Miscellaneous Isabelle/Isar examples

Makarius Wenzel

With contributions by Gertrud Bauer and Tobias Nipkow

September 11, 2023

Abstract

Isar offers a high-level proof (and theory) language for Isabelle. We give various examples of Isabelle/Isar proof developments, ranging from simple demonstrations of certain language features to a bit more advanced applications. The "real" applications of Isabelle/Isar are found elsewhere.

Contents

1 Structured statements within Isar proofs

theory *Structured-Statements*

imports *Main*
begin

1.1 Introduction steps

notepad
begin
⟨*proof*⟩
end

1.2 If-and-only-if

notepad
begin
⟨*proof*⟩
end

1.3 Elimination and cases

notepad
begin
⟨*proof*⟩
end

1.4 Induction

notepad
begin
⟨*proof*⟩
end

1.5 Suffices-to-show

notepad
begin
⟨*proof*⟩
end

end

2 Basic logical reasoning

theory *Basic-Logic*
imports *Main*
begin

2.1 Pure backward reasoning

In order to get a first idea of how Isabelle/Isar proof documents may look like, we consider the propositions I, K, and S. The following (rather explicit) proofs should require little extra explanations.

lemma I: $A \longrightarrow A$
⟨*proof*⟩

lemma K: $A \longrightarrow B \longrightarrow A$
⟨*proof*⟩

lemma S: $(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow A \longrightarrow C$
⟨*proof*⟩

Isar provides several ways to fine-tune the reasoning, avoiding excessive detail. Several abbreviated language elements are available, enabling the writer to express proofs in a more concise way, even without referring to any automated proof tools yet.

Concluding any (sub-)proof already involves solving any remaining goals by assumption[1]. Thus we may skip the rather vacuous body of the above proof.

lemma $A \longrightarrow A$
⟨*proof*⟩

Note that the **proof** command refers to the *rule* method (without arguments) by default. Thus it implicitly applies a single rule, as determined from the syntactic form of the statements involved. The **by** command abbreviates any proof with empty body, so the proof may be further pruned.

lemma $A \longrightarrow A$
⟨*proof*⟩

Proof by a single rule may be abbreviated as double-dot.

lemma $A \longrightarrow A$ ⟨*proof*⟩

Thus we have arrived at an adequate representation of the proof of a tautology that holds by a single standard rule.[2]

Let us also reconsider K. Its statement is composed of iterated connectives. Basic decomposition is by a single rule at a time, which is why our first version above was by nesting two proofs.

The *intro* proof method repeatedly decomposes a goal's conclusion.[3]

lemma $A \longrightarrow B \longrightarrow A$

[1] This is not a completely trivial operation, as proof by assumption may involve full higher-order unification.

[2] Apparently, the rule here is implication introduction.

[3] The dual method is *elim*, acting on a goal's premises.

⟨*proof*⟩

Again, the body may be collapsed.

lemma $A \longrightarrow B \longrightarrow A$
⟨*proof*⟩

Just like *rule*, the *intro* and *elim* proof methods pick standard structural rules, in case no explicit arguments are given. While implicit rules are usually just fine for single rule application, this may go too far with iteration. Thus in practice, *intro* and *elim* would be typically restricted to certain structures by giving a few rules only, e.g. **proof** (*intro impI allI*) to strip implications and universal quantifiers.

Such well-tuned iterated decomposition of certain structures is the prime application of *intro* and *elim*. In contrast, terminal steps that solve a goal completely are usually performed by actual automated proof methods (such as **by** *blast*.

2.2 Variations of backward vs. forward reasoning

Certainly, any proof may be performed in backward-style only. On the other hand, small steps of reasoning are often more naturally expressed in forward-style. Isar supports both backward and forward reasoning as a first-class concept. In order to demonstrate the difference, we consider several proofs of $A \wedge B \longrightarrow B \wedge A$.

The first version is purely backward.

lemma $A \wedge B \longrightarrow B \wedge A$
⟨*proof*⟩

Above, the projection rules *conjunct1* / *conjunct2* had to be named explicitly, since the goals B and A did not provide any structural clue. This may be avoided using **from** to focus on the $A \wedge B$ assumption as the current facts, enabling the use of double-dot proofs. Note that **from** already does forward-chaining, involving the *conjE* rule here.

lemma $A \wedge B \longrightarrow B \wedge A$
⟨*proof*⟩

In the next version, we move the forward step one level upwards. Forward-chaining from the most recent facts is indicated by the **then** command. Thus the proof of $B \wedge A$ from $A \wedge B$ actually becomes an elimination, rather than an introduction. The resulting proof structure directly corresponds to that of the *conjE* rule, including the repeated goal proposition that is abbreviated as *?thesis* below.

