Experiments in Verification

SS 2011

Christian Sternagel

Computational Logic
Institute of Computer Science
University of Innsbruck

March 11, 2011

Today's Topics

- Organization
- Formal Verification
- Isabelle/HOL Basics
- Functional Programming in HOL

Organization

Lecture

- LV-Nr. 703523
- VO 1
- `http://cl-informatik.uibk.ac.at/teaching/ss11/eve/`
- slides are also available online
- office hours: Tuesday 12:00 – 14:00 in 3N01
- online registration required before 23:59 on March 31
- grading: semester project

Lecture

- LV-Nr. 703523
- VO 1
- http://cl-informatik.uibk.ac.at/teaching/ss11/eve/
- slides are also available online
- office hours: Tuesday 12:00 – 14:00 in 3N01
- online registration required before 23:59 on March 31
- grading: semester project

Schedule

The lecture is blocked to 4 sessions of 3 hours each. The sessions take place on:

session 1	March	11
session 2	March	25
session 3	April	1
session 4	April	15

The Project

- after last session (on April 15) projects will be distributed
- work alone or in small groups
- projects have to be finished before August 1
- on delivery you will have to answer questions about your project

Formal Verification

What is Verification?

- part of software testing process
- part of V&V (verification and validation)

  verification: built right (software meets specifications)
  validation: built right thing (software fulfills intended purpose)

What is Verification?

- part of software testing process
- part of V&V (verification and validation)

  verification: built right (software meets specifications)

  validation: built right thing (software fulfills intended purpose)

Formal Verification

Proving or disproving the correctness of intended algorithms with respect to a certain formal specification.

Model-Theoretic (Model Checking)

systematically exhaustive exploration of the mathematical model

Model-Theoretic (Model Checking)

systematically exhaustive exploration of the mathematical model

Proof-Theoretic (Logical Inference)

theorem proving software

Example – Verification

given set of formulas $\Phi = \{\neg A, B \longrightarrow A, B\}$; check whether it is valid

Example – Verification

given set of formulas $\Phi = \{\neg A, B \longrightarrow A, B\}$; check whether it is valid

Truth Table (Model-Theoretic)

A	B	$\neg A$	$B \longrightarrow A$	Φ
0	0	1	1	0
0	1	1	0	0
1	0	0	1	0
1	1	0	1	0

Example – Verification

given set of formulas $\Phi = \{\neg A, B \longrightarrow A, B\}$; check whether it is valid

Example – Verification

given set of formulas $\Phi = \{\neg A, B \longrightarrow A, B\}$; check whether it is valid

Natural Deduction Proof (Proof-Theoretic)

1	$\neg A$	premise
2	$B \longrightarrow A$	premise
3	B	premise
4	$\neg B$	MT 2, 1
5	$\bot$	$\neg$e 3, 4

Model-Theoretic (Model Checking)

systematically exhaustive exploration of the mathematical model

Proof-Theoretic (Logical Inference)

