Getting started with Isabelle/Isar
Exercises

Makarius Wenzel
TU München

August 2007

Contents

1 Summation Formulae

1.1 Polynomial sums

▷ *Produce structured proofs of the following theorems, using induction and calculational reasoning in Isar.*

Note that existing tactic scripts are of limited use in reconstructing structured proofs; nevertheless the hints of automated steps below can be re-used to finish trivial sub-problems. The $\sum$ symbol can be entered as "`\<Sum>`"; recall that numerals in Isabelle/HOL are polymorphic.

```
theorem
  fixes n :: nat
  shows "2 * (∑i=0..n. i) = n * (n + 1)"
  by (induct n) simp_all
```

```
theorem
  fixes n :: nat
  shows "(∑i=0..<n. 2 * i + 1) = n²"
  by (induct n) (simp_all add: power_eq_if nat_distrib)
```

```
theorem
  fixes n :: nat
  shows "(∑i=0..<n. 2^i) = 2^n - (1::nat)"
  by (induct n) (simp_all split: nat_diff_split)
```

```
theorem
  fixes n :: nat
  shows "2 * (∑i=0..<n. 3^i) = 3^n - (1::nat)"
  by (induct n) (simp_all add: nat_distrib)
```

```
theorem
  fixes n :: nat
  assumes "0 < k"
  shows "(k - 1) * (∑i=0..<n. k^i) = k^n - (1::nat)"
  by (induct n) (insert ‘0 < k‘, simp_all add: nat_distrib)
```

▷ *Try explicit statements vs. term abbreviations: implicit "...",* `?case`, `?thesis` *as well as explicit* **is**/**let** *pattern matching.*

▷ *Try explicit* **fix**/**assume** *vs. implicit* **case** *abbreviations.*

▷ *Which facts are relevant to solve local problems automatically?*

1.2 Division and divisibility

The following statements are more conventional, using explicit division on the RHS.

```
theorem
  fixes n :: nat
  shows "(∑i=0..n. i) = n * (n + 1) div 2"
```

Here we state divisibility in the result expression explicitly:

```
theorem
  fixes n :: nat
  shows "∃k. n * (n + 1) = 2 * k"
```

▷ *Re-use your structured proof texts from above, but not the theorems.*

2 Porting tactic scripts

▷ *Turn your Isabelle tactic scripts from yesterday's exercises into Isar proof texts. Observe the following hints on producing "proper Isar" by avoiding certain tactical features of the input language of Isabelle/Isar.*

- **apply** and **done** as the main constituents of unstructured tactic scripts need to be replaced by explicit proof structure. Note that typical two-step proofs of the *decompose–finish* form (such as `induct`–`auto` or `cases`–`auto` or `rule`–`auto`) may be turned into proper Isar using **by** with two methods: "**by** `initial_method terminal_method`".

- **prefer** and **defer**, as well as any goal addressing within tactic expressions should be replaced by properly laid out sub-proofs. Note that Isar is tolerant wrt. the order of sub-goals tackled in multiple **fix**–**assume**–**show** patterns.

- Methods ending with `_tac` (e.g. `rule_tac`, `induct_tac`) refer to tactic emulations that are inappropriate in structured Isar texts, because they allows to address sub-goals numerically, or refer to hidden parts of a sub-goal in the visible text (via instantiation) etc.

  Proper methods `rule`, `cases`, and `induct` are able to replace more detailed tactic specifications, because of the richer contextual information available in Isar proofs (with explicit statements, indication of previous facts via **then**, **from**, **using** etc.).

- There is no need to present auxiliary results in "normal form" of certain automated tools. In proper Isar, intermediate results may be easily inserted into the course of reasoning using **have** or **obtain**, expressed in the most natural form of the problem at hand. Then automated tools can deal with normalization and finishing in terminal proof steps: "**by** `auto`" or "**by** `simp`" etc.

- Avoid backwards reasoning with transitivity rules, but express single-step calculations in forward-style via **also** and **finally**.

- Avoid explicit instantiations of rules, but state the fully instantiated propositions as intermediate results as required.

3 Context-Free Grammars

This exercise is concerned with context-free grammars (CFGs) being defined as inductive sets (see also section 7.4 in the Isabelle/HOL tutorial, and 4.5 of http://isabelle.in.tum.de/exercises/).

