Preuves Interactives et Applications

Burkhart Wolff

http://www.lri.fr/~wolff/teach-material/2020-2021/M2-CSMR

Université Paris-Saclay

Induction, Cases and Structured Proofs

Outline

- Inductive Sets and lfp-Fixed Points revisited
- Induction and Case-Distinctions considered logically
- Induction and Cases in Isabelle
- Introduction to Structured Proofs in Isar

Specification Mechanism Commands

- Datatype Definitions (similar SML):
  Examples:

```
datatype  mynat = ZERO | SUC mynat

datatype 'a list = MT | CONS "'a" "'a list"
```

Induction and lfp-Fixed-Points Revisited

Command Inductive Set

- Inductively Defined Sets:

```
inductive_set    <c> :: " τ set"  [for A::τ]
          where  <thmname> : "<φ>"
                   | ...
               | <thmname> = <φ>
```

```
inductive     <c> :: "  τ ⇒ bool" for A::τ

          where  <thmname> : "<φ>"
                   | ...
               | <thmname> = <φ>
```

```
example:     inductive_set     Even :: "int set"
                 where        null: "0 ∈ Even"
                             | plus:"x ∈ Even ⟹ x+2 ∈ Even"

                            | min :"x ∈ Even ⟹ x-2 ∈ Even"
```

Command Inductive Set

- These are not built-in constructs in Isabelle, rather they are based on a series of definitions and typedefs.
- The machinery behind is based on a fixed-point combinator on sets:

$$\text{lfp} :: \text{“}(\text{'}\alpha\ \text{set} \Rightarrow \text{'}\alpha\ \text{set}) \Rightarrow \text{'}\alpha\ \text{set”}$$

which can be conservatively defined by:

$$\text{lfp}\ f = \bigcap \{u.\ f\ u \subseteq u\}$$

and which enjoys a constrained fixed-point property:

$$\text{mono}\ f \Longrightarrow \text{lfp}\ f = f\ (\text{lfp}\ f)$$

Command Inductive Set

- Example : Even (see before)
  - the set Even is conservatively defined by:

    $$\text{Even} = \text{lfp } (\lambda\, X{::}\text{int set}.\quad \{0\} \cup (\lambda\, x.\, x + 2) \text{ ` } X \cup (\lambda\, x.\, x - 2) \text{ ` } X)$$

    where _ `_ :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set is a "map" on sets
  - from which the properties:

    null: "0 ∈ Even"

    plus: "x ∈ Even ⟹ x+2 ∈ Even"

    min :"x ∈ Even ⟹ x-2 ∈ Even"

    can be derived automatically

Command Inductive Set

- Example : Even (see before)
  - More important: it derives an

    induction scheme

    for the Even set.
  - That is: if we know that
    - some x is in Even
    - and some property P over some arbitrary a is maintained (invariant) for a+2 and a-2
    - P x holds.

Command Inductive Set

- Example : Even (see before)
  - In textbooks on Natural Deduction (like van Dalens Book) we might find this formalized in:

$$\frac{x \in Even \quad P(0) \quad \begin{array}{c}[a \in Even; P(a)]_a \\ \vdots \\ P(a+2)\end{array} \quad \begin{array}{c}[a \in Even; P(a)]_a \\ \vdots \\ P(a-2)\end{array}}{P(x)}$$

  - Note that a is free and does only occur in these sub-proof-trees

Command Inductive Set

- Example : Even (see before)
  - Isabelle derives this as theorem from the lfp definition and displays it in Pure follows:

$$x \in Even$$

$$\Longrightarrow P\ 0$$

$$\Longrightarrow \bigwedge x.\ x \in Even \Longrightarrow P\ x \Longrightarrow P\ (x + 2)$$

$$\Longrightarrow \bigwedge x.\ x \in Even \Longrightarrow P\ x \Longrightarrow P\ (x - 2)$$

$$\Longrightarrow P\ x$$

Command Inductive Set

- Example : Even (see before)
  - or equivalently:

assumes "$x \in$ Even"

and base: "P 0"

and step1: "$\bigwedge x. [\![x \in \text{Even}; P\ x]\!] \Longrightarrow P\ (x + 2)$"

and step2: "$\bigwedge x. [\![x \in \text{Even}; P\ x]\!] \Longrightarrow P\ (x - 2)$"

shows "P x"