lemma $A \wedge B \longrightarrow B \wedge A$
⟨*proof*⟩

In the subsequent version we flatten the structure of the main body by doing forward reasoning all the time. Only the outermost decomposition step is left as backward.

lemma $A \wedge B \longrightarrow B \wedge A$
⟨*proof*⟩

We can still push forward-reasoning a bit further, even at the risk of getting ridiculous. Note that we force the initial proof step to do nothing here, by referring to the $-$ proof method.

lemma $A \wedge B \longrightarrow B \wedge A$
⟨*proof*⟩

With these examples we have shifted through a whole range from purely backward to purely forward reasoning. Apparently, in the extreme ends we get slightly ill-structured proofs, which also require much explicit naming of either rules (backward) or local facts (forward).

The general lesson learned here is that good proof style would achieve just the *right* balance of top-down backward decomposition, and bottom-up forward composition. In general, there is no single best way to arrange some pieces of formal reasoning, of course. Depending on the actual applications, the intended audience etc., rules (and methods) on the one hand vs. facts on the other hand have to be emphasized in an appropriate way. This requires the proof writer to develop good taste, and some practice, of course.

For our example the most appropriate way of reasoning is probably the middle one, with conjunction introduction done after elimination.

lemma $A \wedge B \longrightarrow B \wedge A$
⟨*proof*⟩

2.3 A few examples from "Introduction to Isabelle"

We rephrase some of the basic reasoning examples of [4], using HOL rather than FOL.

2.3.1 A propositional proof

We consider the proposition $P \vee P \longrightarrow P$. The proof below involves forward-chaining from $P \vee P$, followed by an explicit case-analysis on the two *identical* cases.

lemma $P \vee P \longrightarrow P$
⟨*proof*⟩

Case splits are *not* hardwired into the Isar language as a special feature. The **next** command used to separate the cases above is just a short form of managing block structure.

In general, applying proof methods may split up a goal into separate "cases", i.e. new subgoals with individual local assumptions. The corresponding proof text typically mimics this by establishing results in appropriate contexts, separated by blocks.

In order to avoid too much explicit parentheses, the Isar system implicitly opens an additional block for any new goal, the **next** statement then closes one block level, opening a new one. The resulting behaviour is what one would expect from separating cases, only that it is more flexible. E.g. an induction base case (which does not introduce local assumptions) would *not* require **next** to separate the subsequent step case.

In our example the situation is even simpler, since the two cases actually coincide. Consequently the proof may be rephrased as follows.

lemma $P \vee P \longrightarrow P$
⟨*proof*⟩

Again, the rather vacuous body of the proof may be collapsed. Thus the case analysis degenerates into two assumption steps, which are implicitly performed when concluding the single rule step of the double-dot proof as follows.

lemma $P \vee P \longrightarrow P$
⟨*proof*⟩

2.3.2 A quantifier proof

To illustrate quantifier reasoning, let us prove $(\exists\, x.\ P\ (f\ x)) \longrightarrow (\exists\, y.\ P\ y)$. Informally, this holds because any a with $P\ (f\ a)$ may be taken as a witness for the second existential statement.

The first proof is rather verbose, exhibiting quite a lot of (redundant) detail. It gives explicit rules, even with some instantiation. Furthermore, we encounter two new language elements: the **fix** command augments the context by some new "arbitrary, but fixed" element; the **is** annotation binds term abbreviations by higher-order pattern matching.

lemma $(\exists\, x.\ P\ (f\ x)) \longrightarrow (\exists\, y.\ P\ y)$
⟨*proof*⟩

While explicit rule instantiation may occasionally improve readability of certain aspects of reasoning, it is usually quite redundant. Above, the basic proof outline gives already enough structural clues for the system to infer both the rules and their instances (by higher-order unification). Thus we may as well prune the text as follows.

lemma $(\exists\, x.\ P\ (f\ x)) \longrightarrow (\exists\, y.\ P\ y)$
⟨*proof*⟩

Explicit $\exists$-elimination as seen above can become quite cumbersome in practice. The derived Isar language element "**obtain**" provides a more handsome way to do generalized existence reasoning.

lemma $(\exists\, x.\ P\ (f\ x)) \longrightarrow (\exists\, y.\ P\ y)$
⟨*proof*⟩

Technically, **obtain** is similar to **fix** and **assume** together with a soundness proof of the elimination involved. Thus it behaves similar to any other forward proof element. Also note that due to the nature of general existence reasoning involved here, any result exported from the context of an **obtain** statement may *not* refer to the parameters introduced there.

