2,3)
var l2= li
l2= l2.updated(1,10)
println(li(1))
```

V 10 || R 2

`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 2

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

Exercise 3

Using `for` and `println` build a usual 10×10 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 4

Print all the keys of map `m1`

Outline

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 1
```
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 5

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:Int) => x + 1)`
  Remark: it is written $(\lambda x. x + 1)$ 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 =>B) and results in a List[B]

```
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 6

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 2 (Other examples of partial application)

`(_:String).size`   `(_:Int) + (_:Int)`   `(_:String) == "toto"`

Exercise 7

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

Exercise 8

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`.

Outline

Class definition and constructors

- `class` C(v1: type1, ..., vn:typen) `{ ... }`
  (C(v1: type1, ..., vn:typen): the primary constructor)

  e.g.
```
class Rational(n:Int,d:Int){
    val num=n        // can use var instead
    val den=d        // to have mutable objects
    def isNull():Boolean=(this.num==0)
  }
```
- Objects instances can be created using `new`:
  `val r1= new Rational(3,2)`
- Fields and methods of an object can be accessed via "dot notation"
  `if (r1.isNull()) println("rational is null")`
  `val double_r1= new Rational(r1.num*2,r1.den)`

Exercise 9

Complete the `Rational` class with an `add(r:Rational):Rational` function.

Overriding, operator definitions and implicit conversions

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

Exercise 10

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 11

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 12

Add an implicit conversion from `Int` *to* `Rational`.

Traits

- Traits stands for interfaces (as in Java)

```
trait IntQueue {
      def get:Int
      def put(x:Int):Unit
}
```

- The keyword `extends` defines trait implementation

```
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`

```
trait IntQueue {
      def get:Int
      def put(x:Int):Unit
}

object InfiniteQueueOfOne extends IntQueue{
      def get=1
      def put(x:Int)={}
}
```

- A singleton object does not need to be "created" by `new`

```
InfiniteQueueOfOne.put(10)
InfiniteQueueOfOne.put(15)
val x=InfiniteQueueOfOne.get
```


Type abstraction and Polymorphism

Parameterized function/class/trait can be defined using type parameters

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

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}
}

def first[T1,T2](pair:(T1,T2)):T1=
      pair match case (x,y) => x
```

Case classes

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

```
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:

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


Case classes and pattern-matching

```
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
```

- `match case` can directly inspect objects built with case classes

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

Exercise 13

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

Outline

Interoperablity 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

```
import scala.collection.JavaConverters._

val l1:java.util.List[Int]= new java.util.ArrayList[Int]()
l1.add(1); l1.add(2); l1.add(3)  // l1: java.util.List[Int]

val sb1= l1.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

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

```
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,l,r)) = (if x=y then True else ((member x l)
                                                   ∨ (member x r)))"
```

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)) true
                        else member[A](x, l) || member[A](x, r))
    }
}
```


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

```
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:

```
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

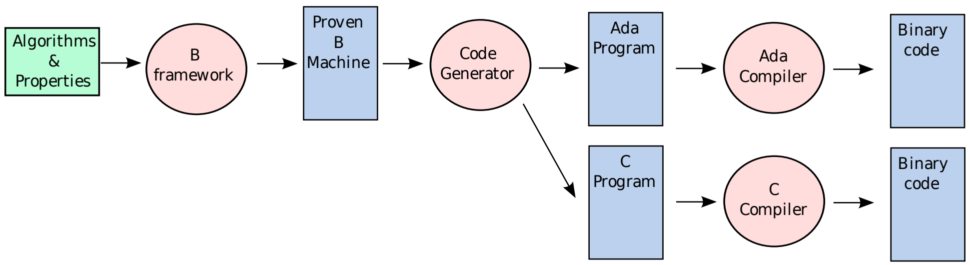
–

Certified Programming

Outline

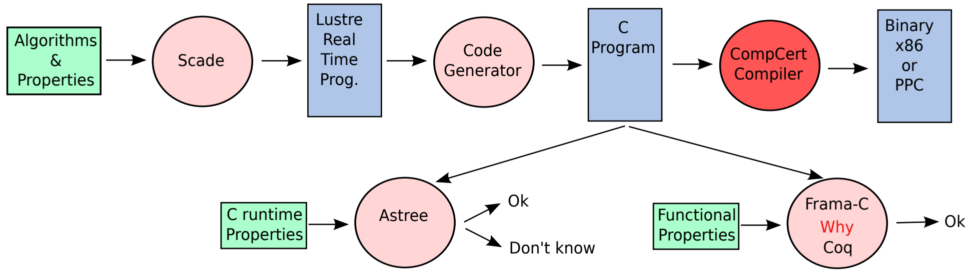
1. Certified program production lines
   - Some examples of certified code production lines
   - What are the weak links?
   - How to certify a compiler?
   - How to certify a static analyzer of code?
   - How to guarantee the correctness of proofs?
2. Methodology for formally defining programs and properties
   - Simple programs have simple proofs
   - Generalize properties when possible
   - Look for the smallest trusted base

B code production line



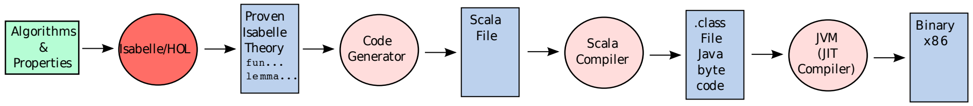
- The first certified code production line used in the industry
- For security critical code
- Used for onboard automatic train control of metro 14 (RATP)
- Several industrial users: RATP, Alstom, Siemens, Gemalto

Scade/Astree/CompCert code production line



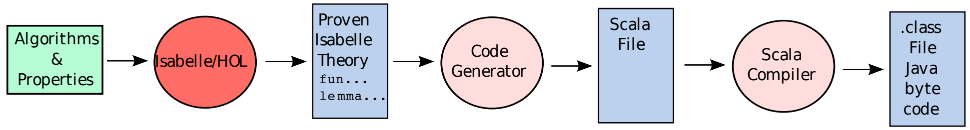
- The (next) Airbus code production line
- For security critical code (*e.g* flight control)
- Scade uses model-checking to verify programs or find counterexamples
- Astree is a static analyzer of C programs *proving* the absence of
  - division by zero, out of bound array indexing
  - arithmetic overflows
- 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
- Formalization/Verification of other parts is ongoing research
  e.g. some research efforts for certifying a JVM

What are the weak links of such lines?



