Structured Induction Proofs in Isabelle/Isar

Makarius Wenzel

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~wenzelm/

Abstract. Isabelle/Isar is a generic framework for human-readable formal proof documents, based on higher-order natural deduction. The Isar proof language provides general principles that may be instantiated to particular object-logics and applications. We discuss specific Isar language elements that support complex induction patterns of practical importance. Despite the additional bookkeeping required for induction with local facts and parameters, definitions, simultaneous goals and multiple rules, the resulting Isar proof texts turn out well-structured and readable. Our techniques can be applied to non-standard variants of induction as well, such as co-induction and nominal induction. This demonstrates that Isar provides a viable platform for building domain-specific tools that support fully-formal mathematical proof composition.

1 Motivation

1.1 The Isar philosophy

Isabelle/Isar [15, 16, 7, 17] is intended as a generic framework for developing formal mathematical documents with full proof checking. The Isabelle/Isar system is well integrated with existing theorem prover interface technology [1] and document preparation based on PDF-LaTeX.[1] The main objective is the design of a human-readable structured proof language, which is called the "primary proof format" in Isar terminology.

Such a primary proof language is somewhere in the middle between the extremes of primitive proof objects and actual natural language. In this respect, Isar is a bit more formalistic than Mizar [12, 10], using explicit logical connectives for certain reasoning schemes where Mizar would prefer English words; see [19, 18] for further comparisons of these systems. We argue that any effort of building a library of formalized mathematics heavily depends on a reasonable notion of structured proofs — the Mizar Mathematical Library [6] provides some empirical evidence for this.

So Isar challenges the traditional way of recording informal proofs in mathematical prose, as well as the common tendency to see fully formal proofs directly as objects of some logical calculus (e.g. λ-terms in a version of type theory). In fact, Isar is better understood as an interpreter of a simple block-structured language for describing data flow of local facts and goals, interspersed with occasional invocations of proof methods.

[1] In fact, the present paper has been prepared as an Isabelle/Isar theory document, which means that the proofs and proof outlines encountered here have been checked by the system.

Everything is reduced to logical inferences internally, but these steps are somewhat marginal compared to the overall bookkeeping of the interpretation process. Thanks to careful design of the syntax and semantics of Isar language elements, a formal record of Isar instructions may actually appear as an intelligible text to the attentive reader.

Consuming well-structured Isar proofs is less demanding than producing them in the first place. The main effort is to get started with new formalizations, while modifying existing proofs is much easier. With sufficient background material available, a proof author will first consume previous developments, and then produce some derivative work. This general procedure is taken for granted in science, but is not quite established yet in formal theory development — despite the relative success of the Mizar Mathematical Library.

1.2 Generic natural deduction

Isabelle/Isar is based on the existing logical framework of Isabelle/Pure [9], which provides a generic platform for higher-order natural deduction. The generic approach of Pure inference systems is extended by Isar towards actual proof texts. Applications require another intermediate layer: an object-logic. Presently Isabelle/HOL [8] (simply-typed higher-order logic) is being used most of the time. Isabelle/ZF is less extensively developed, although it would probably fit better for classical mathematics.

The Isar proof language offers various common principles that are parameterized by entities of the object-logic and application context, including plug-in proof methods. The most basic method is called *rule* and refers to a generic natural deduction step of the underlying framework (essentially combined forward-backward chaining). The application context provides declarations of canonical introduction and elimination rules for various kinds of connectives and derived operations, covering the object-logic $\wedge$, $\vee$, $\forall$, $\exists$, $\ldots$, set-theory $\cap$, $\cup$, $\bigcap$, $\bigcup$, $\ldots$, and any application specific notions such as operators of lattice theory, topology, etc. With reasonable rule declarations in the context, proofs involving common elements of discourse proceed in an implicit manner, where rules are determined by the structure of propositions stated explicitly. For example:

assume $x \in A$ **and** $x \in B$
then have $x \in A \cap B$ **by** *rule*

Abbreviations like "**..**" for "**by** *rule*" make this less formalistic. Similarly **proof** ... **qed** already initiates a certain reasoning pattern from the context, unless an alternative is named explicitly. Note that beyond selecting single rules by the syntactic structure of goals and facts, which works by means of higher-order unification, Isabelle/Isar never appeals to hidden automated reasoning. Any such proof tools need to be invoked explicitly, usually in terminal position such as "**by** *simp*".