2.3.3 Deriving rules in Isabelle

We derive the conjunction elimination rule from the corresponding projections. The proof is quite straight-forward, since Isabelle/Isar supports non-atomic goals and assumptions fully transparently.

theorem *conjE*: $A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

end

3 Correctness of a simple expression compiler

theory *Expr-Compiler*
  imports *Main*
begin

This is a (rather trivial) example of program verification. We model a compiler for translating expressions to stack machine instructions, and prove its correctness wrt. some evaluation semantics.

3.1 Binary operations

Binary operations are just functions over some type of values. This is both for abstract syntax and semantics, i.e. we use a "shallow embedding" here.

type-synonym *'val binop* = *'val* ⇒ *'val* ⇒ *'val*

3.2 Expressions

The language of expressions is defined as an inductive type, consisting of variables, constants, and binary operations on expressions.

datatype (*dead 'adr*, *dead 'val*) *expr* =
    Variable 'adr

```
| Constant 'val
| Binop 'val binop ('adr, 'val) expr ('adr, 'val) expr
```

Evaluation (wrt. some environment of variable assignments) is defined by primitive recursion over the structure of expressions.

```
primrec eval :: ('adr, 'val) expr ⇒ ('adr ⇒ 'val) ⇒ 'val
  where
    eval (Variable x) env = env x
  | eval (Constant c) env = c
  | eval (Binop f e1 e2) env = f (eval e1 env) (eval e2 env)
```

3.3 Machine

Next we model a simple stack machine, with three instructions.

```
datatype (dead 'adr, dead 'val) instr =
    Const 'val
  | Load 'adr
  | Apply 'val binop
```

Execution of a list of stack machine instructions is easily defined as follows.

```
primrec exec :: (('adr, 'val) instr) list ⇒ 'val list ⇒ ('adr ⇒ 'val) ⇒ 'val list
  where
    exec [] stack env = stack
  | exec (instr # instrs) stack env =
      (case instr of
        Const c ⇒ exec instrs (c # stack) env
      | Load x ⇒ exec instrs (env x # stack) env
      | Apply f ⇒ exec instrs (f (hd stack) (hd (tl stack)) # (tl (tl stack))) env)

definition execute :: (('adr, 'val) instr) list ⇒ ('adr ⇒ 'val) ⇒ 'val
  where execute instrs env = hd (exec instrs [] env)
```

3.4 Compiler

We are ready to define the compilation function of expressions to lists of stack machine instructions.

```
primrec compile :: ('adr, 'val) expr ⇒ (('adr, 'val) instr) list
  where
    compile (Variable x) = [Load x]
  | compile (Constant c) = [Const c]
  | compile (Binop f e1 e2) = compile e2 @ compile e1 @ [Apply f]
```

The main result of this development is the correctness theorem for *compile*. We first establish a lemma about *exec* and list append.

```
lemma exec-append:
  exec (xs @ ys) stack env =
    exec ys (exec xs stack env) env
```

⟨*proof*⟩

theorem *correctness*: *execute* (*compile e*) *env* = *eval e env*
⟨*proof*⟩

In the proofs above, the *simp* method does quite a lot of work behind the scenes (mostly "functional program execution"). Subsequently, the same reasoning is elaborated in detail — at most one recursive function definition is used at a time. Thus we get a better idea of what is actually going on.

lemma *exec-append'*:
exec (*xs* @ *ys*) *stack env* = *exec ys* (*exec xs stack env*) *env*
⟨*proof*⟩

theorem *correctness'*: *execute* (*compile e*) *env* = *eval e env*
⟨*proof*⟩

end

4 Fib and Gcd commute

theory *Fibonacci*
imports *HOL−Computational-Algebra.Primes*
begin[4]

4.1 Fibonacci numbers

fun *fib* :: *nat* ⇒ *nat*
where
fib 0 = *0*
| *fib* (*Suc 0*) = *1*
| *fib* (*Suc* (*Suc x*)) = *fib x* + *fib* (*Suc x*)

lemma [*simp*]: *fib* (*Suc n*) > *0*
⟨*proof*⟩

Alternative induction rule.

theorem *fib-induct*: *P 0* ⟹ *P 1* ⟹ (⋀*n*. *P* (*n* + *1*) ⟹ *P n* ⟹ *P* (*n* + *2*)) ⟹ *P n*
for *n* :: *nat*
⟨*proof*⟩

4.2 Fib and gcd commute

A few laws taken from [1].