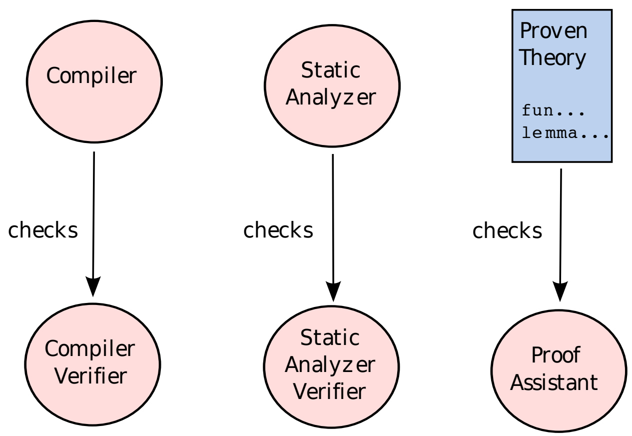
1. The initial choice of algorithms and properties
2. The verification tools (analyzers and proof assistants)
3. Code generators/compilers

$\Longrightarrow$ we need some guaranties on each link!

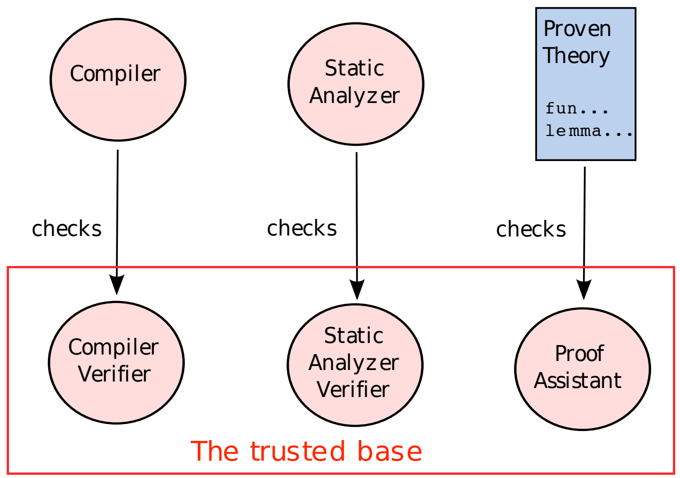
1. Certification of compilers
2. Certification of static analyzers
3. Verification of proofs in proof assistant
4. Methodology for formally defining algorithms and properties

$\Longrightarrow$ we need to limit the trusted base!

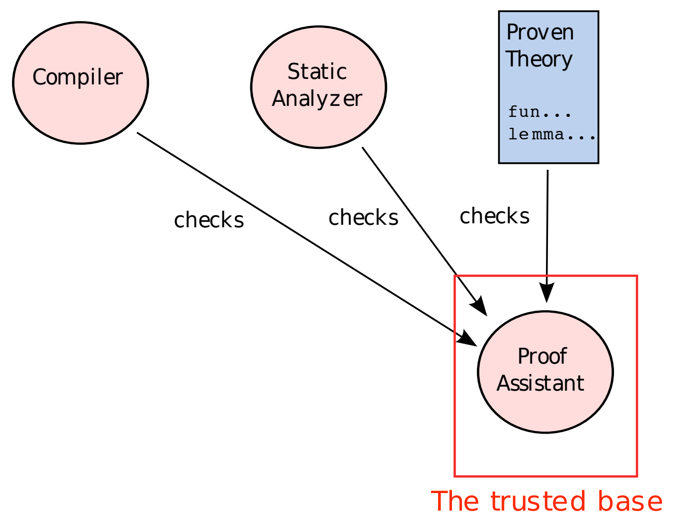
How to limit the trusted base?



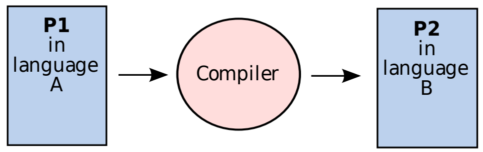
How to limit the trusted base?



How to limit the trusted base?



How to certify a compiler?

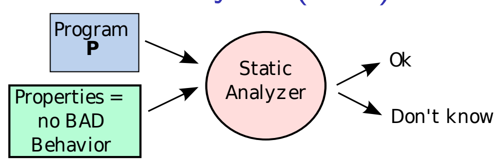


What is the property to prove? $\forall$ **P1**. **P1** «behaves» like **P2**

How can we prove this?

- Need to formally describe behaviors of programs:
  - Formal semantics for language A and language B
  - Close to defining an interpreter (using terms and functions) ($\approx$TP4) *i.e.* define `evalA(prog,inputs)` and `evalB(prog,inputs)`
- Then, prove that $\forall$ **P1 P2** s.t. **P2**=compil(**P1**):
  - $\forall$ inputs. `evalA`(**P1**,inputs) stops $\longleftrightarrow$ `evalB`(**P2**,inputs) stops, and
  - $\forall$ inputs. `evalA`(**P1**,inputs) = `evalB`(**P2**,inputs)
- Proving this by hand is unrealistic (recall the size of Java semantics)
- Use a proof assistant... compiler is correct if the proof assistant is!

How to certify a static analyzer (SAn)? (TP67)



What is the property to prove?

$\forall$ **P**. SAn(**P**)=True $\longrightarrow$ «nothing bad happens when executing **P**»

How can we prove this?

