

# Experiments in Verification

SS 2009


Christian Sternagel (VO)<sup>1</sup>

Computational Logic  
Institute of Computer Science  
University of Innsbruck

6 March 2009

---

<sup>1</sup>[christian.sternagel@uibk.ac.at](mailto:christian.sternagel@uibk.ac.at)

A large, faint watermark of the University of Innsbruck seal is visible in the bottom-left corner. The seal is circular and contains a central figure holding a staff and a book, surrounded by Latin text including 'SIGILLVM' and 'CESAREVM'.

# Lecture

## Facts

- ▶ Who? Christian Sternagel
- ▶ Where? RR 20
- ▶ LV-Nr. 703523
- ▶ VO 1
- ▶ <http://cl-informatik.uibk.ac.at/teaching/ss09/eve/>
- ▶ office hours: Friday 15:00 – 17:00 in 3M09
- ▶ grading: project

# Schedule

## Sessions

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

1. 06 March 2009
2. 13 March 2009
3. 20 March 2009
4. 27 March 2009

# The Project

## Procedure

- ▶ after last session (on March 27) 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

# This Time

## Session 1

formal verification, Isabelle/HOL basics, functional programming in HOL

## Session 2

simplification, function definitions, induction, calculational reasoning

## Session 3

natural deduction, propositional logic, predicate logic

## Session 4

sets, relations, inductively defined sets, advanced topics

# What is Verification?

## Answers

- ▶ 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.*

# What Methods Do Exist?

## 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*

# Example

## Problem

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

## Truth Table (Model-Theoretic)

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



# Example

## Problem

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	$\perp$	$\neg$ e 3, 4

# What Methods Do Exist?

## 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*

# System Architecture

<b>Proof General</b>	Emacs based interface
<b>Isabelle/HOL</b>	Higher-Order Logic
<b>Isabelle</b>	generic theorem prover
<b>Standard ML</b>	implementation language

# Higher-Order Logic

HOL is

Functional Programming + Logic

HOL has

- ▶ datatypes (**datatype**)
- ▶ recursive functions (**fun**)
- ▶ logical operators ( $\wedge$ ,  $\vee$ ,  $\longrightarrow$ ,  $\forall$ ,  $\exists$ , ...)

# The Isabelle System

## Setup

- ▶ custom settings in file `~/isabelle/etc/settings`
- ▶ you will need at least:  
`ISABELLE_DOC_FORMAT=pdf`  
`PDF_VIEWER=<program>`

## Main Components

- ▶ `isatool`: mainly for documentation (i.e., `isatool doc`)
- ▶ Isabelle: interactive proof development in ProofGeneral (i.e.,  
`$ Isabelle <File>.thy`)

# Proof General

## Useful Shortcuts

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

## Theory Files (\*.thy)

### General Structure

```
theory Name imports  $T_1 \dots T_n$  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

## Definition

$\tau$	$\stackrel{\text{def}}{=} \text{bool} \mid \text{nat} \mid \dots$	base types
	$\mid 'a \mid 'b \mid \dots$	type variables
	$\mid \tau \Rightarrow \tau$	total functions
	$\mid \tau * \tau$	pairs
	$\mid \tau \text{ list}$	lists
	$\mid \dots$	user-defined types

## Remark (Function Type is Right-Associative)

$$\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \quad \equiv \quad \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$$



# Types – Examples

`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

## Definition

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

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

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

# Formulas (Terms of Type bool)

## Definition

$\varphi$	$\stackrel{\text{def}}{=} \text{True} \mid \text{False}$	Boolean constants
	$\mid \sim \varphi$	negation
	$\mid \varphi = \varphi$	equality
	$\mid \varphi \& \varphi \mid \varphi \mid \varphi \mid \varphi \dashrightarrow \varphi$	binary operators
	$\mid \text{ALL } x. \varphi \mid \text{EX } x. \varphi$	quantifiers

## Operator Priorities

$= \ \ \ \ \sim \ \ \ \ \& \ \ \ \ \mid \ \ \ \ \dashrightarrow \ \ \ \ \text{ALL, EX}$

# Formulas – Examples

$\sim A \mid A$

False  $\rightarrow$  P

$a = b \ \& \ b = c \rightarrow a = c$

$(\text{ALL } x. P \ x) = (\sim(\text{EX } x. \sim(P \ x)))$

law of excluded middle

anything follows from False

transitivity of equality

variant of *De Morgan's Law*

# Remarks

## Type Constraints

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

## 3 Kinds of Variables

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

# Examples

## Type Constraints

- ▶  $(x :: \text{nat}) + y$ , since  $+$  has type  $'a \Rightarrow 'a \Rightarrow 'a$
- ▶  $(0 :: \text{nat}) + y$ , since  $0$  has type  $'a$
- ▶  $\text{Suc } 0$ , no constraint necessary since  $\text{Suc}$  has type  $\text{nat} \Rightarrow \text{nat}$

## 3 Kinds of Variables

- ▶ in  $'x + y'$ ,  $x$  and  $y$  are free
- ▶ in  $'\text{ALL } x. P \ x'$ ,  $x$  is bound and  $P$  is free
- ▶ in  $'(\sim\sim?P) = ?P'$ ,  $P$  is schematic

# An Introductory Theory – Session1.thy

## Opening