3.1 Two grammars

The most natural definition of valid sequences of parentheses is this:

$$S \quad \rightarrow \quad \epsilon \quad | \quad '(' \; S \; ')' \quad | \quad S \; S$$

where ϵ is the empty word.
A second, somewhat unusual grammar is the following one:

$$T \quad \rightarrow \quad \epsilon \quad | \quad T \; '(' \; T \; ')'$$

▷ *Model both grammars as inductive sets* `S` *and* `T` *and prove* `S = T`.

The alphabet:

```
datatype alpha = A | B
```

Standard grammar:

```
consts S :: "alpha list set"

inductive S
intros
  S1: "[] ∈ S"
  S2: "w ∈ S ⟹ [A] @ w @ [B] ∈ S"
  S3: "v ∈ S ⟹ w ∈ S ⟹ v @ w ∈ S"
```

Nonstandard grammar:

```
consts T :: "alpha list set"

inductive T
intros
  T1: "[] ∈ T"
  T23: "v ∈ T ⟹ w ∈ T ⟹ v @ ([A] @ w @ [B]) ∈ T"
```

3.2 Equivalence proof

```
lemma T_in_S:
  assumes "w ∈ T"
  shows "w ∈ S"

lemma S_in_T:
  assumes "w ∈ S"
  shows "w ∈ T"

theorem "S = T"
```

4 Compilation with Side Effects

This exercise extends the compiler example in Section 3.3 of the Isabelle/HOL tutorial: expressions may have side effects; see also 6.3 of http://isabelle.in.tum.de/exercises/.

4.1 Expressions

▷ *Complete the subsequent definitions of expressions and evaluation within an environment of variable assignments.*

```
types 'v binop = "'v ⇒ 'v ⇒ 'v"

datatype ('a, 'v) exp =
    Const 'v
  | Var 'a
  | Binop "'v binop"  "('a, 'v) exp"  "('a, 'v) exp"
  | Assign 'a "('a, 'v) exp"

consts
  val :: "('a, 'v) exp ⇒ ('a ⇒ 'v) ⇒ 'v × ('a ⇒ 'v)"
primrec
  "val (Const c) env = (c, env)"
  "val (Var x) env = (env x, env)"
  "val (Binop f e1 e2) env =
     (let (x, env1) = val e1 env;
          (y, env2) = val e2 env1
      in (f x y, env2))"
  "val (Assign a e) env =
     (let (x, env') = val e env
      in (x, env' (a := x)))"
```

Pure expressions are exactly those without syntactical occurrence of assignment.

```
consts
  pure :: "('a, 'v) exp ⇒ bool"
```

▷ *Produce a meaningful structured proof that evaluation of pure expressions does not change the environment.* (Technically, this is a trivial induction.)

```
theorem "pure e ⟹ snd (val e env) = env"
```

4.2 Machine instructions

▷ *Observe the subsequent definitions of machine instructions and execution of instructions in an environment.*

```
datatype ('a, 'v) instr =
    CLoad 'v
  | VLoad 'a
  | Store 'a
  | Apply "'v binop"

consts
  exec :: "('a, 'v) instr list ⇒ 'v list ⇒ ('a ⇒ 'v) ⇒
    'v list × ('a ⇒ 'v)"
primrec
  "exec [] vs hp = (vs, hp)"
  "exec (i # is) vs hp =
    (case i of
      CLoad v ⇒ exec is (v # vs) hp
    | VLoad a ⇒ exec is (hp a # vs) hp
    | Store a ⇒ exec is vs (hp (a:= hd vs))
    | Apply f ⇒ exec is (f (hd (tl vs)) (hd vs) # tl (tl vs)) hp)"

lemma
  "exec [CLoad (3::nat),
         VLoad x,
         CLoad 4,
         Apply (op *),
         Apply (op +)]
        [] (λx. 0) = ([3], λx. 0)"
  by simp
```

4.3 Compilation

▷ *Complete the definition of compilation of expressions. Produce a structured proof for the main correctness statement.*

```
consts
  compile :: "('a, 'v) exp ⇒ ('a, 'v) instr list"

theorem correctness:
    "exec (compile e) [] s = ([fst (val e s)], snd (val e s))"
```