Command Inductive Set

- Example 2: WellTypedness

  - datatype "τ" = TV_τ string | $Type_\tau$ string "τ list"
  - type_synonym ctxt = "(string × τ) list"
  - definition fun_typ :: "τ ⇒ τ ⇒ τ" (infixr "$\Rightarrow_\tau$" 70)

    where "fun_typ τ τ' ≡ $Type_\tau$ (''fun'') [τ, τ']"

  - datatype "term" = Var string | Const string
    | Abs string "term"
    | App "term" "term" (infix "°" 80)

where the pragma (infixr "$\Rightarrow_\tau$" 70) **instructs Isabelle's parser and pretty-printer to accept** " t $\Rightarrow_\tau$ t' " **as alternative notation for** " fun_typ t t' ".

Command Inductive Set

- Example 2: WellTypedness (inductively defined)

```
inductive  wellTyped :: "ctxt ⇒ ctxt ⇒ term ⇒ τ ⇒ bool"
  where
     con : "    (s, τ) ∈ set Σ   ⟹ wellTyped Σ Γ (Const  s) (instantiate f τ)"
   | var : "    (s, τ) ∈ set Γ   ⟹ wellTyped Σ Γ (Var  s) τ"
   | appl: "    wellTyped Σ Γ f (τ ⇒τ τ')
              ⟹ wellTyped Σ Γ a τ
              ⟹ wellTyped Σ Γ (f ° a) τ' "
   | abstr: "   wellTyped Σ ((x,τ) # (filter (λp. fst p ≠ x) Γ)) body τ'
              ⟹ wellTyped Σ Γ (Abs x body) (τ ⇒τ τ')"
```

which reduces syntactically with a pragma for mixfix notation
(see chapter 8.2 in the Isar Reference Manual)

("((_),(_) ⊢/ (_) :: (_))" [60,0,60] 60)

to

Command Inductive Set

- Example 2: WellTypedness (inductively defined)

```
inductive  is_WELLFORMED :: "ctxt ⇒ ctxt ⇒ term ⇒ τ ⇒ bool"
                            ("((_),(_) ⊢/ (_) :: (_))" [60,0,60] 60)
  where
     con : "    (s, τ) ∈ set Σ    ⟹ (Σ,Γ ⊢ (Const  s) :: instantiate f τ)"
   | var : "    (s, τ) ∈ set Γ     ⟹ (Σ,Γ ⊢ (Var  s) :: τ)"
   | appl: "    (Σ,Γ ⊢ f :: τ ⇒τ τ') ⟹ (Σ,Γ ⊢ a :: τ )
              ⟹ (Σ,Γ ⊢ f ° a :: τ'  )"
   | abstr: "   (Σ,(x,τ) # (filter (λp. fst p ≠ x) Γ) ⊢ body  :: τ' )
              ⟹ (Σ,Γ ⊢ Abs x body :: τ ⇒τ τ'  )"
```

which gives the types inference rules not only a precise meaning, but also derived proof principles like

A First Glimpse on Case-Distinction and Induction Rules

Command Inductive Set

- Example 2: WellTypedness (derived consequences)

is_WELLFORMED.induct:

x1, x2 ⊢ x3 :: x4 ⟹

(⋀s τ Σ Γ instantiate f. (s, τ) ∈ set Σ ⟹ P Σ Γ (Const s) (instantiate f τ)) ⟹

(⋀s τ Γ Σ. (s, τ) ∈ set Γ ⟹ P Σ Γ (Var s) τ) ⟹

(⋀Σ Γ f τ τ' a. Σ,Γ ⊢ f :: τ ⇒τ τ' ⟹

P Σ Γ f (τ $\Rightarrow_{\tau}$ τ') ⟹ Σ,Γ ⊢ a :: τ ⟹ P Σ Γ a τ ⟹ P Σ Γ (f ° a) τ') ⟹

(⋀Σ x τ Γ body τ'.

Σ,(x, τ) # filter (λp. fst p ≠ x) Γ ⊢ body :: τ' ⟹

P Σ ((x, τ) # filter (λp. fst p ≠ x) Γ) body τ' ⟹ P Σ Γ (Abs x body) (τ ⇒τ τ')) ⟹

P x1 x2 x3 x4

Command Inductive Set

- Example 2: WellTypedness (derived consequences)

is_WELLFORMED.cases:

$a1, a2 \vdash a3 :: a4 \Longrightarrow$

$(\bigwedge s\ \tau\ \Sigma\ \Gamma$ instantiate f.