- Again, we need to formally describe behaviors of programs:
  - Formal semantics of language of **P**, define `eval(prog,inputs)`
- We need to formalize the analyzer and what is a «bad» behavior
  - Formalize «bad» , *i.e.* define a `BAD` predicate on program results
  - Formalize the analyser SAn
- Then, prove that the static analyzer is safe:
  $\forall$ **P**. $\forall$ inputs. (SAn(**P**)= True) $\longrightarrow$ $\neg$ `BAD(eval(`**P**,inputs`))`
- Again, proving this by hand is unrealistic
- Use a proof assistant... analyzer is correct if the proof assistant is!

Static analysis – the quiz

Quiz 1

- *What is a static analyzer good at?*

V	*Proving a property*
R	*Finding bugs*

- *Is a static analyzer running the program to analyze?*

V	*Yes*
R	*No*

- *Is a static analyzer has access to the user inputs?*

V	*Yes*
R	*No*

- *Given a program* **P**, `eval` *and* `BAD`, *can we verify by computation that for all* inputs, $\neg$ `BAD(eval(`**P**,inputs`))`*?*

V	*Yes*	R	*No*

- *Given a program* **P**, *and* `SAn` *can we verify by computation that* `SAn(`**P**`)=True`*?*

V	*Yes*	R	*No*

How to certify a static analyzer (SAn)? (II)

Isabelle file `cm6.thy`

Exercise 1

Define a static analyzer `san` *for such programs:*

`san:: program ⇒ bool`

Exercise 2

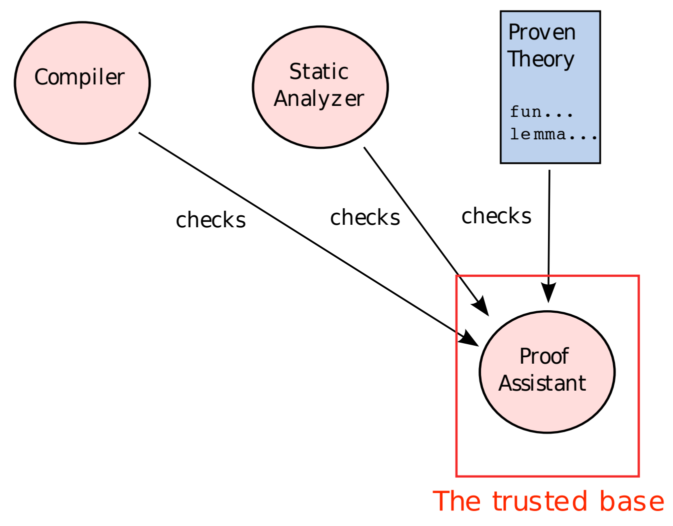
Define the `BAD` *predicate on program states:*

`BAD:: pgState ⇒ bool`

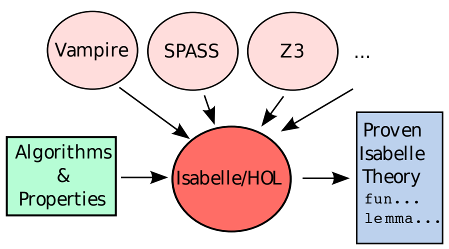
Exercise 3

Define the correctness lemma for the static analyzer `san`.

In the end, we managed to do this...



How to guarantee correctness of proofs in proof assistants?



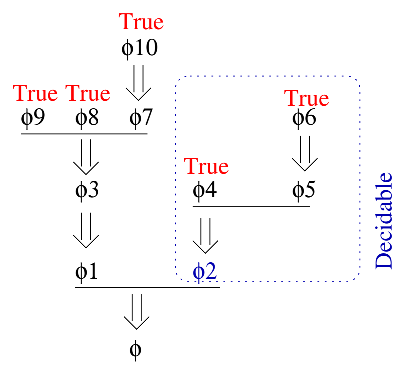
How to be convinced by the proofs done by a proof assistant?

- Relies on complex algorithms
- Relies on complex logic theories
- Relies on complex decision procedures

$\Longrightarrow$ 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 $\Longrightarrow$ fewer details

Decision Proc. $\Longrightarrow$ automatization

Axioms and decision procedures are the main weaknesses of proof assistants

Choices made in Coq, Isabelle/HOL, PVS, ACL2, etc. are very different

Proof handling : differences between proof assistants

	Coq	PVS	Isabelle	ACL2
Axioms	minimum and fixed	free	minimum and fixed	free
Decision procedures	proofs checked by Coq	trusted (no check)	proofs checked by Isabelle	trusted (no check)
Proof terms	built-in	no	additional	no
System automatization	basic	in between	in between	good
Counterexample generator	basic	basic	yes	yes

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 $(x + 4) * (y + 5) \geq x * y$ 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

Outline

1. 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?
2. Methodology for formally defining programs and properties
   1. Simple programs have simple proofs
   2. Generalize properties when possible
   3. Look for the smallest trusted base

Simple programs have simple proofs : Simple is beautiful

Example 2 (The `intersection` function of TP2/3)

An «optimized» version of `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 `lem1`, `lem2`, ... needed on `f`
3. (If necessary) program `fopt(x)` an optimized version of `f`
   - Optimize computation of `fopt`
   - Use optimized data structure if necessary
4. Prove that ∀ `x. f(x)=fopt(x)`
5. Using the previous lemma, prove again `lem1`, `lem2`, ... on `fopt`

Simple programs have simple proofs (II)

Exercise 4

The function `fastReverse` *is a tail-recursive version of* `reverse`*. Prove the classical lemmas on* `fastReverse` *using the same properties of* `reverse`*:*

- `fastReverse (fastReverse l)=l`
- `fastReverse (l1@l2)= (fastReverse l2)@(fastReverse l1)`

Exercise 5

Prove that the fast exponentiation function `fastPower` *enjoys the classical properties of exponentiation:*

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

Generalize properties when possible

Exercise 6 (On functions `member` and `intersection` of TP2/3)

- *Prove that*

`((member e l1) ∧ (member e l2)) ⟶ (member e (intersection l1 l2))`

- *How to generalize this property?*
- *What is the problem with the given function* `intersection`*?*

Exercise 7 (On 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`*?*

Exercise 8 (On functions `member` and `delete` of TP2/3)

- *Try to prove that*
  `member x l ⟶ member y l ⟶ x≠y ⟶ (member x (delete y l))`
- *How to generalize the property to ease the proof?*

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

To verify a function `f`, define lemmas using `f` and:

- functions of the trusted base
- other proven functions

Example 3

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

Remark: Then can be some interdependent functions to prove!