The subsequent example performs basic natural deduction with an explicit rule applied in a purely backwards manner; the arising sub-proofs are completed accordingly.

fix $n :: nat$
have $P\ n$
proof (*rule nat-induct*)

```
  show P 0 ⟨proof⟩
next
  fix n
  assume P n
  show P (Suc n) ⟨proof⟩
qed
```

This is already a structured induction proof in Isabelle/Isar, so in theory we could conclude the present paper just now. In practice, though, various issues arise in presenting an inductive statement $P\ n$ properly — slightly annoying auxiliary structure can occur here. We shall introduce a significantly improved version of the Isar *induct* method that enables extraneous logical bookkeeping to be suppressed from the proof text.

1.3 Case-study: complete induction with local definitions

Consider the following problem of recreational mathematics: "Given some function f on natural numbers, such that $f\ (f\,n) < f\ (n+1)$ for all n, show that f is the identity." Here we formalize only the easier part: $n \leq f\,n$ for arbitrary n. The proof is by complete induction on $f\,n$, along the $<$ relation on natural numbers.

We shall work in Isabelle/HOL, which uses fairly standard mathematical notation. Special attention needs to be paid to the "three arrows" $\Rightarrow/\bigwedge/\Longrightarrow$ of Isabelle/Pure (cf. §2.1), which occur whenever abstract syntax or inference rules of the logical framework need to be introduced explicitly. Note that Isar proofs may reference previous facts either by name (e.g. *asm*) or proposition (e.g. ‹$n = m + 1$›). Furthermore "**.**" stands for "**by** *this*", i.e. immediate composition of facts without any rule in between.

```
1  lemma example:
2    fixes f :: nat ⇒ nat
3    assumes asm: ⋀n. f (f n) < f (n + 1)
4    shows n ≤ f n
5  proof (induct k = f n fixing: n rule: less-induct)
6    case (less k)
7    then have IH: ⋀m. f m < f n ⟹ m ≤ f m by simp
8    show ?case
9    proof cases
10     assume n = 0
11     then show ?thesis by simp
12   next
13     assume n ≠ 0
14     then obtain m where n = m + 1 by (cases n) simp-all
15     from asm have f (f m) < f n unfolding ‹n = m + 1› .
16     with IH have f m ≤ f (f m) .
17     also note ‹f (f m) < f n›
18     finally have f m < f n .
19     with IH have m ≤ f m .
20     also note ‹f m < f n›
21     finally have m < f n .
22     then show n ≤ f n using ‹n = m + 1› by simp
```

```
23      qed
24    qed
```

The general structure of our proof is as follows: (i) the main statement (lines 1–4), (ii) initiating the induction (lines 5–8), (iii) splitting the proof body into two cases and solving the trivial one (lines 9–12), (iv) finish the interesting second case with two appeals to the induction hypothesis (lines 13–23).

Part (i) already starts the proof by giving a structured Isar statement, which consists of several proof context elements (**fixes**, **assumes**) followed by the main conclusion (**shows**). Thus we may commence the actual reasoning immediately, without decomposing the problem into its constituent parts first. The final result will be extracted from the initial statement as a closed formula, namely $(\bigwedge n.\, f\,(f\,n) < f\,(n+1)) \Longrightarrow n \leq f\,n$.

Part (ii) shall be our main technical concern. We have used the *induct* method, providing some additional information about the precise mode of reasoning to be performed. The arguments include the induction variable k, which is locally defined as $k = f\,n$. We also fix n as an auxiliary induction parameter now, because the induction step will require different instances of $f\,n$. We further need to specify the underlying induction principle *less-induct*, because the default rule for natural numbers would merely consider zero and immediate successor cases. The subsequent **case** element augments the proof context by additional parameters and assumptions stemming from the induction step (rule *less-induct* has only one such case, namely *less*). From this we conclude the induction hypothesis *IH* in a convenient form — the raw hypothesis would still mention the local definition of $k = f\,n$, but we prefer the expanded version here. Then we are ready to work on the inductive conclusion: "**show** *?case*" (the abbreviation of *?case* for the recurrent proposition $n \leq f\,n$ also stems from the *less* context).

