Isabelle Tutorial:

System, HOL and Proofs

Burkhart Wolff
(with contributions by Makarius Wenzel)

Université Paris-Sud

What we will talk about

What we will talk about

Isabelle with:

- Elementary Forward Proofs
- Tactic Proofs (“apply style”)
- Proof Contexts and Structured Proof

The Syntactic Category <proof>

- Notations for proofs so far:
  - ellipses:

    sorry, oops
  - "one-liners" simp and auto:

    by(<method>)   (abbrev: apply(...) done)
  - "apply-style proofs", backward-proofs:

    apply(<method>) … apply(<method>)

    done <method>
  - structured proofs:

    proof (<method>) … qed

The Syntactic Category <proof>

- Notations for proofs so far:
  - ellipses:

    sorry, oops
  - "one-liners" simp and auto:

    by(<method>) (abbrev: apply(...) done)
  - "apply-style proofs", backward-proofs:

    apply(<method>) … apply(<method>)

    done <method>
  - structured proofs:

    proof (<method>) … qed

Introduction to structured proofs in Isabelle/HOL

Simple Proof Commands (Recall)

- Simple (Backward) Proofs:

```
lemma  <thmname> :
  [ <contextelem>+ shows ]"<φ>"
  <proof>
```

  - where <contextelem> declare elements of a proof context Γ (to be discussed further)
  - where <proof> is just a call of a high-level proof method by(simp), by(auto), by(metis), by(arith) or the discharger sorry (for the moment).

How to Declare Structured Goals (Recall)

- (Simple) Context Element Declarations are:

  – fixed variables: fix <x> [:: <τ>]

  – assumptions: assume [<thmname>:] „<ϕ>"

    and [<thmname>:] „<ϕ>"

How to Declare Structured Goals

- Recall from the Logical Framework:

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}$$

or

$$A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A_{n+1})\ldots)$$

or

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$$

or

theorem
assumes A_1 and … and A_n
shows A_{n+1}

How to Declare Structured Goals

- Recall from the Logical Framework:

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}$$

Local Proof Contexts

or

$$A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A_{n+1}).$$

or

$$\llbracket A_1; \ldots; A_n \rrbracket \Longrightarrow A_{n+1}$$

or

theorem

assumes A_1 and … and A_n

shows A_{n+1}

How to Declare Local Proof Contexts

- In contrast (Rich) Context Elements are:
  - fixed variables: fixes <x> [:: <τ>]
  - assumptions: assumes [<thmname>:] „<ϕ>“
  - local definition: defines <x> ≡ <t>
  - reconsidering facts: notes a1=b1 … an=bn
  - intermed. results: have [<thmname>:] „<ϕ>“<proof>

Local Proof Contexts (Recall)

- A number of commands in Isabelle/Pure are concerned with Proofs, i.e. the syntactic category <proof>.
- When starting a proof, Isabelle creates a proof context which is:

$$\Gamma \vdash_{\Theta} \phi \quad + \text{ additional information}$$

- Commands transforming proof contexts are called methods (Θ remains fix)
- The command "done" closes a proof

A means to denote Rich Proof Contexts:

Notepads

How to Build "Rich Proof Contexts"

- A constructor for proof-contexts is:

```
notepad
begin
    { fix x
        assume r1: "(A x ⟹ B x) ⟹ C"
        assume r2: "A x ⟹ B x"
        have D sorry
} print_statement this
find_theorems C
end
```

How to Build “Rich Proof Contexts“

A notepad has a local proof-state with environments for

- fixes
- assumptions
- bindings
- facts
- using
- cases
- … and the final goal: shows

- notepads are the building blocks of structured proofs and can be nested.
- a stack of notepad states (with fixes, assumptions, bindings …) can be seen as “the state of the Isar-engine”

How to Build "Rich Proof Contexts"

- A notepad has a local proof-state with environments for
  - fixes
  - assumptions ("facts") (inspect via thm)
  - bindings (inspect via print_binds)
  - using ("a buffer for assms")
  - cases (inspect via print_cases)
  - … and the final goal: shows
- notepads are the building blocks of structured proofs and can be nested.
- a stack of notepad states (with fixes, assumptions, bindings …) can be seen as "the state of the Isar-engine"

Demo V II

- use Some aspects of Isabelle/Isar;
- use Makarius.tar.gz

Fundamentals of Structured Proofs in Isabelle/Isar

Structured Proofs in <proof>

- Notations for proofs so far:
  - ellipses:

    sorry, oops
  - "one-liners" simp and auto:

    by(<method>)   (abbrev: apply(...) done)
  - "apply-style proofs", backward-proofs:

    apply(<method>) … apply(<method>)

    done <method>
  - structured proofs:

    proof (<method>) … qed

Structured Proofs

- the structured <proof> option

  - proof (<initial method>)
            <notepad> {next <notepad>}*
       qed   [(<final method>)]

allows to declare a number
of notepads (declaratively declared subproblems)
that were

MATCHED

against the subgoals after <initial method>

The Syntactic Category <proof>

- structured proofs (in detail):

```
proof ((<method>) | -)
   <notepad>
   {next   <notepad>}*
  qed [ (<method>) ]
```

- notepads:

<rich ctxt element>* show "<ϕ>" <proof>

The Syntactic Category <proof>

- structured proofs:
  - allow to declare sub-goals declaratively
    (eased by pattern-matching and abbreviations)
  - subgoals were matched against the proof context
    (order irrelevant, lifting over parameters and assumptions)
  - allow for advanced notation
    for matching constructs following
    induction and case distinction
  - can be nested
  - extensible (see ITP2014: “Eisbach”)

Running Example 1

- Running Example: A compact proof for *reverse_conc* in Example *Induction.thy* might look like this:

```
lemma reverse_conc:
"reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
by (induct xs) (simp_all add: conc_empty conc_assoc)
```

Compact imperative, apply-style proof via by persuing induction and subsequent simplification on the resulting subgoals.