Example 4 (Prove a `parser` and a `prettyPrinter` on programs)

- `parser:: string ⇒ prog`
- `prettyPrinter:: prog ⇒ string`

The property to prove is: ∀ `p. parser(prettyPrinter p) = p`

`prettyPrinter` is more likely to be trusted since it is simpler

Analyse et Conception Formelles

Lesson 7

–

Program verification methods

Outline

1. Testing
2. Model-checking
3. Assisted proof
4. Static Analysis
5. A word about protoypes/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.

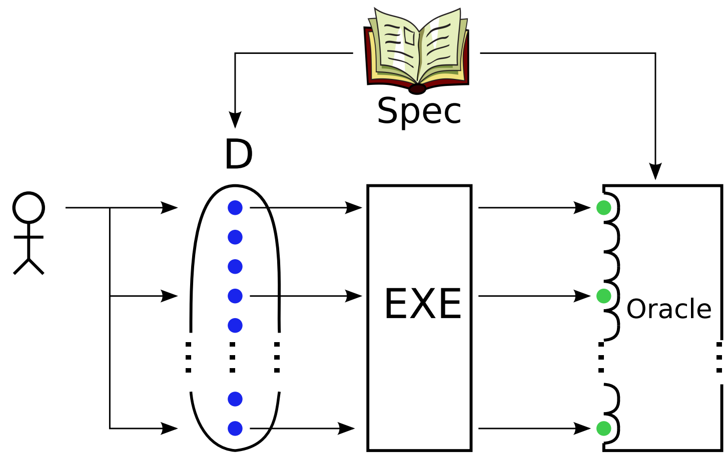
Definition 4 (Domain (of Definition))

The set of all possible inputs of a program, as defined by the specification.

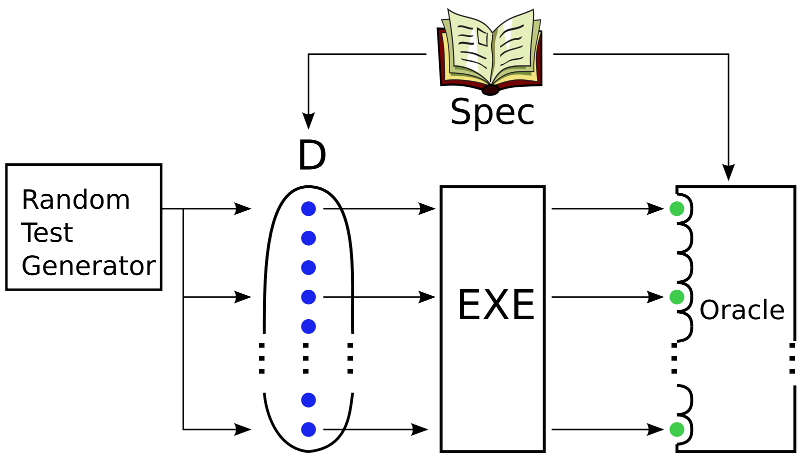
Notations

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

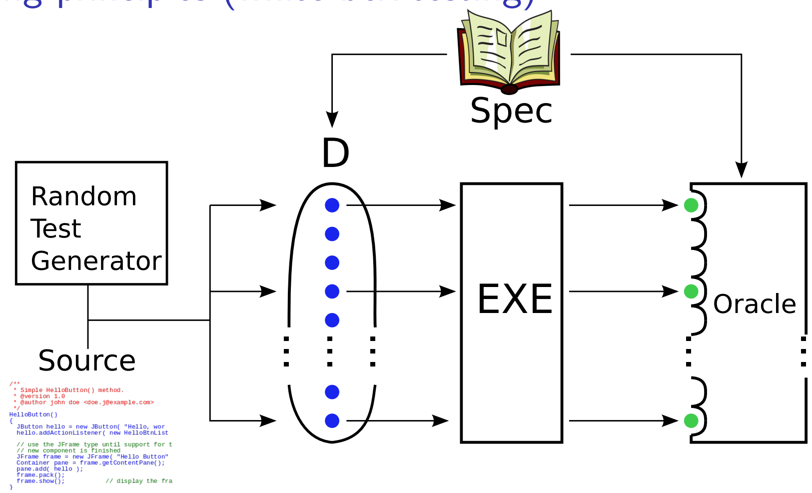


Testing principles (random generators)



This is what Isabelle/HOL `quickcheck` does (and TP4Bis)

Testing principles (white box testing)



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)

```
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 2;}
 else return 3;
}
```


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:

```
@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);
    }
}
```


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

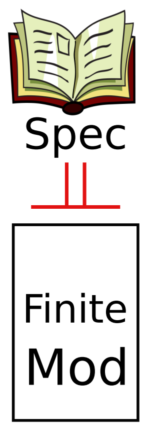
Some tool names

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

One killer result

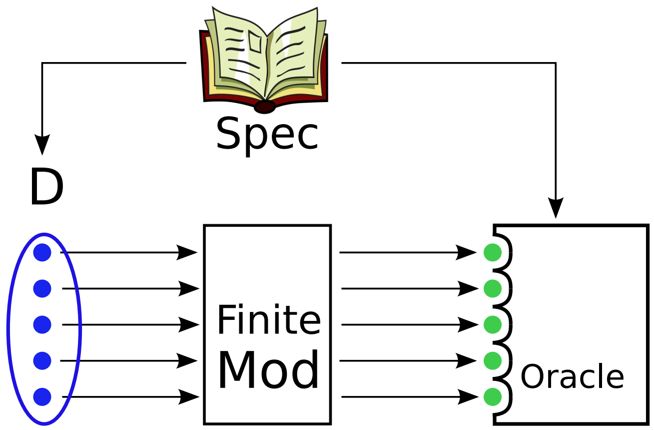
SAGE (running on 200 PCs/year) found 1/3 of security bugs in Windows 7
`https://www.microsoft.com/en-us/security-risk-detection/`

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)



Where D, Mod and Oracle are finite.