Part (iii) is not very special, but note that the *cases* method applies *tertium-non-datur*.

Part (iv) is most interesting from the mathematical viewpoint. Technically, we merely conduct a few steps of calculational reasoning [3] in Isar (with "glue statements" **also** and **finally**), while results are composed in a mostly trivial manner; we rarely invoke automated reasoning support here. In line 14, syntactic case analysis converts between $m + 1$ and *Suc m*, which is the preferred representation in the Isabelle/HOL library.

In informal mathematical practice, one would probably just present the main body of part (iv), and include some hints about the induction. Compared to that, our proof is almost twice as long. In particular, part (ii) requires further investigation: the 4 lines being spent here are not that bad, as we shall see now.

The following version performs the induction without any specific support. In order to present the problem as a closed inductive proposition, the local definition and variable generalization is simulated within the object-logic as $\forall n.\, k = f\,n \longrightarrow n \leq f\,n$.

```
1   lemma example:
2     fixes f :: nat ⇒ nat
3     assumes asm: ⋀n. f (f n) < f (n + 1)
4     shows n ≤ f n
5   proof −
6     { fix k have ∀ n. k = f n ⟶ n ≤ f n
```

```
7         proof (rule less-induct)
8           fix k assume less: ⋀m. m < k ⟹ ∀n. m = f n ⟶ n ≤ f n
9           show ∀n. k = f n ⟶ n ≤ f n
10          proof
11            fix n
12            show k = f n ⟶ n ≤ f n
13            proof
14              assume k = f n
15              with less have IH: ⋀m. f m < f n ⟹ m ≤ f m by simp
16              show n ≤ f n ⟨proof⟩
17            qed
18          qed
19        qed
20      } then show ?thesis by simp
21    qed
```

Here we have spent 21 lines without doing anything useful, since the proof of $n \leq f\,n$ in line 16 has been left out! (Lines 9–23 of the previous version may be pasted here.)

Dealing with compound formulas such as $\forall n.\ k = f\,n \longrightarrow n \leq f\,n$ imposes extra inconveniences in our framework, because the object-language needs to be decomposed into Pure first. Even worse, this extra work is multiplied in induction proofs: (i) reformulate the original problem (line 6), (ii) decompose both the induction hypothesis and conclusion in the proof body (lines 8–15), (iii) apply the modified result to solve the main problem (line 20). Some parts have been automated in an ad-hoc fashion using "**by** *simp*".

One could argue now that Isar should not insist on Pure rule composition, but let the user transform the logical structure of a pending problem more directly. This would approximate the Mizar language, with separate language elements to dig into connectives of first-order logic. On the other hand, it would not really solve the problem at hand. Turning $n \leq f\,n$ into $\forall n.\ k = f\,n \longrightarrow n \leq f\,n$ already demands efforts that are irrelevant to the application. Since interactive proof development usually requires some experimentation to figure out the induction parameters in the first place, we would like to reduce logical clutter as much as possible.

As a rule of thumb, Mizar is better at decomposing statements of first-order logic, but Isar does not require any such decomposition, if we manage to represent a problem in Pure form. The basic idea of our enhanced *induct* method is to make complex induction proof patterns appear as a native part of the Isar framework. This is achieved by internalizing portions of Isar proof context into the object-logic, and reverse the effect before handing over to the user to finish the induction cases.

Subsequently, we shall present further details of the Isar framework in §2, describe the *induct* method in §3, and review common induction patterns in §4.

2 Foundations

Isabelle/Isar consists of three main layers: The Isabelle/Pure framework for primitive natural deduction, Isar proof contexts for managing various kinds of local parameters, assumptions, facts, abbreviations etc., and the Isar/VM ("virtual machine") interpreter that implements an incremental model of structured proof composition.

2.1 The Pure framework

The Pure logic [9] is an intuitionistic fragment of higher-order logic. In type-theoretic parlance, there are three levels of λ-calculus with corresponding arrows: $\Rightarrow$ for syntactic function space (terms depending on terms), $\bigwedge$ for universal quantification (proofs depending on terms), and $\Longrightarrow$ for implication (proofs depending on proofs).

