universität freiburg

Isar Proofs

Mathias Fleury

January 12, 2024

Statements

Case distinction

Let's Write everything

Let

Organizing Proofs

Conclusion

Aim

Isar (based on MizAr) tries to produce readable proofs forward proofs, while apply is backwards.

But remember: readable does not mean easy to write.

Not Isar (I)

```
theorem
fixes f :: bool ⇒ bool
shows f (f (f b)) = f b
proof (cases b)
  case True
  note b = True
  show ?thesis
  proof (cases f True)
  case True
   assume fT : f True
   then show f(f(f(b))) = f b using fT b by simp
  next case False
   assume fF: ¬ f True
   then show ?thesis
       proof (cases f False)
       case True
        then show ?thesis using b fF by simp
       next case False
```

Not Isar (II)

```
theorem Kaminski-theorem:
  fixes f :: bool ⇒ bool
  shows f (f (f b)) = f b
  apply (cases b)
   apply (cases ‹f True›)
    apply (cases ‹f False›) — indentation indicates how many goals are left
     apply auto[] — to force auto to work on first goal only
    apply auto[]
   apply (cases ‹f False›)
    apply auto[]
   apply auto[]
  apply (cases ‹f True›)
   apply (cases ‹f False›)
    apply auto[]
   apply auto[]
  apply (cases ‹f False›)
   apply auto[]
  apply auto[]
```

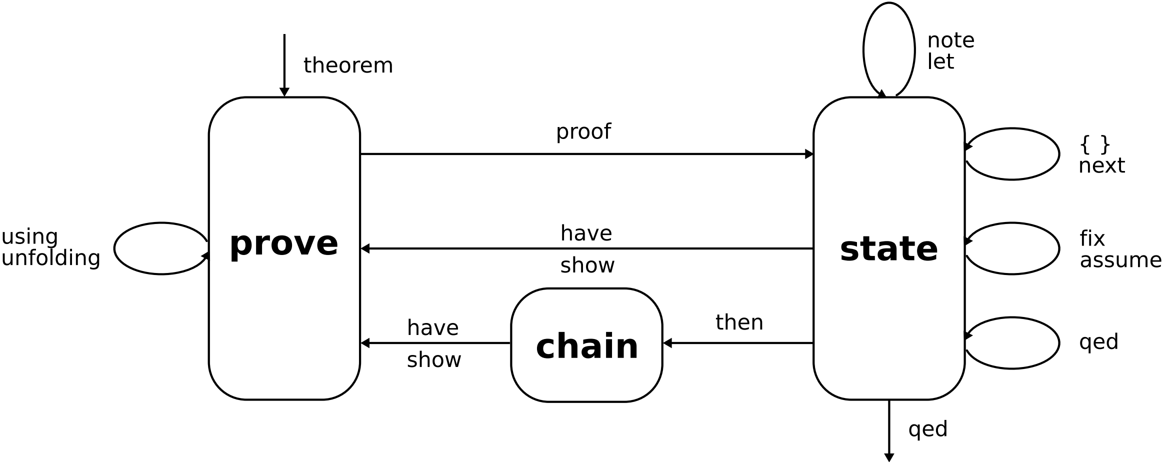
Not Isar (II)

Apply-style is:

- hard to understand
- hard to maintain

  But there is a trade-off: typically refinement proofs are easy but very big, making it unclear whether Isar is a good idea or not.

Statements


theorem
note
let
proof
{ }
next
using
unfolding
prove
have
show
state
fix
assume
have
show
chain
then
qed
qed

In general: use the suggested completion. If there is none, it is

```
proof –
  show ?thesis
    sorry
qed
```

notepad begin
have name-P: P and name-Q: Q if A_1 and A_2 for x y and z
proof –

short for $\bigwedge x\ y\ z.\ [\![A_1;\ A_2]\!] \Longrightarrow P$ and $\bigwedge x\ y\ z.\ [\![A_1;\ A_2]\!] \Longrightarrow Q$.

The assumptions are called A_1
A_2

```
  show P
    sorry
  show Q
    sorry
qed
```

Here the steps are name $[\![A_1;\ A_2]\!] \Longrightarrow P$ and $[\![A_1;\ A_2]\!] \Longrightarrow Q$.

end

```
lemma
  obtains P where
    ‹P x› and
    ‹x ⟶ P (¬x)›
proof –
  obtain z where
    z
    by blast
  let ?P = ‹λ-. True›
  show thesis
    using that[of ?P]
    by auto
qed
```

- In lemmas: the version with ’s’
- Within a proof block, the version without ’s’

```
lemma
  obtains P where
    ‹P x› and
    ‹x ⟶ P (¬x)›
proof –
  let ?P = ‹λ-. True›
  obtain P where
    ‹P x› and
    ‹x ⟶ P (¬x)›
    by auto
  show thesis
    using that ‹P x› ‹x ⟶ P (¬x)›
    by fast
qed
```

- Within a proof block, the version without 's'

```
lemma
  obtains P where
    ‹P x› and
    ‹x ⟶ P (¬x)›
proof –
  let ?P = ‹λ-. True›
  obtain P where
    ‹P x› and
    ‹x ⟶ P (¬x)›
    by auto
  then show thesis
    using that
    by auto
qed
```

- Use 'then' to thread a context

Case distinction

```
lemma
  obtains P where
    ‹P x› and
    ‹x ⟶ P (¬x)›
proof –
  consider
    (C1) ‹x› |
    (C2) ‹¬x›
    by blast
  then show ?thesis
  proof cases
    — Isabelle suggest the cases to insert!
    oops
```

Let's Write everything

case + show ?thesis is the same as write assume and explicitely naming the goal

```
lemma
  fixes n :: nat
  assumes ‹P n›
  shows ‹f n›
  using assms
proof (induction n)
  assume ‹P 0›
  show ‹f 0›
    sorry
next
  fix n :: nat
  assume ‹P n ⟹ f n› and ‹P (Suc n)›
  show ‹f (Suc n)›
    sorry
qed
```

Be careful: if the show is not correct, error only

Be careful: if the show is not correct, error only in the show, not in the assume

```
lemma
  fixes n :: nat
  assumes ‹P n›
  shows ‹f n›
  using assms
proof (induction n)
  assume ‹P 0›
  show ‹f 0›
    sorry
oops
```

Let

let ?Q = ‹True›
let ?P = ‹term ?Q›

Remark that abbreviation are not folded:

term ?P

same as

term ‹term True :: ′a›

Organizing Proofs

At the most basic level there is context.
Allows to share assumptions and fixed variables.

Locales

Basically named version of context
With inheritance.
See locale tutorial.
Typically, equivalent to "from now we assume that".

```
locale mylocale =
  fixes zero :: ‹'a :: {plus}›
  assumes ‹⋀a b :: 'a. a + b = b + a› and
    ‹⋀a. a + zero = a›
interpretation mylocale ‹0::nat›
  by unfold-locales auto
```

Locales

Or classes with usual limits:

- only type per class
- only one instantiation per type (no (Z, divide) and (Z, +) as monoids).

Higher-Level

Isabelle mimics LaTeX in order to produce HTML and PDFs, so:

- there are sessions with ROOT files
- you can split the development over multiple files
- section / subsection / ... all exist

Conclusion

There is lot more in Isar.
Reading the Isar-ref documentation is not a good idea.