Model-checking principle explained in Isabelle/HOL

Automaton `digiCode.as` and Isabelle file `cm7.thy`

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 the Final state by typing three letters is to have typed A,B,C.

Model-checking, to sum-up

Strong and weak points

- \+ Automatic and efficient
- \+ Can find bugs and prove the property
- – For finite models only (*e.g* not on source code!)
- \+ Can deal with huge finite models (10^{120} states)
  More than the number of atoms in the universe!
- \+ Can deal with finite abstractions of infinite models *e.g.* source code
- – Incomplete on abstractions (but can find real bugs!)

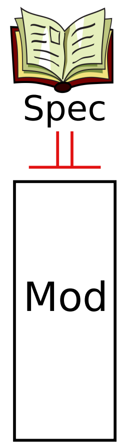
Some tool names

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

One killer result

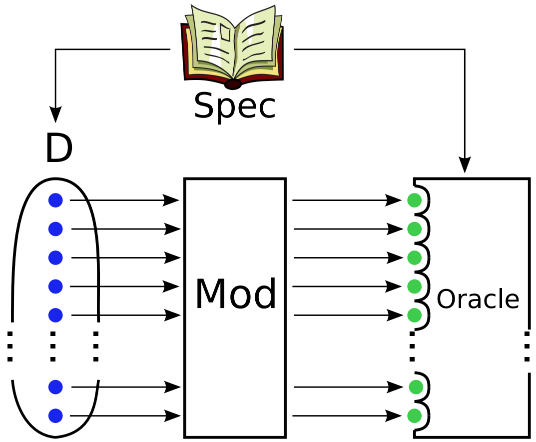
INTEL processors are commonly model-checked

Assisted proof principles



Where $\models$ is the usual logic consequence. This is proven directly on formulas Mod and Spec. This proof guarantees that...

Assisted proof principles (II)



Where D, Mod, Oracle can be infinite.

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

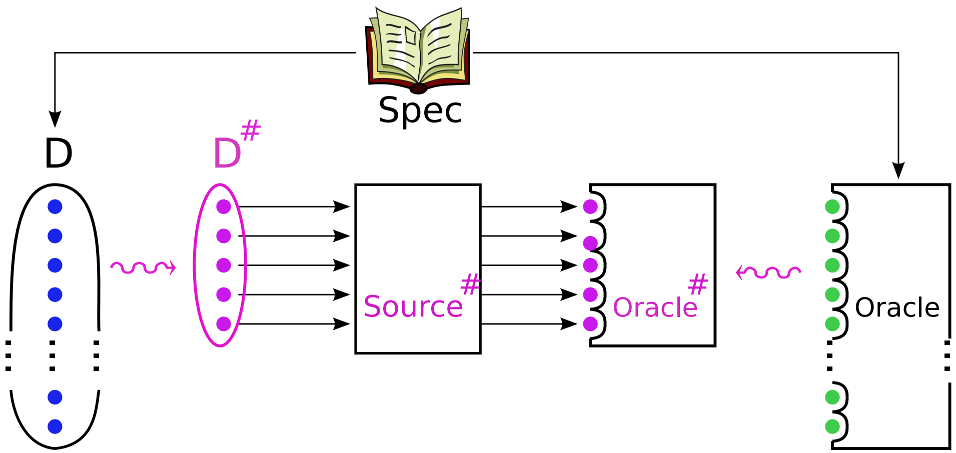
Some tool names

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

One killer result

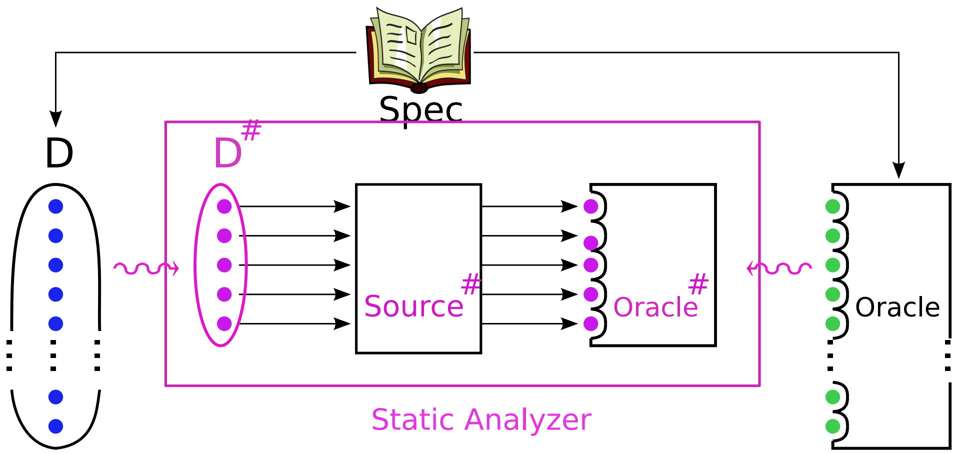
CompCert certified C compiler

Static Analysis principles



Where abstraction ⇝ is a correct abstraction

Static Analysis principles (II)



Where abstraction ⇝ is a correct abstraction

Static Analysis principles – Abstract Interpretation (III)

The abstraction '⇝' is based on the abstraction function abs:: D ⇒ $D^\#$

Depending on the verification objective, precision of abs can be adapted

Example 7 (Some abstractions of program variables for D=int)

