Experiments in Verification

SS 2009

Christian Sternagel (VO)[1]

Computational Logic
Institute of Computer Science
University of Innsbruck

6 March 2009

[1] christian.sternagel@uibk.ac.at

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	$\neg A$	$B \longrightarrow A$	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	$\bot$	$\neg$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

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

`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** $T_1 \ldots T_n$ **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

Definition

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

Remark (Function Type is Right-Associative)

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

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)
	$\mid$	$t\ t$	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*

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`
`(%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`)

Definition

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

Operator Priorities

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

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*

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

Examples

Type Constraints

- `(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`

3 Kinds of Variables

- 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

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

- 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 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

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

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 – 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

find_theorems ⟨*args*⟩	find all theorems matching ⟨*args*⟩
normal_form ⟨*term*⟩	simplify ⟨*term*⟩
print_cases	show currently available cases
prop ⟨*formula*⟩	show proposition ⟨*formula*⟩
term ⟨*term*⟩	show term ⟨*term*⟩ and its type
thm ⟨*name*⟩	show theorem called ⟨*name*⟩
typ ⟨*type*⟩	show type ⟨*type*⟩
value ⟨*term*⟩	execute ⟨*term*⟩

General Structure of a Proof

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

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 ⟨*const*⟩ automatically provides lemma ⟨*const*⟩_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)"`