$a1 = \Sigma \Longrightarrow a2 = \Gamma \Longrightarrow a3 = \text{Const } s \Longrightarrow a4 = \text{instantiate } f\ \tau \Longrightarrow (s, \tau) \in \text{set } \Sigma \Longrightarrow P) \Longrightarrow$

$(\bigwedge s\ \tau\ \Gamma\ \Sigma.$

$a1 = \Sigma \Longrightarrow a2 = \Gamma \Longrightarrow a3 = \text{Var } s \Longrightarrow a4 = \tau \Longrightarrow (s, \tau) \in \text{set } \Gamma \Longrightarrow P) \Longrightarrow$

$(\bigwedge \Sigma\ \Gamma\ f\ \tau\ \tau'\ a.$

$a1=\Sigma \Longrightarrow a2=\Gamma \Longrightarrow a3=f \circ a \Longrightarrow a4=\tau' \Longrightarrow \Sigma,\Gamma \vdash f :: \tau \Rightarrow_\tau \tau' \Longrightarrow \Sigma,\Gamma \vdash a :: \tau \Longrightarrow P) \Longrightarrow$

$(\bigwedge \Sigma\ x\ \tau\ \Gamma$ body $\tau'.$

$a1=\Sigma \Longrightarrow a2=\Gamma \Longrightarrow a3 = \text{Abs } x \text{ body} \Longrightarrow a4=\tau \Rightarrow_\tau \tau' \Longrightarrow \Sigma,(x,\tau)\#\text{filter}(\lambda p.\ \text{fst } p \neq x)\ \Gamma \vdash \text{body} :: \tau' \Longrightarrow P)$

$\Longrightarrow\ P$

Command Inductive Set

- Remarks
  - Induction schemes (closely related to fixpoints, recursion, and while-loops) are the major weapon in HOL proofs that can NOT be done by automated provers
  - they can refer to (inductive) datatypes, sets and therefore relations and are always the means of choice if we want to express that something is „closed under a set of rules"
  - Usually there are several choices of induction schemes, their instantiation, and the target they are applied on.
  - Like invariants of while-loops, it may be that some generalization of a property can be proven inductively, the concrete property, however, not directly.

Command Inductive Set

- Remarks
  - Obviously, induction rules and/or case-distinction rules over non-trivial inductive schemes are difficult to read
  - … and to get implemented correctly
  - … it gives therefore confidence to have them derived in Isabelle …

Command Inductive Set

- Parametric Inductively Defined Sets
  (like transitive closure on paths):

```
inductive     <c> [ for <v>:: "<τ>" ]
      where  <thmname> : "<φ>"
                       | ...
             | <thmname> = <φ>
```

example: inductive path for rel ::"'a ⇒ 'a ⇒ bool"

where base : "path rel x x"

| step : "rel x y ⟹ path rel y z ⟹ path rel x z"

Command Inductive Set

- Inductively Defined Sets: Example path.
  Isabelle/HOL:

$$\text{path rel x y}$$
$$\Longrightarrow \bigwedge x.\ P\ x\ x;$$
$$\Longrightarrow \bigwedge x\ y\ z.\ \ rel\ x\ y \Longrightarrow path\ rel\ y\ z \Longrightarrow P\ y\ z \Longrightarrow P\ x$$
$$\Longrightarrow P\ x\ y$$

- Text-book:

$$\cfrac{path\ rel\ x\ y \quad [P\ a\ a]_a \quad \begin{matrix}[rel\ a\ b; path\ rel\ b\ c; P\ b\ c]_{a,b,c} \\ \vdots \\ P\ a\ c\end{matrix}}{P\ x\ y}$$

Command Inductive Set

- **Note: an equivalent (appending) induction scheme with the same power:**

$$\text{path rel x y}$$
$$\Longrightarrow (\bigwedge x.\ P\ x\ x)$$
$$\Longrightarrow (\bigwedge x\ y\ z.\ \llbracket \text{path rel x y};\ P\ x\ y;\ \text{rel y z} \rrbracket \Longrightarrow P\ x$$
$$\Longrightarrow P\ x\ y$$

