Experiments in Verification

SS 2010

Christian Sternagel (VO)[1]

Computational Logic
Institute of Computer Science
University of Innsbruck

19 March 2010

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

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)"`

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

Term Rewriting

Example (Addition and Multiplication on Natural Numbers)

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

$$0 + y \to y \qquad\qquad 0 \times y \to 0$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad\qquad \mathsf{s}(x) \times y \to y + (x \times y)$$

- 'compute' 1×2

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

In Isabelle

```
datatype num = Zero | Succ num

notation Zero ("0")
notation Succ ("s'(_')")

primrec add :: "num ⇒ num ⇒ num" (infixl "+" 65)
where
  "(0::num) + y = y" |
  "s(x)     + y = s(x + y)"

primrec mul :: "num ⇒ num ⇒ num" (infixl "×" 70)
where
  "(0::num) × y = 0" |
  "s(x)     × y = y + (x × y)"
```

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 $\times$ use Unicode Tokens (see next slide)
- you automatically get lemmas `num.simps`, `add.simps`, and `mul.simps`

Unicode Tokens

<table>
<tr><th>ASCII</th><th>Unicode Token</th><th>shown as</th><th>ASCII</th><th>Unicode Token</th><th>shown as</th></tr>
<tr><td>=></td><td>\<Rightarrow></td><td>⇒</td><td>ALL</td><td>\<forall></td><td>∀</td></tr>
<tr><td>--></td><td>\<longrightarrow></td><td>⟶</td><td>EX</td><td>\<exists></td><td>∃</td></tr>
<tr><td>==></td><td>\<Longrightarrow></td><td>⟹</td><td>&</td><td>\<and></td><td>∧</td></tr>
<tr><td>!!</td><td>\<And></td><td>⋀</td><td>|</td><td>\<or></td><td>∨</td></tr>
<tr><td>==</td><td>\<equiv></td><td>≡</td><td>~</td><td>\<not></td><td>¬</td></tr>
<tr><td>~=</td><td>\<noteq></td><td>≠</td><td>%</td><td>\<lambda></td><td>λ</td></tr>
<tr><td>:</td><td>\<in></td><td>∈</td><td>*</td><td>\<times></td><td>×</td></tr>
<tr><td>~:</td><td>\<notin></td><td>∉</td><td>o</td><td>\<circ></td><td>∘</td></tr>
<tr><td>Un</td><td>\<union></td><td>∪</td><td>[|</td><td>\<lbrakk></td><td>⟦</td></tr>
<tr><td>Int</td><td>\<inter></td><td>∩</td><td>|]</td><td>\<rbrakk></td><td>⟧</td></tr>
<tr><td>Union</td><td>\<Union></td><td>⋃</td><td><=</td><td>\<subseteq></td><td>⊆</td></tr>
<tr><td>Inter</td><td>\<Inter></td><td>⋂</td><td><</td><td>\<subset></td><td>⊂</td></tr>
</table>

- activate via Proof-General → Options → Unicode Tokens

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 ⟨*theorem-name*⟩[simp]
- deleting a lemma from the simpset
  declare ⟨*theorem-name*⟩[simp del]

Example

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

A More Complete Grammar for Proofs

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

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
```

The `simp` Method

General Format

`simp` ⟨*list of modifiers*⟩

Modifiers

- `add:` ⟨*list of theorem names*⟩
- `del:` ⟨*list of theorem names*⟩
- `only:` ⟨*list of theorem names*⟩

Example

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

A General Format for Stating Theorems

$$
\begin{array}{rcl}
\mathit{theorem} & \stackrel{\text{def}}{=} & \mathit{kind}\ \mathit{goal} \\
& | & \mathit{kind}\ \mathit{name}:\ \mathit{goal} \\
& | & \mathit{kind}\ [\mathit{attributes}]:\ \mathit{goal} \\
& | & \mathit{kind}\ \mathit{name}[\mathit{attributes}]:\ \mathit{goal}
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathit{goal} & \stackrel{\text{def}}{=} & (\textbf{fixes}\ \mathit{variables})^{?}\ (\textbf{assumes}\ \mathit{prop}^{+})^{?}\ \textbf{shows}\ \mathit{prop}^{+} \\
& | & \mathit{prop}^{+}
\end{array}
$$

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

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
```

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
```

The `simp` Method (cont'd)

More Modifiers

- `(no_asm)` assumptions are ignored
- `(no_asm_simps)` 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

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:`

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`

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 left-hand sides
- 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: `isabelle doc functions`

Additional Commands

- **`also`**: to apply transitivity automatically
- **`finally`**: to reconsider first left-hand side
- ...: to abbreviate previous right-hand side

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`