(1) abs:: int ⇒ {⊥, ⊤} where ⊥ ≡ "undefined" and ⊤ ≡ "any int"

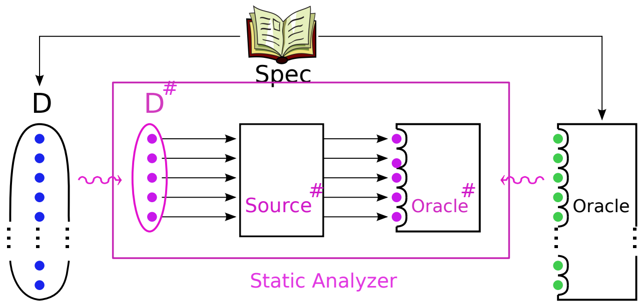
(2) abs:: int ⇒ {⊥, Neg, Pos, Zero, NegOrZero, PosOrZero, ⊤}

(3) abs:: int ⇒ {⊥} ∪ Intervals on $\mathbb{Z}$

Example 8 (Program abstraction with abs (1), (2) and (3))

	(1)	(2)	(3)
`x:= y+1;`	x=⊥	x=⊥	x=⊥
`read(x);`	x=⊤	x=⊤	x=]-∞;+∞[
`y:= x+10`	y=⊤	y=⊤	y=]-∞;+∞[
`u:= 15;`	u=⊤	u=Pos	u=[15;15]
`x:= |x|`	x=⊤	x=PosOrZero	x=[0;+∞[
`u:= x+u;`	u=⊤	u=Pos	u=[15;+∞[

Static Analysis: proving the correctness of the analyzer

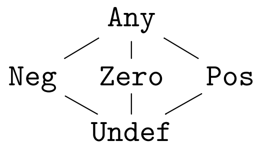


- Formalize semantics of Source language, *i.e.* formalize an `eval`
- Formalize the oracle: `BAD` predicate on program states
- Formalize the abstract domain $D^\#$
- Formalize the static analyser `SAn`:: program ⇒ bool
- Prove correctness of `SAn`: ∀ **P**. `SAn`(**P**) ⟶ (¬ `BAD`(`eval`(**P**)))
- ... Relies on the proof that ⇝ is a correct abstraction

Static Analysis principle explained in Isabelle/HOL

To abstract `int`, we define `absInt` as the abstract domain ($D^\#$):

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



Remark 1

Have a look at the concretization function (called `concrete`*) defining sets of integers represented by abstract elements* `Neg`, `Zero`, *etc.*

Exercise 3

Define the function `absPlus`:: `absInt` ⇒ `absInt` ⇒ `absInt` *(noted* $+^\#$*)*

Exercise 4 (Prove that $+^\#$ is a correct abstraction of +)

$x \in \texttt{concrete}(x^a) \wedge y \in \texttt{concrete}(y^a) \longrightarrow (x+y) \in \texttt{concrete}(x^a +^\# y^a)$

Static Analysis, to sum-up

Strong and weak points

+ Can prove the property
+ Automatic
+ On the source code
– Not designed to find bugs

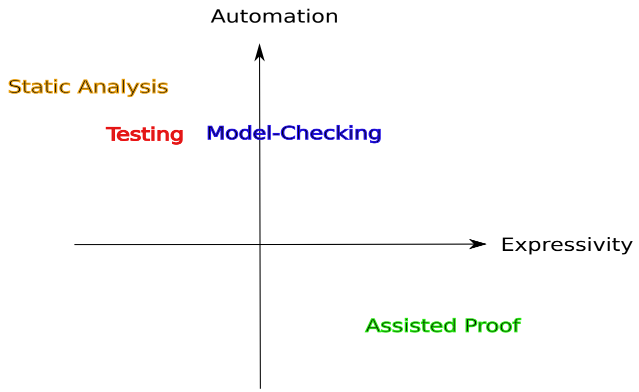
Some tool names

Astree (Airbus), Polyspace, Sawja, Infer (Facebook)...

One killer result

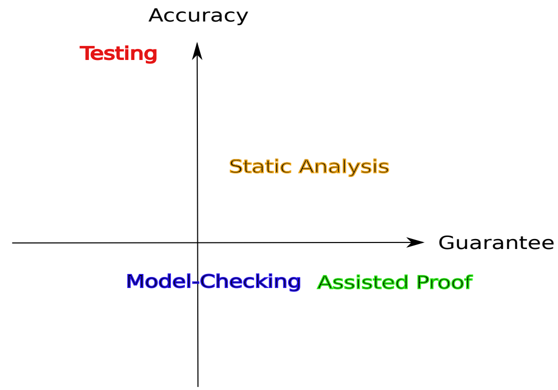
Astree was used to successfully analyze 10^6 lines of code of the Airbus A380 flight control system

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

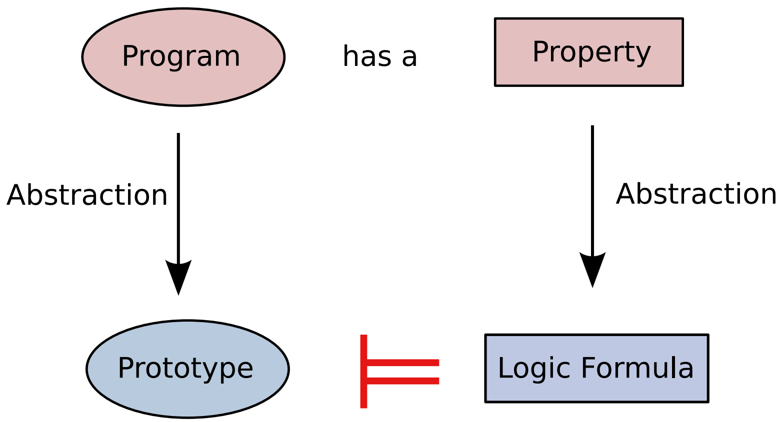
To sum-up on all presented techniques



- 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:



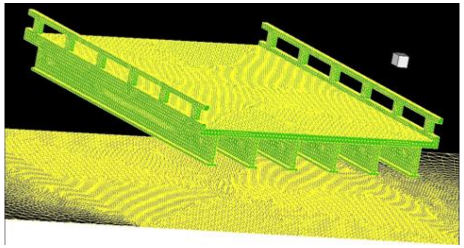
This is the case for model-checking and assisted proof.