- The choice of the induction scheme matters for the task ahead …

Data-Types and Recursive Funs

- Recall: Datatype Definitions (similar SML):
  (Machinery behind : complex series of const and typedefs !)

datatype ($'a_1..'a_n$) T =
<c> :: "<τ>" | ... | <c> :: "<τ>"

- Recall: Recursive Function Definitions:

fun <c> ::"<τ>" where
"<c> <pattern> = <t>"
| ...
| "<c> <pattern> = <t>"

Command Inductive Datatype

- Example: Induction Scheme from Datatype Definitions
  - $(\bigwedge a.\ P\ (leaf\ a))$
    $\Longrightarrow (\bigwedge a\ t\ t'.\ P\ t \Longrightarrow P\ t' \Longrightarrow P\ (node\ a\ t\ t'))$
    $\Longrightarrow P\ tree$
  - Textbook:

$$\frac{[P(leaf\ a)]_a \qquad \begin{array}{c}[P\ t; P\ t']_{a,t,t'} \\ \vdots \\ P(node\ a\ t\ t')\end{array}}{P\ tree}$$

Command Inductive Datatype

- Example: Recursive Function Definition

```
fun reflect :: "'a tree ⇒ 'a tree"

    where  a : "reflect (leaf x) = leaf x"
             | b : "reflect (node x t t') = node x t' t"
```

- Example Proof: lemma “reflect(reflect t) = t”:
  - Proof by induction (apply style; since tree.induct is just an ordinary (introduction) rule, *this works by rule*)

```
apply(rule_tac tree=t in tree.induct)
 apply(simp add: a)
apply(simp add: b)
done
```

Induction and Cases considered logically

Induction vs. Case-Split

- The commands inductive, inductive_set and datatype generate another important schema of rules which is an important weapon:

Case-Splits

- Most basic form: disjE

$$\frac{A \vee B \qquad \begin{array}{c}[A] \\ \vdots \\ Q\end{array} \qquad \begin{array}{c}[B] \\ \vdots \\ Q\end{array}}{Q}$$

Induction vs. Case-Split

- For the datatype tree, this rule present itself like this:

$$(\bigwedge a.\ y = \text{leaf } a \Longrightarrow Q)$$

$$\Longrightarrow (\bigwedge x\ t\ t'.\ y = \text{node } x\ t\ t' \Longrightarrow Q)$$

$$\Longrightarrow Q$$

$$\dfrac{\begin{array}{c}[x = (leaf\ a)]_a \\ \vdots \\ Q\end{array} \qquad \begin{array}{c}[x = node\ a\ t\ t']_{a,t,t'} \\ \vdots \\ Q\end{array}}{Q}$$

Induction vs. Case-Split

- For the inductive sets, the case split rule path.cases presents itself like this:

$$
\begin{array}{ll}
 & \text{path rel a1 a2} \\
\Longrightarrow & \bigwedge x.\quad a1 = x \Longrightarrow a2 = x \Longrightarrow P \\
\Longrightarrow & \bigwedge x\ y\ z.\ \ a1 = x \Longrightarrow a2 = z \Longrightarrow \\
 & \qquad\qquad \text{rel}\ x\ y \Longrightarrow \text{path rel}\ y\ z \Longrightarrow P \\
\Longrightarrow & P
\end{array}
$$

Induction and Cases in Isabelle

Induction and Case-Splitting Support

- induction and case-splitting were supported by specific methods attempting to figure out automatically which rule to use
- There are apply-style proof methods:

```
apply(induct_tac „<term>“)
```

```
apply(case_tac „<term>“)
```

which work with arbitrary open parameters of a subgoal …

Induction and Case-Splitting Support

- induction and case-splitting were supported by specific methods attempting to figure out automatically which rule to use
- There are proof methods giving support

  for an own structured proof-language Isar

```
apply(induct „<term>“ <options ... >)
```

```
apply(cases „<term>“)
```

which act on parameters which are “fixed” (see later).