```
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



# Example

## Lists

<code>Nil</code>	corresponds to	<code>[] :: 'a list</code>
<code>Cons (0::nat) Nil</code>	corresponds to	<code>[0] :: nat list</code>
<code>Cons 0 (Cons 1 Nil)</code>	corresponds to	<code>[0,1] :: 'a list</code>

# Datatypes

## The General Format

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

- ▶  $\alpha_i$  parameters
- ▶  $C_j$  constructor names

## Every Datatype Has ...

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

# Syntactic Sugar for Lists

Via notation ...

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

...or Inline

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

# Functions on Datatypes

## Primitive Recursion

over datatype  $t$  uses equations of the form

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

where

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

## Intuition

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

## Example – Two Functions on Lists

### Concatenating Two Lists

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

### Reversing a List

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

# An Introductory Proof

## Theorem

`"rev(rev xs) = xs"`

## Proof.

Whiteboard



# Some Helpful Commands

<b>find_theorems</b> $\langle args \rangle$	find all theorems matching $\langle args \rangle$
<b>normal_form</b> $\langle term \rangle$	simplify $\langle term \rangle$
<b>print_cases</b>	show currently available cases
<b>prop</b> $\langle formula \rangle$	show proposition $\langle formula \rangle$
<b>term</b> $\langle term \rangle$	show term $\langle term \rangle$ and its type
<b>thm</b> $\langle name \rangle$	show theorem called $\langle name \rangle$
<b>typ</b> $\langle type \rangle$	show type $\langle type \rangle$
<b>value</b> $\langle term \rangle$	execute $\langle term \rangle$

# General Structure of a Proof

*proof*  $\stackrel{\text{def}}{=} \mathbf{proof} \text{ method}^? \text{ statement}^* \mathbf{qed} \text{ method}^?$   
 |  $\mathbf{by} \text{ method} \text{ method}^?$

*statement*  $\stackrel{\text{def}}{=} \mathbf{fix} \text{ variables}$   
 |  $\mathbf{assume} \text{ proposition}^+$   
 |  $(\mathbf{from} \text{ fact}^+)^? (\mathbf{show} \mid \mathbf{have}) \text{ proposition} \text{ proof}$

*proposition*  $\stackrel{\text{def}}{=} (\text{label}:)^? \text{"term"}$

*fact*  $\stackrel{\text{def}}{=} \text{label}$   
 |  $\text{'term'}$



# An Introductory Proof (cont'd)

## Isabelle-Proof

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

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

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

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

# Basic Types – Natural Numbers

## Definition

```
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

## Predefined Operations

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

# Basic Types – Option

## Definition

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

## Predefined Operations

- ▶ `the :: 'a option => 'a`
- ▶ `o2s :: 'a option => 'a set`

# Definitions – Type Synonyms

## Example

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

## Definitions – Constant Definitions

### Example

```
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  $\langle const \rangle$  automatically provides lemma  $\langle const \rangle\_def$ , stating equality between constant and its definition

# The Definitional Approach

Only Total Functions Are Allowed ...

or else ...

```
axioms f: "f x = f x + (1::nat)"
```

```
lemma everything: "P"
```

```
proof -
```

```
  fix f x
```

```
  have "f x = f x + (1::nat)" by (rule f)
```

```
  from this show "P" by simp
```

```
qed
```

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

## Exercises

### length

- ▶ define a primitive recursive function `length` that computes the length of a list
- ▶ prove `"length(xs@ys) = length xs + length ys"`

### snoc

- ▶ define a primitive recursive function `snoc` that appends an element at the end of a list (do not use `@`)
- ▶ prove `"rev(x#xs) = snoc (rev xs) x"`

### replace

- ▶ define a primitive recursive function `replace` such that `replace x y zs` replaces all occurrences of `x` in the list `zs` by `y`
- ▶ prove `"rev(replace x y zs) = replace x y (rev zs)"`