[4]Isar version by Gertrud Bauer. Original tactic script by Larry Paulson. A few proofs of laws taken from [1].

lemma *fib-add*: $fib\ (n + k + 1) = fib\ (k + 1) * fib\ (n + 1) + fib\ k * fib\ n$
  (**is** *?P n*)
  — see [1, page 280]
⟨*proof*⟩

lemma *coprime-fib-Suc*: $coprime\ (fib\ n)\ (fib\ (n + 1))$
  (**is** *?P n*)
⟨*proof*⟩

lemma *gcd-mult-add*: $(0::nat) < n \Longrightarrow gcd\ (n * k + m)\ n = gcd\ m\ n$
⟨*proof*⟩

lemma *gcd-fib-add*: $gcd\ (fib\ m)\ (fib\ (n + m)) = gcd\ (fib\ m)\ (fib\ n)$
⟨*proof*⟩

lemma *gcd-fib-diff*: $gcd\ (fib\ m)\ (fib\ (n - m)) = gcd\ (fib\ m)\ (fib\ n)$ **if** $m \leq n$
⟨*proof*⟩

lemma *gcd-fib-mod*: $gcd\ (fib\ m)\ (fib\ (n\ mod\ m)) = gcd\ (fib\ m)\ (fib\ n)$ **if** $0 < m$
⟨*proof*⟩

theorem *fib-gcd*: $fib\ (gcd\ m\ n) = gcd\ (fib\ m)\ (fib\ n)$
  (**is** *?P m n*)
⟨*proof*⟩

end

5 Basic group theory

theory *Group*
  imports *Main*
begin

5.1 Groups and calculational reasoning

Groups over signature $(* :: \alpha \Rightarrow \alpha \Rightarrow \alpha,\ 1 :: \alpha,\ inverse :: \alpha \Rightarrow \alpha)$ are defined as an axiomatic type class as follows. Note that the parent classes *times*, *one*, *inverse* is provided by the basic HOL theory.

class *group* = *times* + *one* + *inverse* +
  assumes *group-assoc*: $(x * y) * z = x * (y * z)$
    and *group-left-one*: $1 * x = x$
    and *group-left-inverse*: $inverse\ x * x = 1$

The group axioms only state the properties of left one and inverse, the right versions may be derived as follows.

theorem (**in** *group*) *group-right-inverse*: $x * inverse\ x = 1$
⟨*proof*⟩

With *group-right-inverse* already available, *group-right-one* is now established much easier.

theorem (**in** *group*) *group-right-one*: $x * 1 = x$
⟨*proof*⟩

The calculational proof style above follows typical presentations given in any introductory course on algebra. The basic technique is to form a transitive chain of equations, which in turn are established by simplifying with appropriate rules. The low-level logical details of equational reasoning are left implicit.

Note that "..." is just a special term variable that is bound automatically to the argument[5] of the last fact achieved by any local assumption or proven statement. In contrast to *?thesis*, the "..." variable is bound *after* the proof is finished.

There are only two separate Isar language elements for calculational proofs: "**also**" for initial or intermediate calculational steps, and "**finally**" for exhibiting the result of a calculation. These constructs are not hardwired into Isabelle/Isar, but defined on top of the basic Isar/VM interpreter. Expanding the **also** and **finally** derived language elements, calculations may be simulated by hand as demonstrated below.

theorem (**in** *group*) $x * 1 = x$
⟨*proof*⟩

Note that this scheme of calculations is not restricted to plain transitivity. Rules like anti-symmetry, or even forward and backward substitution work as well. For the actual implementation of **also** and **finally**, Isabelle/Isar maintains separate context information of "transitivity" rules. Rule selection takes place automatically by higher-order unification.

5.2 Groups as monoids

Monoids over signature $(* :: \alpha \Rightarrow \alpha \Rightarrow \alpha,\ 1 :: \alpha)$ are defined like this.

class *monoid* = *times* + *one* +
  assumes *monoid-assoc*: $(x * y) * z = x * (y * z)$
    and *monoid-left-one*: $1 * x = x$
    and *monoid-right-one*: $x * 1 = x$

Groups are *not* yet monoids directly from the definition. For monoids, *right-one* had to be included as an axiom, but for groups both *right-one* and *right-inverse* are derivable from the other axioms. With *group-right-one* derived as a theorem of group theory (see $?x * (1::?'a) = ?x$), we may still instantiate *group* $\subseteq$ *monoid* properly as follows.

[5] The argument of a curried infix expression happens to be its right-hand side.

instance *group* $\subseteq$ *monoid*
⟨*proof*⟩

The **instance** command actually is a version of **theorem**, setting up a goal that reflects the intended class relation (or type constructor arity). Thus any Isar proof language element may be involved to establish this statement. When concluding the proof, the result is transformed into the intended type signature extension behind the scenes.