theorem proving software

We focus on *logical inference* using Isabelle/HOL

Isabelle/HOL Basics

System Architecture

Standard ML implementation language

System Architecture

Isabelle/Pure generic proof assistant

Standard ML implementation language

System Architecture

Isabelle/HOL Higher-Order Logic

Isabelle/Pure generic proof assistant

Standard ML implementation language

System Architecture

Proof General Emacs interface

Isabelle/HOL Higher-Order Logic

Isabelle/Pure generic proof assistant

Standard ML implementation language

System Architecture

Isabelle/jEdit jEdit based interface

Isabelle/Scala connects ML to JVM

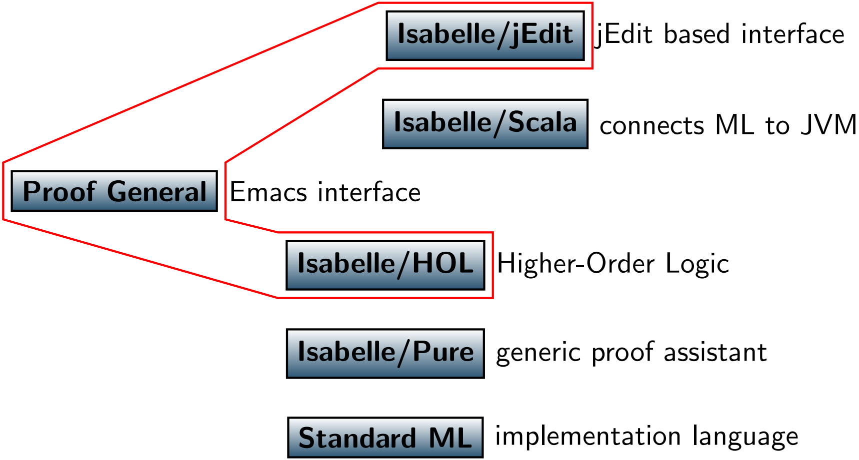
Proof General Emacs interface

Isabelle/HOL Higher-Order Logic

Isabelle/Pure generic proof assistant

Standard ML implementation language

System Architecture



Higher-Order Logic

- HOL = Functional Programming + Logic
- datatypes (**`datatype`**)
- recursive functions (**`fun`**)
- logical operators ($\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$, ...)

Setup of the Isabelle System

- custom settings in file `~/.isabelle/Isabelle2011/etc/settings`
- you will need at least:
  `ISABELLE_DOC_FORMAT=pdf`
  `PDF_VIEWER=`⟨*program*⟩

Setup of the Isabelle System

- custom settings in file `~/.isabelle/Isabelle2011/etc/settings`
- you will need at least:
  `ISABELLE_DOC_FORMAT=pdf`
  `PDF_VIEWER=`⟨*program*⟩

Main Component

- `isabelle doc`: for documentation
- `isabelle emacs`: interactive proof development in ProofGeneral (i.e., `$ isabelle emacs` ⟨*File*⟩`.thy`)
- `isabelle jedit`: interactive proof development in jEdit (i.e., `$ isabelle jedit` ⟨*File*⟩`.thy`)

Proof General – Useful Shortcuts

`Ctrl + C, Ctrl + Backspace`	undo and delete last step
`Ctrl + C, Ctrl + B`	go to bottom
`Ctrl + C, Ctrl + C`	interrupt process
`Ctrl + C, Ctrl + F`	find (lemmas, theorems, definitions, ...)
`Ctrl + C, Ctrl + N`	next step
`Ctrl + C, Ctrl + Return`	go to cursor position
`Ctrl + C, Ctrl + U`	undo last step
`Ctrl + C, Ctrl + V`	evaluate Isabelle command
`Ctrl + C, Ctrl + W`	clear output window
`Ctrl + G`	abort current emacs-command

Theory Files (`*.thy`) – General Structure

theory *Name* **imports** $T_1 \dots T_n$ **begin**

...

end

Theory Files (*.thy) – General Structure

theory *Name* **imports** $T_1 \ldots T_n$ **begin**

...

end

Explanation

- content of file `Name.thy`
- creates a new theory called *Name*
- depending on theories T_1 to T_n
- all proofs and definitions go between **begin** and **end**

Theory Files (*.thy) – General Structure

theory *Name* **imports** $T_1 \ldots T_n$ **begin**

...

end

Explanation

- content of file `Name.thy`
- creates a new theory called *Name*
- depending on theories T_1 to T_n
- all proofs and definitions go between **begin** and **end**

Example – Empty.thy

```
theory Empty imports Main begin end
```

Types

τ	$\stackrel{\text{def}}{=}$	`bool` | `nat` | ...	base types
	|	`'a` | `'b` | ...	type variables
	|	τ `=>` τ	total functions
	|	τ `*` τ	pairs
	|	τ `list`	lists
	|	...	user-defined types

Types

$$
\begin{array}{rcll}
\tau & \overset{\text{def}}{=} & \texttt{bool} \mid \texttt{nat} \mid \ldots & \text{base types} \\
 & \mid & \texttt{'a} \mid \texttt{'b} \mid \ldots & \text{type variables} \\
 & \mid & \tau \;\texttt{=>}\; \tau & \text{total functions} \\
 & \mid & \tau \;\texttt{*}\; \tau & \text{pairs} \\
 & \mid & \tau \;\texttt{list} & \text{lists} \\
 & \mid & \ldots & \text{user-defined types}
\end{array}
$$