Running Example 2

- Running Example as simple structured proof:

```
lemma reverse_conc':
        "reverse (conc xs ys) =  conc (reverse ys) (reverse xs)"
 proof (induct xs) (* *)
    show "reverse (conc Empty ys) = conc (reverse ys) (reverse Empty)"
        by(simp add: conc_empty)
 next
    fix a xs
    assume A: "reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
    show       "reverse (conc (Seq a xs) ys) =
                 conc (reverse ys) (reverse (Seq a xs))"
        using A by(simp add: conc_assoc)
 qed
```

Running Example 2

- Running Example as simple structured proof:

At position (* *) the output of the Isar-Engine is:

1. reverse (conc Empty ys) = conc (reverse ys) (reverse Empty)

2. $\bigwedge$a xs.
   reverse (conc xs ys) = conc (reverse ys) (reverse xs) $\Longrightarrow$
   reverse (conc (Seq a xs) ys) = conc (reverse ys) (reverse (Seq a xs))

which is exactly matched by the re-declaration in the two notepads separated by next. Redeclaration is a means both to increase *readability* and *portability* (since this is formally checked text, the redundancy will not be a source of degrading correctness during development), but clearly redundancy may be unwanted blur.

By the way, the order of the notepads does not play a role:

Running Example 2'

- Running Example as simple structured proof:

```
lemma reverse_conc":
        "reverse (conc xs ys) =  conc (reverse ys) (reverse xs)"
 proof (induct xs) (* *)
    fix a xs
    assume A: "reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
    show      "reverse (conc (Seq a xs) ys) =
                conc (reverse ys) (reverse (Seq a xs))"
        using A by(simp add: conc_assoc)
 next
    show "reverse (conc Empty ys) = conc (reverse ys) (reverse Empty)"
        by(simp add: conc_empty)
 qed
```

Running Example 3

- Running Example as structured proof with abbreviations (declared by matching):

```
lemma reverse_conc''':
        "reverse (conc xs ys) =  conc (reverse ys) (reverse xs)"
         (is "?lhs xs = ?rhs xs ")
  proof (induct xs) print_binds
     show "?lhs Empty = ?rhs Empty"
          by(simp add: conc_empty)
next
     fix a xs
     assume A:"?lhs xs = ?rhs xs" show "?lhs(Seq a xs) = ?rhs(Seq a xs)"
          using A by(simp add: conc_assoc)
  qed
```

Running Example 3

- Running Example as simple structured proof:

At position print_binds the output of the Isar-Engine is:

term bindings:
?lhs ≡ λa. reverse (conc a ys)
?rhs ≡ λa. conc (reverse ys) (reverse a)
?thesis ≡ reverse (conc xs ys) = conc (reverse ys) (reverse xs)

which shows the two schema-variables ?lhs and ?rhs defined by HO-Unification and, by the way, explains what ?thesis is:
the schema-variable that is by default matched against the conclusion of the goal.

As one can see, these bindings may be reused in the re-declarations and can reduce blur dramatically. If carefully used, this can increase the understanding of the proof substantially.

Running Example 4

- Running Example as cases structured proof:

```
lemma reverse_conc':
          "reverse (conc xs ys) =  conc (reverse ys) (reverse xs)"
 proof (induct xs) print_cases
    case Empty show "?case" by(simp add: conc_empty)
 next
    case (Seq a xs) from Seq.hyps show "?case"
          by(simp add: conc_assoc)
 qed
```

Running Example 4

- Running Example as simple structured proof:

At position print_cases the output of the Isar-Engine is:

```
cases:
 Empty:
   let "?case" = "reverse (conc Empty ys) = conc (reverse ys) (reverse Empty)"
 Seq:
   fix a_ xs_
   let "?case" = "reverse (conc (Seq a_ xs_) ys) =
                      conc (reverse ys) (reverse (Seq a_ xs_))"
   assume "Seq.hyps":"reverse(conc xs_ ys)=conc (reverse ys)(reverse xs_)"
     and "Seq.prems" :
```

- This is an environment of environments, that modify the bindings and assumptions accordingly. A sub-environment is activated with the case switch, the outer syntax for case-selectors may be parameterized by arguments that instantiate the fixes of that case.

Running Example 4

- Note that this setup in the cases-environment of the Isar-Engine is an effect of the init-method, in our case (induct xs).

  A few methods influence the case-environment:

  - induct (but not the older: induct_tac)
  - cases (but not the older: case_tac)
  - ...

Running Example 5 ...

- Note that

  ```
  case (Seq a xs) show "?case" using Seq.hyps
              by(simp add: conc_assoc)
  ```

- is equivalent to:

  ```
  case (Seq a xs) from Seq.hyps show "?case"
              by(simp add: conc_assoc)
  ```

  is equivalent to:

  ```
  case (Seq a xs) from `reverse (conc xs ys) =
                       conc (reverse ys) (reverse xs)`
  show "?case"  by(simp add: conc_assoc)
  ```

  where `<pattern>` allows explicit search of an assumption in the local proof context; all these variants offer different proof abstraction levels.

Demo V III

- use Some aspects of Isabelle/Isar;
- use Makarius.tar.gz, Compilation.thy