Term syntax provides explicit notation for abstraction $\lambda x :: \alpha.\ b(x)$ and application $t\ u$, while types are usually implicit thanks to type-inference; terms of type *prop* are called propositions. Logical statements are composed via $\bigwedge x :: \alpha.\ B(x)$ and $A \Longrightarrow B$. Primitive reasoning operates on judgments of the form $\Gamma \vdash \varphi$, with standard introduction and elimination rules for $\bigwedge$ and $\Longrightarrow$ that refer to fixed parameters $x_1, \ldots, x_m$ and hypotheses $A_1, \ldots, A_n$ from the context Γ. The corresponding proof terms are left implicit.

An object-logic introduces another layer: type o for object propositions, term constants $Tr :: o \Rightarrow prop$ as (implicit) object-truth judgment and connectives like $\wedge :: o \Rightarrow o \Rightarrow o$ or $\forall :: (\alpha \Rightarrow o) \Rightarrow o$, and axioms for object rules such as $conj\text{-}I\colon A \Longrightarrow B \Longrightarrow A \wedge B$ or $all\text{-}I\colon (\bigwedge x.\ B\ x) \Longrightarrow \forall x.\ B\ x$. Derived object rules are represented as theorems of Pure.

Since Pure propositions may be nested arbitrarily, the resulting natural deduction framework extends Gentzen's version [5] such that rules may take arbitrary rules as assumptions [11]. For example, the induction rules $P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ n$ and $(\bigwedge n.\ (\bigwedge m.\ m < n \Longrightarrow P\ m) \Longrightarrow P\ n) \Longrightarrow P\ n$ both fit nicely into this framework.

2.2 Isar proof contexts

In judgments $\Gamma \vdash \varphi$ of the primitive framework, Γ essentially acts like a proof context. Isar elaborates this idea towards a higher-level notion, with separate information for type-inference, term abbreviations, local facts, and generic hypotheses parameterized by discharge rules. For example, the context element **assumes** A introduces a hypothesis with $\Longrightarrow$ introduction as discharge rule; **notes** $a = b$ defines local facts; **defines** $x = a$ and **fixes** $x :: \alpha$ introduce local terms.

Top-level theorem statements may refer directly to such Isar elements to establish a conclusion **shows** B within a certain context; the final result will be in discharged form. There are separate Isar commands to build contexts within a proof body, notably **fix**, **assume**, **obtain** etc. Using Isar proof notation, we can explain the latter three formally:

{
  fix x
  have $B\ x$ $\langle proof \rangle$
}
note $\langle \bigwedge x.\ B\ x \rangle$

{
  assume A
  have B $\langle proof \rangle$
}
note $\langle A \Longrightarrow B \rangle$

{
  obtain x **where** $A\ x$ $\langle proof \rangle$
  have B $\langle proof \rangle$
}
note $\langle B \rangle$

Here **note** is used to indicate the fact resulting from each proof block. The **obtain** above involves a proof of $\bigwedge C.\ (\bigwedge x.\ A\ x \Longrightarrow C) \Longrightarrow C$, which happens to be the body of $\exists$ elimination in first-order logic; it then acts like **fix** x **assume** $A\ x$ using that discharge rule. See [16, §5.3] for more on generalized elimination in Isar.

Isar also supports named context segments called *cases*: the language element **case** c expands to **fix** $x_1\ \ldots\ x_m$ **assume** $A_1\ \ldots\ A_n$ according to a given definition of c in the context; the variant **case** $(c\ y_1\ \ldots\ y_k)$ specifies explicit names for fixed variables introduced here. Cases may also provide term abbreviations, typically *?case* for the conclusion. Isar proof methods are entitled to define cases for the current proof body.

2.3 Isar/VM transitions

A structured Isar proof text consists of a sequence of proof commands, which are interpreted as transitions of the Isar virtual machine. The basic idea is analogous to evaluating algebraic expressions on a stack machine: $(a + b) \cdot c$ then corresponds to a sequence of single transitions for each symbol $(,\ a,\ +,\ b,\ ),\ \cdot,\ c$. In Isar the algebraic values are local facts or goals, and the operations are logical inferences.