Remark (Function Type is Right-Associative)

$$\tau_1 \;\texttt{=>}\; \tau_2 \;\texttt{=>}\; \tau_3 \quad \equiv \quad \tau_1 \;\texttt{=>}\; (\tau_2 \;\texttt{=>}\; \tau_3)$$

Examples – Types

Examples – Types

`nat` a natural number, e.g., `0`

Examples – Types

`nat`	a natural number, e.g., `0`
`nat => bool`	a predicate on nats, e.g., `even`

Examples – Types

`nat`	a natural number, e.g., `0`
`nat => bool`	a predicate on nats, e.g., `even`
`nat => nat => nat`	a binary function on nats, e.g., `+`

Examples – Types

`nat`	a natural number, e.g., `0`
`nat => bool`	a predicate on nats, e.g., `even`
`nat => nat => nat`	a binary function on nats, e.g., `+`
`'a * 'b => 'a`	a polymorphic function on pairs, e.g., `fst`

Examples – Types

`nat`	a natural number, e.g., `0`
`nat => bool`	a predicate on nats, e.g., `even`
`nat => nat => nat`	a binary function on nats, e.g., `+`
`'a * 'b => 'a`	a polymorphic function on pairs, e.g., `fst`
`('a => 'b) => 'a list => 'b list`	a higher-order function on lists, e.g., `map`

Terms

t	$\stackrel{\text{def}}{=}$	`x`	constant or variable (identifier)
	$\mid$	$t\ t$	function application
	$\mid$	`%`x`.` t	lambda abstraction
	$\mid$	**if** t **then** t **else** t	if-clauses
	$\mid$	**let** x = t **in** t	let-bindings
	$\mid$	**case** t **of** p => t | ... | p => t	*case − expressions*
	$\mid$	...	lots of syntactic sugar

where p is a *pattern*

Terms

$$
\begin{array}{lcll}
t & \stackrel{\text{def}}{=} & \texttt{x} & \text{constant or variable (identifier)} \\
 & | & t\ t & \text{function application} \\
 & | & \texttt{\%}x\texttt{.}\ t & \text{lambda abstraction} \\
 & | & \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t & \text{if-clauses} \\
 & | & \textbf{let}\ x\ \texttt{=}\ t\ \textbf{in}\ t & \text{let-bindings} \\
 & | & \textbf{case}\ t\ \textbf{of}\ p\ \texttt{=>}\ t\ \texttt{|}\ \dots\ \texttt{|}\ p\ \texttt{=>}\ t & \textit{case} - \textit{expressions} \\
 & | & \dots & \text{lots of syntactic sugar}
\end{array}
$$

where p is a *pattern*

Remark

often necessary to put parentheses around lambda abstractions, if-clauses, let-bindings, and case-expressions; in order to get priorities right

Terms – Examples

`f x` — function `f` applied to value `x`

Terms – Examples

`f x`	function `f` applied to value `x`
`(%x. x + 1)`	the anonymous successor function

Terms – Examples

`f x`	function `f` applied to value `x`
`(%x. x + 1)`	the anonymous successor function
`let s = (%x. x + 1) in s 0`	application of successor to `0`

Terms – Examples

`f x`	function `f` applied to value `x`
`(%x. x + 1)`	the anonymous successor function
`let s = (%x. x + 1) in s 0`	application of successor to `0`
`(%p. case p of (x, y) => x)`	possible implementation of `fst`

Formulas (Terms of Type `bool`)

φ	$\stackrel{\text{def}}{=}$	`True` $\mid$ `False`	Boolean constants
	$\mid$	`~`φ	negation
	$\mid$	φ `=` φ	equality
	$\mid$	φ `&` φ $\mid$ φ `|` φ $\mid$ φ `-->` φ	binary operators
	$\mid$	`ALL` $x.\ \varphi$ $\mid$ `EX` $x.\ \varphi$	quantifiers

Formulas (Terms of Type `bool`)

φ	$\stackrel{\text{def}}{=}$	`True` $\mid$ `False`	Boolean constants
	$\mid$	`~`φ	negation
	$\mid$	φ `=` φ	equality
	$\mid$	φ `&` φ $\mid$ φ `|` φ $\mid$ φ `-->` φ	binary operators
	$\mid$	`ALL` $x.\ \varphi$ $\mid$ `EX` $x.\ \varphi$	quantifiers