5.3 More theorems of group theory

The one element is already uniquely determined by preserving an *arbitrary* group element.

theorem (**in** *group*) *group-one-equality*:
assumes *eq*: $e * x = x$
shows $1 = e$
⟨*proof*⟩

Likewise, the inverse is already determined by the cancel property.

theorem (**in** *group*) *group-inverse-equality*:
assumes *eq*: $x' * x = 1$
shows $inverse\ x = x'$
⟨*proof*⟩

The inverse operation has some further characteristic properties.

theorem (**in** *group*) *group-inverse-times*: $inverse\ (x * y) = inverse\ y * inverse\ x$
⟨*proof*⟩

theorem (**in** *group*) *inverse-inverse*: $inverse\ (inverse\ x) = x$
⟨*proof*⟩

theorem (**in** *group*) *inverse-inject*:
assumes *eq*: $inverse\ x = inverse\ y$
shows $x = y$
⟨*proof*⟩

end

6 Some algebraic identities derived from group axioms – theory context version

theory *Group-Context*
imports *Main*
begin

hypothetical group axiomatization

context
  fixes *prod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\odot$ *70*)
    and *one* :: $'a$
    and *inverse* :: $'a \Rightarrow 'a$
  assumes *assoc*: $(x \odot y) \odot z = x \odot (y \odot z)$
    and *left-one*: $one \odot x = x$
    and *left-inverse*: $inverse\ x \odot x = one$
begin

some consequences

lemma *right-inverse*: $x \odot inverse\ x = one$
⟨*proof*⟩

lemma *right-one*: $x \odot one = x$
⟨*proof*⟩

lemma *one-equality*:
  assumes *eq*: $e \odot x = x$
  shows $one = e$
⟨*proof*⟩

lemma *inverse-equality*:
  assumes *eq*: $x' \odot x = one$
  shows $inverse\ x = x'$
⟨*proof*⟩

end

end

7 Some algebraic identities derived from group axioms – proof notepad version

theory *Group-Notepad*
  imports *Main*
begin

notepad
begin

hypothetical group axiomatization

  ⟨*proof*⟩

end

end

8 Hoare Logic

theory *Hoare*
imports *HOL−Hoare.Hoare-Tac*
begin

8.1 Abstract syntax and semantics

The following abstract syntax and semantics of Hoare Logic over `WHILE` programs closely follows the existing tradition in Isabelle/HOL of formalizing the presentation given in [8, §6]. See also `~~/src/HOL/Hoare` and [3].

type-synonym *'a bexp* = *'a set*
type-synonym *'a assn* = *'a set*
type-synonym *'a var* = *'a ⇒ nat*

datatype *'a com* =
  Basic 'a ⇒ 'a
| *Seq 'a com 'a com* ((-;/ -) [*60*, *61*] *60*)
| *Cond 'a bexp 'a com 'a com*
| *While 'a bexp 'a assn 'a var 'a com*

abbreviation *Skip* (*SKIP*)
  where *SKIP* ≡ *Basic id*

type-synonym *'a sem* = *'a ⇒ 'a ⇒ bool*

primrec *iter* :: *nat ⇒ 'a bexp ⇒ 'a sem ⇒ 'a sem*
  where
    iter 0 b S s s′ ⟷ *s* ∉ *b* ∧ *s* = *s′*
  | *iter* (*Suc n*) *b S s s′* ⟷ *s* ∈ *b* ∧ (∃ *s″*. *S s s″* ∧ *iter n b S s″ s′*)

primrec *Sem* :: *'a com ⇒ 'a sem*
  where
    Sem (*Basic f*) *s s′* ⟷ *s′* = *f s*
  | *Sem* (*c1*; *c2*) *s s′* ⟷ (∃ *s″*. *Sem c1 s s″* ∧ *Sem c2 s″ s′*)
  | *Sem* (*Cond b c1 c2*) *s s′* ⟷ (*if s* ∈ *b then Sem c1 s s′ else Sem c2 s s′*)
  | *Sem* (*While b x y c*) *s s′* ⟷ (∃ *n*. *iter n b* (*Sem c*) *s s′*)

definition *Valid* :: *'a bexp ⇒ 'a com ⇒ 'a bexp ⇒ bool* ((*3*⊢ -/ (*2*-)/ -) [*100*, *55*, *100*] *50*)
  where ⊢ *P c Q* ⟷ (∀ *s s′*. *Sem c s s′* ⟶ *s* ∈ *P* ⟶ *s′* ∈ *Q*)

lemma *ValidI* [*intro?*]: (⋀*s s′*. *Sem c s s′* ⟹ *s* ∈ *P* ⟹ *s′* ∈ *Q*) ⟹ ⊢ *P c Q*
  ⟨*proof*⟩

lemma *ValidD* [*dest?*]: ⊢ *P c Q* ⟹ *Sem c s s′* ⟹ *s* ∈ *P* ⟹ *s′* ∈ *Q*
  ⟨*proof*⟩

8.2 Primitive Hoare rules

From the semantics defined above, we derive the standard set of primitive Hoare rules; e.g. see [8, §6]. Usually, variant forms of these rules are applied in actual proof, see also §8.4 and §8.5.