The Isar/VM state maintains a stack of nodes, each node contains the local proof context, an optional pending goal, and the linguistic mode. The latter determines the type of transition that may be performed next, it essentially alternates between forward and backward reasoning. For example, in *state* mode Isar acts like a mathematical scratchpad, which accepts various context declarations like **fix**, **assume**, and goal statements like **have** and **show**. A goal changes the mode to *prove*, which means that we may now refine the problem via **unfolding** or **proof**. Then we are again in *state* mode of a proof body, which may issue **show** statements to solve pending subgoals. A concluding **qed** will return to the original *state* mode one level upwards. Isar provides convenient abbreviations, such as "**by** $meth_1\ meth_2$" (terminal proof) for "**proof** $meth_1$ **qed** $meth_2$".

In the following sequence of Isar/VM transitions we indicate block structure and mode:

have $A \longrightarrow B$	*open*	*state* $\rightarrow$ *prove*
proof		*prove* $\rightarrow$ *state*
assume A		
show B	*open*	*state* $\rightarrow$ *prove*
$\langle proof \rangle$	*close*	*prove* $\rightarrow$ *state*
qed	*close*	

In structured Isar proof texts, *state* mode is active most of the time, while *prove* mode is left immediately after 1 or 2 refinement steps steps (say to unfold a definition and apply a standard rule). In contrast, unstructured tactic scripts would stay in *prove* mode indefinitely, applying more and more backward refinements until the goal is solved.

3 The *induct* proof method

The *induct* proof method of Isar is a sophisticated wrapper for the basic *rule* method. Arguments provided in the proof text are interpreted as directions for local modifications of the proof context, induction rule, and goal, culminating in the actual logical

refinement step. The resulting proof body will contain several subgoals corresponding to the inductive cases of the instantiated rule after refinement, together with named cases to enable abbreviations **case** *c* **show** *?case* for each induction step, instead of explicit **fix** $x_1 \ldots x_m$ **assume** $A_1 \ldots A_n$ **show** B. The general method invocation looks like this:

facts
(*induct insts fixing: vars rule: rule*)

Here *facts* refers to the implicit facts being passed to any Isar method according to the proof structure, as indicated by **then** or **using** in the text; *insts* is a list of instantiations, either *x* (variable) or $x = a$ (defined variable); *vars* refers to a list of fixed variables; *rule* refers to the underlying induction principle.

Any of these arguments are optional, and there are sensible defaults. For example, giving n :: *nat* as *insts* determines the rule *nat.induct*; giving $\vdash x \in A$ as *facts* determines both the instantiation *x* and rule *A.induct* (for some inductively defined set *A* in the context). Since mutual induction is also supported, arguments *insts*, *vars*, *rule* may be repeated, using the separator **and**. For example, (*induct t* :: *term* **and** *ts* :: *term list*) could refer to the mutual induction principle stemming from a nested datatype in HOL.

3.1 Generic induction rules

We support a variety of induction rules, with minimal assumptions about the logical presentation. Following canonical Pure formulas, rules represent the outermost $\bigwedge$ prefix as schematic variables. The remainder is an iterated implication $\Longrightarrow$, with arbitrary "major premises" in front, followed by the inductive cases, concluded by some predicate expression *Tr* $(P\ x_1 \ldots x_m)$; recall that the object judgment *Tr* is usually implicit.

The inductive cases may not introduce further schematic variables, apart from those already present in the major premise and the conclusion. Thus inductive cases are fully determined after instantiating the other parts of the rule. Occurrences of the predicate in the cases is restricted to *Tr* $(P\ a_1 \ldots a_m)$ for arbitrary term arguments, but with the outer object judgment still intact. In other words, $P\ a_1 \ldots a_m$ needs to be surrounded by $\bigwedge/\Longrightarrow$ connectives. Thus the predicate may essentially represent object rules.

For example, $(\bigwedge n.\ (\bigwedge m.\ m < n \Longrightarrow P\ m) \Longrightarrow P\ n) \Longrightarrow P\ n$ seen before qualifies as induction rule. The equivalent version $(\bigwedge n.\ \forall m{<}n.\ P\ m \Longrightarrow P\ n) \Longrightarrow P\ n$ in Isabelle/HOL does not work, because $\forall m < n.\ P\ m$ hides the predicate inside a closed object-formula. A rule for some inductive set *A* looks like $x \in A \Longrightarrow \mathit{cases} \Longrightarrow P\ x$, but not $\mathit{cases} \Longrightarrow \forall x \in A.\ P\ x$ as is occasionally seen in set-theory texts. Here the restricted conclusion has been split to exhibit the major premise $x \in A$ in frontmost position.