Operator Precedence

`=` $\succ$ `~` $\succ$ `&` $\succ$ `|` $\succ$ `-->` $\succ$ `ALL`, `EX`

Formulas – Examples

`~A | A`	law of excluded middle

Formulas – Examples

`~A | A`	law of excluded middle
`False --> P`	anything follows from `False`

Formulas – Examples

`~A | A`	law of excluded middle
`False --> P`	anything follows from `False`
`a = b & b = c --> a = c`	transitivity of equality

Formulas – Examples

`~A | A`	law of excluded middle
`False --> P`	anything follows from `False`
`a = b & b = c --> a = c`	transitivity of equality
`(ALL x. P x) = (~(EX x. ~(P x)))`	variant of *De Morgan's Law*

Remark – Type Constraints

- $(t :: \tau)$ states that term t is of type τ
- in presence of overloaded constants and functions (like 0 and +), sometimes necessary to add constraints

Remark – Type Constraints

- $(t :: \tau)$ states that term t is of type τ
- in presence of overloaded constants and functions (like `0` and `+`), sometimes necessary to add constraints

Examples

Remark – Type Constraints

- $(t :: \tau)$ states that term t is of type τ
- in presence of overloaded constants and functions (like `0` and `+`), sometimes necessary to add constraints

Examples

- `(x::nat) + y`, since `+` has type `'a => 'a => 'a`

Remark – Type Constraints

- (t `::` τ) states that term t is of type τ
- in presence of overloaded constants and functions (like `0` and `+`), sometimes necessary to add constraints

Examples

- `(x::nat) + y`, since `+` has type `'a => 'a => 'a`
- `(0::nat) + y`, since `0` has type `'a`

Remark – Type Constraints

- (t :: τ) states that term t is of type τ
- in presence of overloaded constants and functions (like `0` and `+`), sometimes necessary to add constraints

Examples

- `(x::nat) + y`, since `+` has type `'a => 'a => 'a`
- `(0::nat) + y`, since `0` has type `'a`
- `Suc 0`, no constraint necessary since `Suc` has type `nat => nat`

Remark – 3 Kinds of Variables

- free variables (blue in jEdit/ProofGeneral)
- bound variables (green in jEdit/ProofGeneral)
- schematic variables (dark blue in jEdit/ProofGeneral; have leading ?); can be replaced by arbitrary values

Remark – 3 Kinds of Variables

- free variables (blue in jEdit/ProofGeneral)
- bound variables (green in jEdit/ProofGeneral)
- schematic variables (dark blue in jEdit/ProofGeneral; have leading ?); can be replaced by arbitrary values

Examples

Remark – 3 Kinds of Variables

- free variables (blue in jEdit/ProofGeneral)
- bound variables (green in jEdit/ProofGeneral)
- schematic variables (dark blue in jEdit/ProofGeneral; have leading ?); can be replaced by arbitrary values

Examples

- in '$x + y$', x and y are free

Remark – 3 Kinds of Variables

- free variables (blue in jEdit/ProofGeneral)
- bound variables (green in jEdit/ProofGeneral)
- schematic variables (dark blue in jEdit/ProofGeneral; have leading `?`); can be replaced by arbitrary values

Examples

- in '$x + y$', x and y are free
- in '`ALL` x. $P\ x$', x is bound and P is free

Remark – 3 Kinds of Variables

- free variables (blue in jEdit/ProofGeneral)
- bound variables (green in jEdit/ProofGeneral)
- schematic variables (dark blue in jEdit/ProofGeneral; have leading `?`); can be replaced by arbitrary values

Examples

- in '$x + y$', x and y are free
- in '`ALL` x. P x', x is bound and P is free
- in '(`~~`?P) = ?P', P is schematic

Functional Programming in HOL

An Introductory Theory – Session1.thy

```
theory Session1 imports Datatype begin
```

An Introductory Theory – Session1.thy

```
theory Session1 imports Datatype begin
```

A Datatype for Lists

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

An Introductory Theory – Session1.thy

```
theory Session1 imports Datatype begin
```