The *basic* rule represents any kind of atomic access to the state space. This subsumes the common rules of *skip* and *assign*, as formulated in §8.4.

theorem *basic*: $\vdash \{s.\ f\ s \in P\}\ (\mathit{Basic}\ f)\ P$
⟨*proof*⟩

The rules for sequential commands and semantic consequences are established in a straight forward manner as follows.

theorem *seq*: $\vdash P\ c1\ Q \Longrightarrow \vdash Q\ c2\ R \Longrightarrow \vdash P\ (c1;\ c2)\ R$
⟨*proof*⟩

theorem *conseq*: $P' \subseteq P \Longrightarrow \vdash P\ c\ Q \Longrightarrow Q \subseteq Q' \Longrightarrow \vdash P'\ c\ Q'$
⟨*proof*⟩

The rule for conditional commands is directly reflected by the corresponding semantics; in the proof we just have to look closely which cases apply.

theorem *cond*:
  assumes *case-b*: $\vdash (P \cap b)\ c1\ Q$
    and *case-nb*: $\vdash (P \cap -b)\ c2\ Q$
  shows $\vdash P\ (\mathit{Cond}\ b\ c1\ c2)\ Q$
⟨*proof*⟩

The *while* rule is slightly less trivial — it is the only one based on recursion, which is expressed in the semantics by a Kleene-style least fixed-point construction. The auxiliary statement below, which is by induction on the number of iterations is the main point to be proven; the rest is by routine application of the semantics of `WHILE`.

theorem *while*:
  assumes *body*: $\vdash (P \cap b)\ c\ P$
  shows $\vdash P\ (\mathit{While}\ b\ X\ Y\ c)\ (P \cap -b)$
⟨*proof*⟩

8.3 Concrete syntax for assertions

We now introduce concrete syntax for describing commands (with embedded expressions) and assertions. The basic technique is that of semantic "quote-antiquote". A *quotation* is a syntactic entity delimited by an implicit abstraction, say over the state space. An *antiquotation* is a marked expression within a quotation that refers the implicit argument; a typical antiquotation would select (or even update) components from the state.

We will see some examples later in the concrete rules and applications.

The following specification of syntax and translations is for Isabelle experts only; feel free to ignore it.

While the first part is still a somewhat intelligible specification of the concrete syntactic representation of our Hoare language, the actual "ML drivers" is quite involved. Just note that the we re-use the basic quote/antiquote translations as already defined in Isabelle/Pure (see `Syntax_Trans.quote_tr`, and `Syntax_Trans.quote_tr'`,).

syntax
-quote :: *'b ⇒ ('a ⇒ 'b)*
-antiquote :: *('a ⇒ 'b) ⇒ 'b* (*´- [1000] 1000*)
-Subst :: *'a bexp ⇒ 'b ⇒ idt ⇒ 'a bexp* (*-[-'/´-] [1000] 999*)
-Assert :: *'a ⇒ 'a set* ((*{|-|}*) *[0] 1000*)
-Assign :: *idt ⇒ 'b ⇒ 'a com* ((*´- :=/ -*) *[70, 65] 61*)
-Cond :: *'a bexp ⇒ 'a com ⇒ 'a com ⇒ 'a com*
(*(0IF -/ THEN -/ ELSE -/ FI) [0, 0, 0] 61*)
-While-inv :: *'a bexp ⇒ 'a assn ⇒ 'a com ⇒ 'a com*
(*(0WHILE -/ INV - //DO - /OD) [0, 0, 0] 61*)
-While :: *'a bexp ⇒ 'a com ⇒ 'a com* (*(0WHILE - //DO - /OD) [0, 0] 61*)

translations
{|b|} ⇀ CONST Collect (-quote b)
B [a/´x] ⇀ {|´(-update-name x (λ-. a)) ∈ B|}
´x := a ⇀ CONST Basic (-quote (´(-update-name x (λ-. a))))
IF b THEN c1 ELSE c2 FI ⇀ CONST Cond {|b|} c1 c2
WHILE b INV i DO c OD ⇀ CONST While {|b|} i (λ-. 0) c
WHILE b DO c OD ⇌ WHILE b INV CONST undefined DO c OD

⟨*ML*⟩

As usual in Isabelle syntax translations, the part for printing is more complicated — we cannot express parts as macro rules as above. Don't look here, unless you have to do similar things for yourself.