The object-logic and application context already declare common rules for inductive sets and types in proper form, so the user only needs to take care in unusual situations.

3.2 Internal operations

Assume for the moment that only one goal and one induction rule is involved. Then the *induct* method performs operations on the proof context, rule, and goal as follows:

1. context: declare local *defs* for any defined induction variables $x = a$
2. rule: apply *insts* according to the positions in the inductive predicate $P\ x_1 \ldots x_n$
3. rule: expand *defs* in major premises (accommodate the original statement, which mentions the expanded expressions)
4. rule: consume a prefix of *facts* according to the number of major premises (the remainder is a pure induction rule $cases \Longrightarrow P\ x_1 \ldots x_n$)
5. goal: strengthen by inserting the remaining *facts*, and all *defs*
6. goal: strengthen by fixing *vars*, using rule $(\bigwedge x.\ B\ x) \Longrightarrow B\ a$ of the framework
7. goal: internalize $\bigwedge/\Longrightarrow$ into the object-logic, using $(\bigwedge x.\ Tr\ (B\ x)) \equiv Tr\ (\forall x.\ B\ x)$ and $(Tr\ A \Longrightarrow Tr\ B) \equiv Tr\ (A \longrightarrow B)$ of the framework
8. rule: unify conclusion against goal (absorb any internalized auxiliary structure, including *defs*; result is a fully-instantiated induction rule)
9. rule: carefully recover internalized $\bigwedge/\Longrightarrow$ structure in the inductive cases, using the above rules backwards
10. context: extract inductive cases from rule (enable **case** abbreviations in the proof body; cases consist of elements stemming from both the rule and the goal)
11. context: discharge *defs*
12. goal: apply the fully instantiated rule

Simultaneous induction may involve both multiple goals and multiple rules (e.g. from mutually inductive sets or types). Isar already supports simultaneous goal statements, e.g. **shows** *A* **and** *B*. The *induct* method is able to pass this structure through the inductive process, using a local version of Pure conjunction $A \ \&\ B$ internally.

Mutual induction uses packs of rules that share common inductive cases, but deviate in the major premises and conclusion. The rule preparations above essentially work the same with parallel copies of method arguments *insts* and *vars* given for each *rule*. The pack is joined by $\&$ introduction before unifying against the goal. The inner conjunctive structure of the goal is internalized using $(Tr\ A\ \&\ Tr\ B) \equiv Tr\ (A \wedge B)$. The reverse step shuffles $\&$ outermost and eliminates it via $(A\ \&\ B \Longrightarrow C) \equiv (A \Longrightarrow B \Longrightarrow C)$. Extracted cases consist of a shared context (from the rule) and separate sub-cases (from the goals).

4 Common induction patterns

We briefly review common proof patterns supported by our generic *induct* method, using the induction rule of Peano natural numbers as representative example. The proof outlines illustrate augmented contexts via literal facts, e.g. **note** $\langle \bigwedge x.\ A\ n\ x \Longrightarrow P\ n\ x \rangle$. In practice, these facts would just be used in their proper place without further notice.

4.1 Local facts and parameters

Augmenting a problem by additional facts and locally fixed variables is a bread-and-butter method in many applications. This is where unwieldy object-level $\forall$ and $\longrightarrow$ used to occur in the past. Now we are able to use primary means of the language, notably **using** in the proof text and *fixing* in the method specification.

```
lemma
  fixes n :: nat and x :: 'a
  assumes A n x
  shows P n x using ‹A n x›
proof (induct n  fixing: x)
  case 0
  note prem = ‹A 0 x›
  show P 0 x  ⟨proof⟩
next
  case (Suc n)
  note hyp = ‹⋀x. A n x ⟹ P n x›
    and prem = ‹A (Suc n) x›
  show P (Suc n) x  ⟨proof⟩
qed
```