A Datatype for Lists

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

Remark – Inner and Outer Syntax

- terms and types are inner syntax
- inner syntax has to be put between double quotes (but: double quotes around single identifiers may be dropped)

An Introductory Theory – `Session1.thy`

```
theory Session1 imports Datatype begin
```

A Datatype for Lists

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

Remark – Inner and Outer Syntax

- terms and types are inner syntax
- inner syntax has to be put between double quotes (but: double quotes around single identifiers may be dropped)

Syntactic Sugar for Lists – `notation`

```
notation Nil  ("[]")
notation Cons (infixr "#" 65)
```

Syntactic Sugar for Lists – notation

```
notation Nil  ("[]")
notation Cons (infixr "#" 65)
```

Syntactic Sugar for Lists – inlined

```
datatype 'a list = Nil ("[]")
                 | Cons 'a "'a list" (infixr "#" 65)
```

Example Lists

Example Lists

`Nil` corresponds to `[] :: 'a list`

Example Lists

`Nil`	corresponds to `[] :: 'a list`
`Cons (0::nat) Nil`	corresponds to `[0] :: nat list`

Example Lists

`Nil`	corresponds to `[] :: 'a list`
`Cons (0::nat) Nil`	corresponds to `[0] :: nat list`
`Cons 0 (Cons 1 Nil)`	corresponds to `[0,1] :: 'a list`

Datatypes – The General Format

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_n)t = C_1\ \tau_{11}\ \ldots\ \tau_{1k_1} \mid \ldots \mid C_m\ \tau_{m1}\ \ldots\ \tau_{mk_m}$$

- α_i parameters
- C_j constructor names

Datatypes – The General Format

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_n)t = C_1\ \tau_{11}\ \ldots\ \tau_{1k_1} \mid \ldots \mid C_m\ \tau_{m1}\ \ldots\ \tau_{mk_m}$$

- α_i parameters
- C_j constructor names

Every Datatype has ...

- many lemmas proved automatically (e.g., `~([] = x#xs)` for lists)
- a size function `size :: t => nat`
- an induction scheme
- a case analysis scheme

Functions on Datatypes – Primitive Recursion

- primitive recursion over datatype t uses equations of the form

$$f\ x_1\ \ldots (C\ y_1\ \ldots\ y_k)\ \ldots\ x_n = b$$

- where C is constructor of t
- all calls to f in b have form $f\ \ldots\ y_i\ \ldots$ for some i

Functions on Datatypes – Primitive Recursion

- primitive recursion over datatype t uses equations of the form

$$f\ x_1\ \ldots (C\ y_1\ \ldots\ y_k)\ \ldots\ x_n = b$$

- where C is constructor of t
- all calls to f in b have form $f\ \ldots\ y_i\ \ldots$ for some i

Intuition

- every recursive call removes one constructor symbol
- hence f terminates

Example – Concatenating two Lists

```
primrec
  append :: "'a list => 'a list => 'a list"
  (infixr "@" 65)
where
  "[] @ ys = ys"
| "(x # xs) @ ys = x # (xs @ ys)"
```

Example – Reversing a List

```
primrec rev :: "'a list => 'a list" where
  "rev [] = []"
| "rev (x # xs) = rev xs @ (x # [])"
```

An Introductory Proof

```
"rev (rev xs) = xs"
```

An Introductory Proof

```
"rev (rev xs) = xs"
```

Proof

Whiteboard □

Some Diagnostic Commands

find_theorems ⟨*args*⟩	print all theorems matching ⟨*args*⟩
print_cases	print currently available cases
prop ⟨*formula*⟩	print proposition ⟨*formula*⟩
term ⟨*term*⟩	print term ⟨*term*⟩ and its type
thm ⟨*name*⟩	print theorem called ⟨*name*⟩
typ ⟨*type*⟩	print type ⟨*type*⟩
value ⟨*term*⟩	evaluate and print ⟨*term*⟩

