

Experiments in Verification

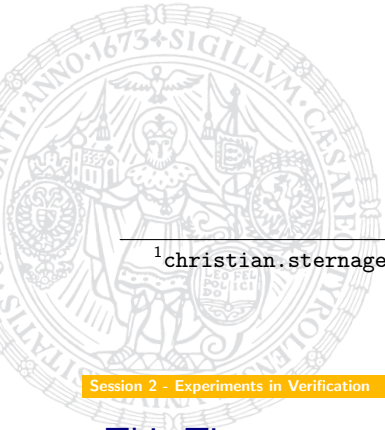
SS 2009

Christian Sternagel (VO)¹

Computational Logic
Institute of Computer Science
University of Innsbruck

13 March 2009

¹christian.sternagel@uibk.ac.at



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

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)`"

Term Rewriting

Example (Addition and Multiplication on Natural Numbers)

- ▶ a set of rules, also called a term rewrite system (TRS)

$$\begin{array}{ll} 0 + y \rightarrow y & 0 \times y \rightarrow 0 \\ s(x) + y \rightarrow s(x + y) & s(x) \times y \rightarrow y + (x \times y) \end{array}$$

- ▶ 'compute' 1×2

$$\begin{aligned} s(0) \times s^2(0) &\rightarrow s^2(0) + (0 \times s^2(0)) \\ &\rightarrow s^2(0) + 0 \\ &\rightarrow s(s(0) + 0) \\ &\rightarrow s(s(0 + 0)) \\ &\rightarrow s^2(0) \end{aligned}$$

In Isabelle

```
datatype num = Zero | Succ num
```

```
notation Zero ("0")
```

```
notation Succ ("s'(_')")
```

```
primrec add (infixl "+" 65)
```

```
where "0::num + y = y"
      | "s(x) + y = s(x + y)"
```

```
primrec mul (infixl "×" 70)
```

```
where "0::num × y = 0"
      | "s(x) × y = y + (x × y)"
```

X-Symbols

ASCII	X-Symbol	shown as	ASCII	X-Symbol	shown as
=>	\<Rightarrow>	⇒	ALL	\<forall>	∀
-->	\<longrightarrow>	→	EX	\<exists>	∃
==>	\<Longrightarrow>	⇒	&	\<and>	∧
!!	\<And>	∧		\<or>	∨
==	\<equiv>	≡	~	\<not>	¬
~=	\<noteq>	≠	%	\<lambd>	λ
:	\<in>	∈	*	\<times>	×
~:	\<notin>	∉	o	\<circ>	◦
Un	\<union>	∪	[\<lbrakk>	⌊
Int	\<inter>	∩]	\<rbrakk>	⌋
Union	\<Union>	∪	<=	\<subseteq>	⊆
Inter	\<Inter>	∩	<	\<subset>	⊂

- ▶ activate via Proof-General → Options → X-Symbol

Explanatory Notes

- ▶ 0 is overloaded, hence we need **type constraints**
- ▶ use ' within **syntax annotations** to escape characters with special meaning, e.g., '(for an opening parenthesis (special meaning: start a group for pretty printing) or '_ for an underscore (special meaning: argument placeholder)
- ▶ you may omit the type of a function if it can be inferred automatically
- ▶ to get symbols like × use **X-Symbols** (see next slide)
- ▶ you automatically get lemmas num.simps, add.simps, and mul.simps

Using Simplification Rules

Automatically

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))" by simp
```

Explicitly (unfolding)

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))"
  unfolding add.simps mul.simps by (rule refl)
```

Modifying the *Simpset*

- ▶ **simpset** is set of simplification rules currently in use
- ▶ adding a lemma to the simpset
declare \langle theorem-name \rangle [simp]
- ▶ deleting a lemma from the simpset
declare \langle theorem-name \rangle [simp del]

Example

```
declare add.simps[simp del]
lemma "0 + s(0) = s(0)"
```

A Proof by Hand

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))"
proof -
  have "s(s(0)) × s(s(0)) =
    s(s(0)) + s(0) × s(s(0))"
    unfolding mul.simps by (rule refl)
  from this have "s(s(0)) × s(s(0)) =
    s(s(0)) + (s(s(0)) + 0 × s(s(0)))"
    unfolding mul.simps .
  from this have "s(s(0)) × s(s(0)) =
    s(s(0)) + (s(s(0)) + 0)"
    unfolding mul.simps .
  from this show ?thesis unfolding add.simps .
qed
```

A More Complete Grammar for Proofs

$$\text{proof} \stackrel{\text{def}}{=} \begin{array}{l} \text{prefix}^* \text{proof method}^? \text{statement}^* \text{qed method}^? \\ | \\ \text{prefix}^* \text{by method method}^? \end{array}$$

$$\text{prefix} \stackrel{\text{def}}{=} \begin{array}{l} \text{apply method} \\ | \\ \text{using fact}^* \\ | \\ \text{unfolding fact}^* \end{array}$$

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

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

$$\text{fact} \stackrel{\text{def}}{=} \begin{array}{l} \text{label} \\ | \\ \text{'term'}$$

The simp Method

General Format

simp \langle list of modifiers \rangle

Modifiers

- ▶ add: \langle list of theorem names \rangle
- ▶ del: \langle list of theorem names \rangle
- ▶ only: \langle list of theorem names \rangle

Example