4.2 Local definitions

Here the idea is to turn sub-expressions of the problem into a defined induction variable. This is often accompanied with fixing of auxiliary parameters in the original expression, otherwise the induction step would refer invariably to particular entities. This combination essentially expresses a partially abstracted representation of inductive expressions.

```
lemma
  fixes a :: 'a ⇒ nat
  assumes A (a x)
  shows P (a x) using ‹A (a x)›
proof (induct n = a x  fixing: x)
  case 0
  note prem = ‹A (a x)›
    and def = ‹0 = a x›
  show P (a x)  ⟨proof⟩
next
  case (Suc n)
  note hyp = ‹⋀x. A (a x) ⟹ n = a x ⟹ P (a x)›
    and prem = ‹A (a x)›
    and def = ‹Suc n = a x›
  show P (a x)  ⟨proof⟩
qed
```

Observe how the local definition $n = a\ x$ recurs in the inductive cases as $0 = a\ x$ and $Suc\ n = a\ x$, according to underlying induction rule. Here we cannot apply definitional expansions immediately, in contrast to the purer induction rule encountered in §1.3.

4.3 Simple simultaneous goals

The most basic simultaneous induction operates on several goals one-by-one, where each case refers to induction hypotheses that are duplicated according to the number of conclusions.

```
lemma
  fixes n :: nat
  shows P n and Q n
proof (induct n)
  case 0 case 1
  show P 0 ⟨proof⟩
next
  case 0 case 2
  show Q 0 ⟨proof⟩
next
  case (Suc n) case 1
  note hyps = ⟨P n⟩ ⟨Q n⟩
  show P (Suc n) ⟨proof⟩
next
  case (Suc n) case 2
  note hyps = ⟨P n⟩ ⟨Q n⟩
  show Q (Suc n) ⟨proof⟩
qed
```

Note that induction with mutual rules is usually even simpler than this, because the collection of simultaneous goals will be consumed by the given pack of rules. The resulting proof body will not require additional nesting or duplication of cases, although separate projections of the mutual induction predicates may be mentioned.

4.4 Compound simultaneous goals

The following pattern illustrates the slightly more complex situation of simultaneous goals with individual local assumptions. In compound simultaneous statements like this, local assumptions need to be included into each goal, using $\Longrightarrow$ of the Pure framework. In contrast, local parameters do not require separate $\bigwedge$ prefixes here, but may be moved into the common context of the whole statement.[2]

```
lemma
  fixes n :: nat
    and x :: 'a and y :: 'b
  shows A n x ⟹ P n x
    and B n y ⟹ Q n y
proof (induct n fixing: x y)
  case 0
  { case 1
    note prem = ⟨A 0 x⟩
    show P 0 x ⟨proof⟩ }
  { case 2
    note prem = ⟨B 0 y⟩
    show Q 0 y ⟨proof⟩ }
next
```

[2] Vacuous quantification does not impose any restriction in the Pure framework — unlike more complex versions of dependent type-theory.

```
  case (Suc n)
  note hyps = ⟨⋀x. A n x ⟹ P n x⟩ ⟨⋀y. B n y ⟹ Q n y⟩
  then have some-intermediate-result ⟨proof⟩
  { case 1
    note prem = ⟨A (Suc n) x⟩
    show P (Suc n) x ⟨proof⟩ }
  { case 2
    note prem = ⟨B (Suc n) y⟩
    show Q (Suc n) y ⟨proof⟩ }
qed
```

Here *induct* provides again nested cases with numbered sub-cases. Unlike the flattened version of §4.3, we are more careful here to share common parts of the body context. In typical applications, there could be a long intermediate proof of general consequences of the induction hypotheses, before finishing each conclusion separately.

5 Conclusion and related work

The present infrastructure for structured induction proofs has emerged from experience with applications of Isabelle/HOL, involving non-trivial reasoning about inductive types, sets and functions. Many Isar proofs in the Isabelle library benefit from this already (in post-2005 development versions). The method implementation provides sufficiently general building blocks to support non-standard variants of induction as well.

Co-induction is the logical dual of induction, and is both like and unlike standard induction in practical use. Instead of introducing a conclusion $P\ n$, co-induction eliminates a fact $\vdash P\ n$ (often written as set-membership). Our technique for *fixing* inductive parameters, essentially universal elimination applied backwards, becomes existential introduction applied forwards. Then the method invocation (*coinduct n fixing*: *x*) would correspond to the common technique in co-induction proofs to single-out certain expressions as existential parameters, which then recur in the co-induction step in eliminated form. The present *coinduct* method of Isabelle/Isar does not yet implement this elaborate scheme, but there are some examples available in *HOL*/*Library*/*Coinductive-List* of the Isabelle distribution that illustrate common proof techniques.