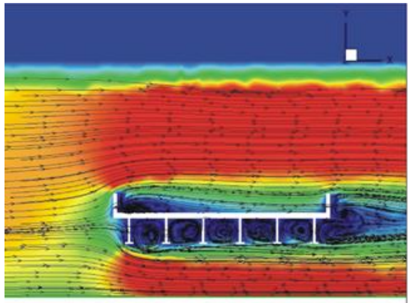
Testing prototypes is a common practice in engineering

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!

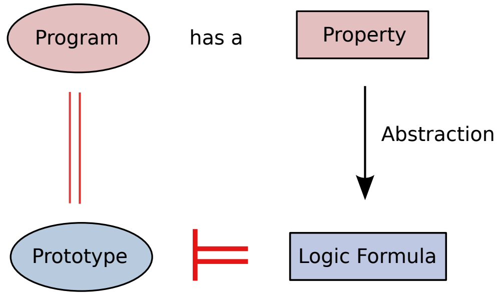
Problem on the prototype $\longrightarrow$ Problem on the real system

But in general, we do not have the opposite:

No problem on the prototype $\not\longrightarrow$ 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: [TP5, TP67, TP89, CompCert]

No problem on the prototype $\longrightarrow$ 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 $\xrightarrow{\text{Abstraction}}$ 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?

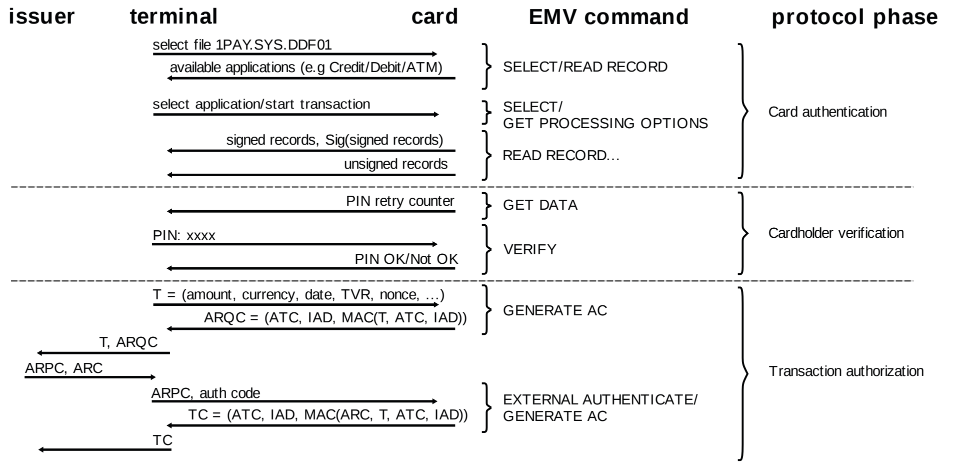
Yes, they do!

Hackers do read scientific papers!

Chip and PIN is Broken

Steven J. Murdoch, Saar Drimer, Ross Anderson, Mike Bond
University of Cambridge
Computer Laboratory
Cambridge, UK

Conference
Security and Privacy
2010
13 pages



Hackers do read scientific papers!

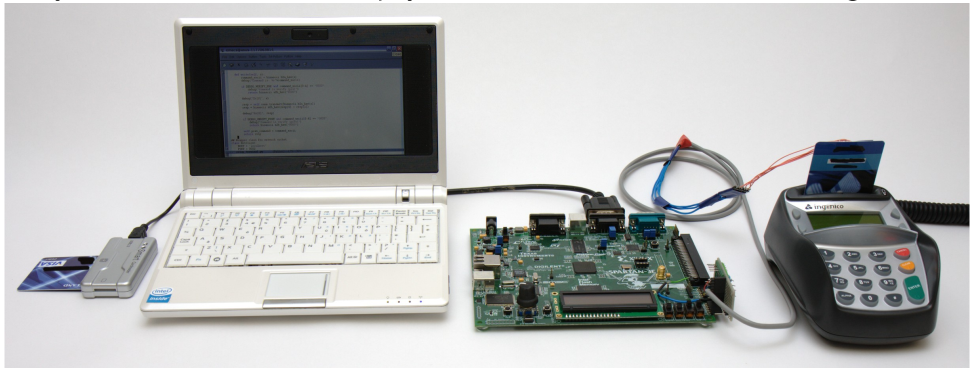
Chip and PIN is Broken

Steven J. Murdoch, Saar Drimer, Ross Anderson, Mike Bond
University of Cambridge
Computer Laboratory
Cambridge, UK

Conference
Security and Privacy
2010
13 pages

They revealed a weakness in the payment protocol of EMV

They showed how to make a payment with a card without knowing the PIN



Hackers do read scientific papers!

When Organized Crime Applies Academic Results
A Forensic Analysis of an In-Card Listening Device

Houda Ferradi, Rémi Géraud, David Naccache, and Assia Tria

[1] École normale supérieure
Computer Science Department
45 rue d'Ulm, F-75230 Paris CEDEX 05, France

Journal of
Cryptographic Engineering
2015

Hackers do read scientific papers!