⟨*ML*⟩

8.4 Rules for single-step proof

We are now ready to introduce a set of Hoare rules to be used in single-step structured proofs in Isabelle/Isar. We refer to the concrete syntax introduce above.

Assertions of Hoare Logic may be manipulated in calculational proofs, with the inclusion expressed in terms of sets or predicates. Reversed order is supported as well.

lemma [*trans*]: $\vdash P\ c\ Q \Longrightarrow P' \subseteq P \Longrightarrow \vdash P'\ c\ Q$
⟨*proof*⟩
lemma [*trans*] : $P' \subseteq P \Longrightarrow \vdash P\ c\ Q \Longrightarrow \vdash P'\ c\ Q$

⟨*proof*⟩

lemma [*trans*]: $Q \subseteq Q' \Longrightarrow \vdash P\ c\ Q \Longrightarrow \vdash P\ c\ Q'$
⟨*proof*⟩
lemma [*trans*]: $\vdash P\ c\ Q \Longrightarrow Q \subseteq Q' \Longrightarrow \vdash P\ c\ Q'$
⟨*proof*⟩

lemma [*trans*]:
$\vdash \{\!|\ ´P\ |\!\}\ c\ Q \Longrightarrow (\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \vdash \{\!|\ ´P'\ |\!\}\ c\ Q$
⟨*proof*⟩
lemma [*trans*]:
$(\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \vdash \{\!|\ ´P\ |\!\}\ c\ Q \Longrightarrow \vdash \{\!|\ ´P'\ |\!\}\ c\ Q$
⟨*proof*⟩

lemma [*trans*]:
$\vdash P\ c\ \{\!|\ ´Q\ |\!\} \Longrightarrow (\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \vdash P\ c\ \{\!|\ ´Q'\ |\!\}$
⟨*proof*⟩
lemma [*trans*]:
$(\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \vdash P\ c\ \{\!|\ ´Q\ |\!\} \Longrightarrow \vdash P\ c\ \{\!|\ ´Q'\ |\!\}$
⟨*proof*⟩

Identity and basic assignments.[6]

lemma *skip* [*intro?*]: $\vdash P\ SKIP\ P$
⟨*proof*⟩

lemma *assign*: $\vdash P\ [´a/´x::'a]\ ´x := ´a\ P$
⟨*proof*⟩

Note that above formulation of assignment corresponds to our preferred way to model state spaces, using (extensible) record types in HOL [2]. For any record field x, Isabelle/HOL provides a functions x (selector) and x-*update* (update). Above, there is only a place-holder appearing for the latter kind of function: due to concrete syntax $´x := ´a$ also contains x-*update*.[7]

Sequential composition — normalizing with associativity achieves proper of chunks of code verified separately.

lemmas [*trans*, *intro?*] = *seq*

lemma *seq-assoc* [*simp*]: $\vdash P\ c1;(c2;c3)\ Q \longleftrightarrow \vdash P\ (c1;c2);c3\ Q$
⟨*proof*⟩

Conditional statements.

lemmas [*trans*, *intro?*] = *cond*

[6] The *hoare* method introduced in §8.5 is able to provide proper instances for any number of basic assignments, without producing additional verification conditions.

[7] Note that due to the external nature of HOL record fields, we could not even state a general theorem relating selector and update functions (if this were required here); this would only work for any particular instance of record fields introduced so far.

lemma [*trans*, *intro?*]:
$\vdash \{\!|\ ´P \wedge ´b\ |\!\}\ c1\ Q$
$\Longrightarrow \vdash \{\!|\ ´P \wedge \neg\ ´b\ |\!\}\ c2\ Q$
$\Longrightarrow \vdash \{\!|\ ´P |\!\}\ IF\ ´b\ THEN\ c1\ ELSE\ c2\ FI\ Q$
⟨*proof*⟩

While statements — with optional invariant.

