

Experiments in Verification

SS 2010

Christian Sternagel (VO)¹

Computational Logic
Institute of Computer Science
University of Innsbruck

12 March 2010

¹christian.sternagel@uibk.ac.at

Schedule

Sessions

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

1. 12 March 2010
2. 19 March 2010
3. 26 March 2010
4. 16 April 2010

Lecture

Facts

- ▶ Who? Christian Sternagel
- ▶ Where? RR 21
- ▶ LV-Nr. 703523
- ▶ VO 1
- ▶ <http://cl-informatik.uibk.ac.at/teaching/ss10/eve/>
- ▶ office hours: Friday 15:00 – 17:00 in 3N01
- ▶ grading: project

The Project

Procedure

- ▶ after last session (on April 16) 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 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

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.

Example

Problem

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

Truth Table (Model-Theoretic)

A	B	$\neg A$	$B \longrightarrow A$	Φ
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

System Architecture

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

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

Higher-Order Logic

HOL is

Functional Programming + Logic

HOL has

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

The Isabelle System

Setup

- ▶ custom settings in file `~/.isabelle/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`)

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
```

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

Types

Definition

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

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

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) => x)</code>	possible implementation of <code>fst</code>

Terms

Definition

$t \stackrel{\text{def}}{=} x$	constant or variable (identifier)
$t t$	function application
$\%x. t$	lambda abstraction
<code>if t then t else t</code>	if-clauses
<code>let x = t in t</code>	let-bindings
<code>case t of p => t ... p => t</code>	case – expressions
...	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

Formulas (Terms of Type bool)

Definition

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

Operator Priorities

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

Formulas – Examples

$\sim A \mid A$	law of excluded middle
$\text{False} \rightarrow P$	anything follows from False
$a = b \ \& \ b = c \rightarrow a = c$	transitivity of equality
$(\text{ALL } x. P \ x) = (\sim(\text{EX } x. \sim(P \ x)))$	variant of <i>De Morgan's Law</i>

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 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

Remarks

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

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

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

`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

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

- ▶ α_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 – Functions on Lists

Concatenating Two Lists

```

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

```

Example – Functions on Lists (cont'd)

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

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

General Structure of a Proof

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

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

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

$fact \stackrel{\text{def}}{=} \text{label}$
 | '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`
- ▶ `Option.set :: 'a option => 'a set`

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

Definitions – Type Synonyms

Example

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

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)"`