Structured Proofs in Isabelle/Isar

Introduction to Isar
Structured Proofs

- A language for structured proofs:

  Isar - Intelligible semi-automated reasoning

- http://isabelle.in.tum.de/Isar/
- supporting a declarative proof-style
  (rather than a procedural one)
- oriented towards "natural deduction style"
- presenting intermediate steps in a
  machine-checked, human readable format

Introduction to Isar Structured Proofs

- Core: the proof environment:

```
proof (<method>)
  [case - fix - assumes - defs- have-]
  show “<goal>” <proof>
next
   ...
next
   [case - fix - assumes - defs- have-]
  show “<goal>” <proof>
qed
```

- … a switch from procedural to declarative style can be done by rephrasing the goals

Introduction to Isar Structured Proofs

- Instead of the goal format:

$$\bigwedge a_1 \ldots a_n.\ A_1 \Longrightarrow \ldots A_m \Longrightarrow P$$

the "Isar"-format:

fix a_1::<typ> ... fix a_n::<typ>
  assume A_1 and ... and A_m
show P

is preferable because ...

Introduction to Isar Structured Proofs

- is preferable
  - labelling of assumptions
  - control of goal parameters
  - intermediate steps "have"
  - support for equational reasoning
  - abbreviations
  - pattern-matching
  - support for cases and inductions, which become proof-structuring concepts

Introduction to Isar Structured Proofs

- The methods induct and cases produce a list of local contexts (shown by the diagnostic command print_cases) with the appropriate fix'es and assume's
- Example:

```
lemma "reflect(reflect t) = t”
    proof(induct t) print_cases
      case (leaf x) then show ?case sorry
    next
      case (node x1a t1 t2) then show ?case sorry
   qed
```

A Structured „Classical" Proof

- Example: (Nested) Proof by Contradiction

```
theorem "((A ⟶ B) ⟶ A) ⟶ A"
proof
  assume "(A ⟶ B) ⟶ A"
  show A
  proof (rule classical)
    assume "¬ A"
    have "A ⟶ B"
    proof
      assume A
      with ‹¬ A› show B by contradiction
    qed
    with ‹(A ⟶ B) ⟶ A› show A ..
  qed
qed
```

Nameless selection from local context

A Structured „Classical“ Proof

- Example: A Calculational Proof

```
lemma (in group) group_right_inverse: "x * inverse x = 1"
proof -
  have "x * inverse x = 1 * (x * inverse x)"
    by (simp only: group_left_one)
  also have "... = 1 * x * inverse x"
    by (simp only: group_assoc)
  also have "... = inverse (inverse x) * inverse x * x * inverse x"
    by (simp only: group_left_inverse)
  also have "... = inverse (inverse x) * (inverse x * x) * inverse x"
    by (simp only: group_assoc)
  also have "... = inverse (inverse x) * 1 * inverse x"
    by (simp only: group_left_inverse)
  also have "... = inverse (inverse x) * (1 * inverse x)"
    by (simp only: group_assoc)
  also have "... = inverse (inverse x) * inverse x"
    by (simp only: group_left_one)
  also have "... = 1"
    by (simp only: group_left_inverse)
  finally show ?thesis .
qed
```

A Structured „Classical" Proof

- Example: Induction, Calculation, Patterns …
  towards a comprehensive human- readable
  proof presentation format

```
theorem sum_of_odds:
  "(∑i::nat=0..<n. 2 * i + 1) = n^Suc (Suc 0)"
  (is "?P n" is "?S n = _")
proof (induct n)
  show "?P 0" by simp
next
  fix n
  let ?two="Suc(Suc(0))"
  have "?S (n + 1) = ?S n + 2 * n + 1"
    by simp
  also assume "?S n = n^?two"
  also have "... + 2 * n + 1 = (n + 1)^?two"
    by simp
  finally show "?P (Suc n)"
    by simp
qed
```

introducing abbreviations by pattern-matching

proof structuring method

local abbreviation

intermediate goal

catching induction hyp

intermediate lemma

presenting main goal in terms of abbrevs

A Structured „Classical“ Proof

- MORE EXAMPLES ON “Proof Patterns”:

  Tobias Nipkow:
  “Programming and Proving in Isabelle/HOL”

  (online documentation)

Conclusion

- Induction is at the heart of interactive proving; this requires the most human ingenuity
- Isabelle offers support for inductive and case-distinction based proofs
- the Isar-language paves the way for adequate presentation of common proof-structures (by induction, by case distinction,...)
- ... and by the way, Isar paved the way for better portability and parallel proof-checking