```
lemma "s(0) × s(0) = s(0)"
by (simp only: add.simps mul.simps)
```

A General Format for Stating Theorems

```

theorem  $\stackrel{\text{def}}{=} \textit{kind goal}$ 
  | kind name : goal
  | kind [attributes] : goal
  | kind name [attributes] : goal

kind  $\stackrel{\text{def}}{=} \text{theorem} \mid \text{lemma} \mid \text{corollary}$ 

goal  $\stackrel{\text{def}}{=} (\text{fixes variables})^? (\text{assumes prop}^+)^? \text{shows prop}^+$ 
  | prop+

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

```

Assumptions

- ▶ by default assumptions are used as simplification rules + assumptions are simplified themselves

```

lemma
  assumes "xs@zs = ys@xs" and "[]@xs = []@[]"
  shows "ys = zs"
using assms by simp

```

- ▶ this can lead to nontermination

```

lemma
  assumes "\x. f x = g(f(g x))"
  shows "f [] = f [] @ []"
using assms by simp

```

Example

```

lemma some_lemma[simp]:
  fixes A :: "bool" (* 'A' has type 'bool' *)
  assumes AnA: "A ^ A" (* give this fact the name 'AnA' *)
  shows "A"
using AnA by simp

```

The simp Method (cont'd)

More Modifiers

- ▶ (no_asm) assumptions are ignored
- ▶ (no_asm_simp) assumptions are not simplified themselves
- ▶ (no_asm_use) assumptions are simplified but not added to simpset

Tracing

- ▶ set Isabelle → Settings → Trace Simplifier
- ▶ useful to get a feeling for simplification rules
- ▶ see which rules are applied
- ▶ find out why simplification loops

Example

```

fun fib :: "nat => nat"
where "fib 0          = Suc 0"
      | "fib(Suc 0)    = Suc 0"
      | "fib(Suc(Suc n)) = fib n + fib(Suc n)"

```

Lemma

```
0 < fib n
```

Digression – Finding Theorems

Start Search

- ▶ either by keyboard shortcut Ctrl+C, Ctrl+F, or
- ▶ clicking the find-icon (a magnifying glass)

Search Criteria

- ▶ a number in parenthesis specifies how menu results should be shown
- ▶ a pattern in double quotes specifies the term to be searched for
- ▶ a pattern may contain wild cards '_', and type constraints
- ▶ precede a pattern by simp: to only search for theorems that could simplify the specified term at the root
- ▶ to search for part of a name use name: "<some string>"
- ▶ negate a search criterion by prefixing a minus, e.g., -name:

Abbreviations

- ▶ **this**: the previous proposition proved or assumed
- ▶ **then: from** this
- ▶ **hence: then have**
- ▶ **thus: then show**
- ▶ **with** <facts>: **from** <facts> this

The Command fun

Some Notes

- ▶ in principle arbitrary pattern matching on lhs
- ▶ patterns are matched top to bottom
- ▶ **fun** tries to prove termination automatically (current method: lexicographic orders)
- ▶ use **function** instead of **fun** to provide a manual termination prove
- ▶ for further information: `isatool doc functions`

Additional Commands

- ▶ **also**: to apply transitivity automatically
- ▶ **finally**: to reconsider first lhs
- ▶ **...**: to abbreviate previous rhs

An Example Proof (Base Case)

```

primrec sum :: nat => nat
where "sum 0      = 0"
      | "sum(Suc n) = Suc n + sum n"

lemma "sum n = (n*(Suc n)) div (Suc(Suc 0))"
proof (induct n)
  case 0 show ?case by simp
next

```

An Example Proof (Step Case)

```

case (Suc n)
hence IH: "sum n = (n*(Suc n)) div (Suc(Suc 0))" .
have "sum(Suc n) = Suc n + sum n" by simp
also have "... = Suc n + ((n*(Suc n)) div (Suc(Suc 0)))"
  unfolding IH by simp
also have "... = ((Suc(Suc 0)*Suc n) div Suc(Suc 0)) +
  ((n*(Suc n)) div Suc(Suc 0))" by arith
also have "... = (Suc(Suc 0)*Suc n + n*(Suc n)) div
  Suc(Suc 0)" by arith
also have "... = ((Suc(Suc 0) + n)*Suc n) div Suc(Suc 0)"
  unfolding add_mult_distrib by simp
also have "... = (Suc(Suc n) * Suc n) div Suc(Suc 0)"
  by simp
finally show ?case by simp
qed

```

An Example Proof (Notes)

- ▶ cases are named by the corresponding **datatype** constructors
- ▶ **?case** is an abbreviation installed for the current goal in each case of an induction proof
- ▶ **case** 0 sets up the assumption corresponding to the base case (i.e., none)
- ▶ **case** (Suc n) sets up the corresponding assumption
 - `fix n assume "sum n = (n*Suc n) div Suc(Suc 0)`
- ▶ arith is a decision procedure for **Presburger Arithmetic**
- ▶ . abbreviates **by** assumption

Exercises

<http://isabelle.in.tum.de/exercises/arith/powSum/ex.pdf>