General Structure of a Proof

$$
\begin{array}{rcl}
\textit{proof} & \stackrel{\text{def}}{=} & \texttt{proof}\ \textit{method}^?\ \textit{statement}^*\ \texttt{qed}\ \textit{method}^? \\
 & \mid & \texttt{by}\ \textit{method}\ \textit{method}^? \\
\\
\textit{statement} & \stackrel{\text{def}}{=} & \texttt{fix}\ \textit{variables} \\
 & \mid & \texttt{assume}\ \textit{proposition}^+ \\
 & \mid & (\texttt{from}\ \textit{fact}^+)^?\ (\texttt{show} \mid \texttt{have})\ \textit{proposition}\ \textit{proof} \\
\\
\textit{proposition} & \stackrel{\text{def}}{=} & (\textit{label}\,\texttt{:})^?\ \texttt{"}\textit{term}\texttt{"} \\
\\
\textit{fact} & \stackrel{\text{def}}{=} & \textit{label} \\
 & \mid & \texttt{`}\textit{term}\texttt{`}
\end{array}
$$

An Introductory Proof (cont'd)

```
lemma append_Nil2[simp]: "xs @ [] = xs"
  by (induct xs) simp_all

lemma append_assoc[simp]:
  "(xs @ ys) @ zs = xs @ (ys @ zs)"
  by (induct xs) simp_all

lemma rev_append[simp]:
  "rev (xs @ ys) = rev ys @ rev xs"
  by (induct xs) simp_all

theorem rev_rev_ident[simp]: "rev (rev xs) = xs"
  by (induct xs) simp_all
```

Basic Types – Natural Numbers

```
datatype nat = 0
             | Suc nat
```

Basic Types – Natural Numbers

```
datatype nat = 0
             | Suc nat
```

Predefined Operations

- addition, subtraction (`+`, `-`)
- multiplication, division (`*`, `div`)
- modulo (`mod`)
- minimum, maximum (`min`, `max`)
- less than (or equal) (`<`, `<=`)

Basic Types – Pairs

- `Pair :: 'a => 'b => 'a * 'b`
- `fst :: 'a * 'b => 'a`
- `snd :: 'a * 'b => 'b`
- `curry :: ('a * 'b => 'c) => 'a => 'b => 'c`
- `split :: ('a => 'b => 'c) => 'a * 'b => 'c`

Basic Types – Option

```
datatype 'a option = None
                   | Some 'a
```

Basic Types – Option

```
datatype 'a option = None
                   | Some 'a
```

Predefined Operations

- `the :: 'a option => 'a`
- `Option.set :: 'a option => 'a set`

Definitions – Type Synonyms

introducing new names for existing types

Definitions – Type Synonyms

introducing new names for existing types

Examples

```
type_synonym number    =  nat
type_synonym gate      =  "bool => bool => bool"
type_synonym 'a plist  =  "('a * 'a) list"
```

Definitions – Constant Definitions

introducing new names for existing expressions

Definitions – Constant Definitions

introducing new names for existing expressions

Examples

```
definition nand :: gate
where "nand A B == ~(A & B)"

definition xor :: gate
where "xor A B == (A & ~B) | (~A & B)"
```

Definitions – Constant Definitions

introducing new names for existing expressions

Examples

```
definition nand :: gate
where "nand A B == ~(A & B)"

definition xor :: gate
where "xor A B == (A & ~B) | (~A & B)"
```

Provided Lemmas

definition of constant ⟨*const*⟩ automatically provides lemma ⟨*const*⟩`_def`, stating equality between constant and its definition

The Definitional Approach

- only total functions are allowed . . .
- or else

```
axiomatization f :: "nat => nat" where
  f_def: "f x = f x + 1"

lemma everything: "P"
proof -
 fix x
 have "f x = f x + 1" by (rule f_def)
 from this show "P" by simp
qed

lemma wrong: "0 = 1" by (rule everything)
```

Exercises

1. define a primitive recursive function `length` that computes the length of a list
2. prove `"length (xs @ ys) = length xs + length ys"`
3. define a primitive recursive function `snoc` that appends an element at the end of a list (do not use `@`)
4. prove `"snoc (rev xs) x = rev (x # xs)"`
5. define a primitive recursive function `replace` such that `replace x y zs` replaces all occurrences of `x` in the list `zs` by `y`
6. prove
   `"replace x y (rev zs) = rev (replace x y zs)"`