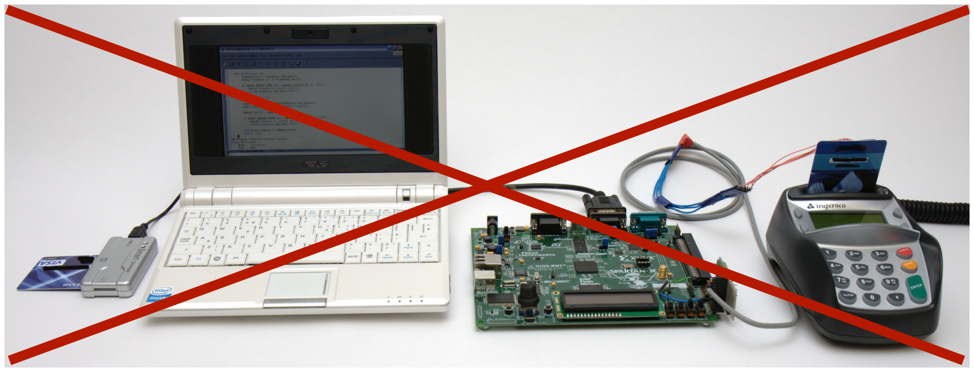
When Organized Crime Applies Academic Results
A Forensic Analysis of an In-Card Listening Device

Houda Ferradi, Rémi Géraud, David Naccache, and Assia Tria

[1] École normale supérieure
Computer Science Department
45 rue d'Ulm, F-75230 Paris CEDEX 05, France

Journal of
Cryptographic Engineering
2015

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!

Isabelle/HOL basics

This is only a short memo for Isabelle/HOL. For a more detailed documentation, please refer to
`http://isabelle.in.tum.de/website-Isabelle2020/documentation.html`

1 Survival kit

1.1 ASCII Symbols used in Logic Formulas

Symbol	ASCII
True	True
False	False
$\wedge$	/\

Symbol	ASCII
$\vee$	\/
$\neg$	~
$\neq$	~=

Symbol	ASCII
$\longrightarrow$	-->
$\leftrightarrow$	=
$\forall$	ALL

Symbol	ASCII
$\exists$	?
λ	%
$\Rightarrow$	=>

1.2 Lemma declaration and visualization

- declare a lemma (resp. theorem) .... `lemma` (resp. `theorem`)

```
lemma "A --> (B \/ A)"
lemma deMorgan: "~(A /\ B)=(~A \/ ~B)"
```

- to visualize the lemma/theorem/simplification rule associated to a given name.......... `thm`

```
thm "deMorgan"
thm "append.simps"
```

- to find and visualize all the lemmas/theorems/simplification rules defined using given symbols `find_theorems`

```
find_theorems "append" "_ + _"
```

1.3 Basic Proof Commands

- search for a counterexample for the first subgoal using SAT-solving .... `nitpick`
- search for a counterexample for the first subgoal using automatic testing .... `quickcheck`
- automatically solve or simplify all subgoals .... `apply auto`
- close the proof of a proven lemma or theorem .... `done`

```
lemma "A --> (B \/ A)"
apply auto
done
```

- abandon the proof of an unprovable lemma or theorem .... `oops`

```
lemma "A /\ B"
nitpick
oops
```

- abandon the proof of a (potentially) provable lemma or theorem .... `sorry`

1.4 Evaluation

- evaluate a term .... `value`

```
value "(1::nat) + 2"          value "[x,y] @ [z,u]"          value "(%x y. y) 1 2"
```

1.5 Basic Definition Commands

- associate a name to a value (or a function) .... `definition`

```
definition "l1=[1,2]"          definition "l2= l1@l1"          definition "f= (%x y. y)"
```

- define a function using equations .... `fun`

```
fun count:: "'a => 'a list => nat"
where
"count _    []  = 0" |
"count e (x#xs) = (if e=x then (1+(count e xs)) else (count e xs))"
```

- define an Abstract Data Type .... `datatype`

```
datatype 'a list = Nil | Cons 'a  "'a list"
```

1.6 Code exportation

- export code (in `Scala, Haskell, OCaml, SML`) for a list of functions .... `export_code`

```
export_code function1 function2 function3 in Scala
```

2 To go further... and faster

- apply structural induction on a variable `x` of an inductive type .... `apply (induct x)`
- apply an induction principle adapted to the function call `(f x y z)` . `apply (induct x y z rule:f.induct)`
- automatically solve or simplify the first subgoal .... `apply simp`
- insert an already defined lemma `lem` in the current subgoal .... `apply (insert lem)`
- do a proof by cases on a variable `x` or on a formula `F` ...... `apply (case_tac "x")` or `apply (case_tac "F")`
- try to prove the first subgoal with Sledgehammer .... Plugins>Isabelle>Sledgehammer
- set the goal number i as the first goal .... `prefer i`
- options of `nitpick`
  - `timeout=t`, nitpick searches for a counterexample during at most `t` seconds. (`timeout=none` is also possible)
  - `show_all`, nitpick displays the chosen domains and interpretations for the counterexample to hold.
  - `expect=s`, specifies the expected outcome of the nitpick call, where `s` can be `none` (no found counterexample) or `genuine` (a counterexample has been found).
  - `card=i-j`, specifies the cardinalities to use for building the SAT problem.
  - `eval=l`, gives a list `l` of terms to eval with the values found for the counterexample.

  ```
  nitpick [timeout=120, card=3-5, eval= "member e l" "length l"]
  ```

- options for `quickcheck`
  - `timeout=t`, quickcheck searches for a counterexample during at most `t` seconds.
  - `tester=tool`, specifies the type of testing to perform, where `tool` can be `random`, `exhaustive` or `narrowing`.
  - `size=i`, specifies the maximal size of the search space of testing values.
  - `expect=s`, specifies the expected outcome of quickcheck, where `s` can be `no_counterexample` (no found counterexample), `counterexample` (a counterexample has been found) or `no_expectation` (we don't know).
  - `eval=l`, gives a list `l` of terms to eval with the values found for the counterexample. Not supported for `narrowing` and `random` testers.

  ```
  quickcheck [tester=narrowing, eval=["member e l","length l"]]
  ```

- setting option values for all calls to `nitpick` .... `nitpick_params`

```
nitpick_params [timeout=120, expect=none]
```

- setting option values for all calls to `quickcheck` .... `quickcheck_params`

```
quickcheck_params [tester=narrowing, timeout=500]
```