Nominal induction [14, 13] augments traditional structural induction by specific means to reason about datatypes involving local binders (e.g. λ-calculus, Π-calculus, functional programming languages). The corresponding induction rules involve a "freshness context" as additional parameter for the inductive predicate. This fits well into our infrastructure for producing complex induction predicates conveniently. We merely need to support an additional "*avoiding*" argument, similar to the present "*fixing*". The resulting *nominal-induct* method has been used by Urban for some key induction proofs of the POPLmark challenge[3]. The complex pattern of §4.4 reflects the structure of a key induction proof in this application.

Other interactive provers provide surprisingly little support for complex inductions.

[3] http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark

In Mizar [12, 10], induction is performed as a forward step, using basic "scheme" application. Thus the inductive cases need to be spelled out beforehand in full detail. (Isar is able to produce symbolic **case** elements due to backward refinement of a known statement and rule.) Mizar lacks specific support for compound inductive predicates.

In Coq [2], the default induct tactic passes *all* of the current proof context through the inductive process. Thus users may have to prune unwanted parts first. This corresponds to our internal treatment of $\bigwedge/\Longrightarrow$, only that Isar specifies wanted elements positively, instead of unwanted ones negatively. Coq lacks any further induction support beyond that, e.g. the user needs to encode defined inductive entities manually.

Other tactical provers, notably Isabelle/HOL [8], have traditionally avoided too elaborate induction tactics, because the exact portion of context to participate here is hard to delimit in unstructured goal states. Even worse, automated tools easily fail due to overly general induction hypotheses. Users are asked to present their problem in object-logic form, and manage the additional logical connectives by means of ad-hoc automation.

Isabelle/Isar provides a framework for checking structured proof documents, but the user needs to produce such texts manually. Proof planning techniques have recently been applied as additional means to assist proof authors in this process. The "proof-centric approach" of [4] aims to automate Isar proof construction, while preserving the ability of the author to intervene. The resulting IsaPlanner system focuses on induction and rippling, although it uses a less sophisticated version of Isar induction so far.

References

[1] D. Aspinall. Proof General: A generic tool for proof development. In *European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2000.

[2] B. Barras et al. *The Coq Proof Assistant Reference Manual, version 8*. INRIA, 2006.

[3] G. Bauer and M. Wenzel. Calculational reasoning revisited — an Isabelle/Isar experience. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2001*, LNCS 2152, 2001.

[4] L. Dixon and J. D. Fleuriot. A proof-centric approach to mathematical assistants. *Journal of Applied Logic: Special Issue on Mathematics Assistance Systems*, To appear.

[5] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Zeitschrift*, 1935.

[6] Mizar mathematical library. http://www.mizar.org/library/.

[7] T. Nipkow. Structured proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, LNCS 2646. Springer, 2003.

[8] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[9] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[10] P. Rudnicki. An overview of the MIZAR project. In *1992 Workshop on Types for Proofs and Programs*. Chalmers University of Technology, Bastad, 1992.

[11] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), 1984.

[12] A. Trybulec. Some features of the Mizar language. Presented at a workshop in Turin, 1993.

[13] C. Urban and S. Berghofer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In *International Joint Conference on Automated Reasoning, IJCAR 2006*, LNCS. Springer, 2006.

[14] C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Conference on Automated Deduction, CADE 2005*, LNCS 3632, 2005.

[15] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs'99*, LNCS 1690, 1999.

[16] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

[17] M. Wenzel and L. C. Paulson. Isabelle/Isar. In F. Wiedijk, editor, *The Seventeen Provers of the World*, LNAI 3600. Springer, 2006.

[18] F. Wiedijk. Comparing mathematical provers. In *Second International Conference on Mathematical Knowledge Management*, LNCS 2594, 2003.

[19] F. Wiedijk and M. Wenzel. A comparison of the mathematical proof languages Mizar and Isar. *Journal of Automated Reasoning*, 29(3-4), 2002.