lemma [*intro?*]: $\vdash (P \cap b)\ c\ P \Longrightarrow \vdash P\ (While\ b\ P\ V\ c)\ (P \cap -b)$
⟨*proof*⟩

lemma [*intro?*]: $\vdash (P \cap b)\ c\ P \Longrightarrow \vdash P\ (While\ b\ undefined\ V\ c)\ (P \cap -b)$
⟨*proof*⟩

lemma [*intro?*]:
$\vdash \{\!|\ ´P \wedge ´b\ |\!\}\ c\ \{\!|\ ´P |\!\}$
$\Longrightarrow \vdash \{\!|\ ´P |\!\}\ WHILE\ ´b\ INV\ \{\!|\ ´P |\!\}\ DO\ c\ OD\ \{\!|\ ´P \wedge \neg\ ´b\ |\!\}$
⟨*proof*⟩

lemma [*intro?*]:
$\vdash \{\!|\ ´P \wedge ´b\ |\!\}\ c\ \{\!|\ ´P |\!\}$
$\Longrightarrow \vdash \{\!|\ ´P |\!\}\ WHILE\ ´b\ DO\ c\ OD\ \{\!|\ ´P \wedge \neg\ ´b\ |\!\}$
⟨*proof*⟩

8.5 Verification conditions

We now load the *original* ML file for proof scripts and tactic definition for the Hoare Verification Condition Generator (see `~~/src/HOL/Hoare`). As far as we are concerned here, the result is a proof method *hoare*, which may be applied to a Hoare Logic assertion to extract purely logical verification conditions. It is important to note that the method requires `WHILE` loops to be fully annotated with invariants beforehand. Furthermore, only *concrete* pieces of code are handled — the underlying tactic fails ungracefully if supplied with meta-variables or parameters, for example.

lemma *SkipRule*: $p \subseteq q \Longrightarrow Valid\ p\ (Basic\ id)\ q$
⟨*proof*⟩

lemma *BasicRule*: $p \subseteq \{s.\ f\ s \in q\} \Longrightarrow Valid\ p\ (Basic\ f)\ q$
⟨*proof*⟩

lemma *SeqRule*: $Valid\ P\ c1\ Q \Longrightarrow Valid\ Q\ c2\ R \Longrightarrow Valid\ P\ (c1;c2)\ R$
⟨*proof*⟩

lemma *CondRule*:
$p \subseteq \{s.\ (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}$
$\Longrightarrow Valid\ w\ c1\ q \Longrightarrow Valid\ w'\ c2\ q \Longrightarrow Valid\ p\ (Cond\ b\ c1\ c2)\ q$
⟨*proof*⟩

lemma *iter-aux*:
$\forall s\ s'.\ Sem\ c\ s\ s' \longrightarrow s \in I \wedge s \in b \longrightarrow s' \in I \Longrightarrow$
$(\bigwedge s\ s'.\ s \in I \Longrightarrow iter\ n\ b\ (Sem\ c)\ s\ s' \Longrightarrow s' \in I \wedge s' \notin b)$
⟨*proof*⟩

lemma *WhileRule*:
$p \subseteq i \Longrightarrow Valid\ (i \cap b)\ c\ i \Longrightarrow i \cap (-b) \subseteq q \Longrightarrow Valid\ p\ (While\ b\ i\ v\ c)\ q$
⟨*proof*⟩

declare *BasicRule* [*Hoare-Tac.BasicRule*]
and *SkipRule* [*Hoare-Tac.SkipRule*]
and *SeqRule* [*Hoare-Tac.SeqRule*]
and *CondRule* [*Hoare-Tac.CondRule*]
and *WhileRule* [*Hoare-Tac.WhileRule*]

⟨*ML*⟩

end

9 Using Hoare Logic

theory *Hoare-Ex*
imports *Hoare*
begin

9.1 State spaces

First of all we provide a store of program variables that occur in any of the programs considered later. Slightly unexpected things may happen when attempting to work with undeclared variables.

record *vars* =
I :: *nat*
M :: *nat*
N :: *nat*
S :: *nat*

While all of our variables happen to have the same type, nothing would prevent us from working with many-sorted programs as well, or even polymorphic ones. Also note that Isabelle/HOL's extensible record types even provides simple means to extend the state space later.

9.2 Basic examples

We look at few trivialities involving assignment and sequential composition, in order to get an idea of how to work with our formulation of Hoare Logic.

Using the basic *assign* rule directly is a bit cumbersome.

lemma $\vdash \{\!|\,\acute{}(N\text{-}update\ (\lambda\text{-}.\ (2 * \acute{}N)))\in \{\!|\,\acute{}N = 10\,|\!\}\,|\!\}\ \acute{}N := 2 * \acute{}N\ \{\!|\,\acute{}N = 10\,|\!\}$
⟨*proof*⟩

Certainly we want the state modification already done, e.g. by simplification. The *hoare* method performs the basic state update for us; we may apply the Simplifier afterwards to achieve "obvious" consequences as well.

lemma $\vdash \{\!|\,True\,|\!\}\ \acute{}N := 10\ \{\!|\,\acute{}N = 10\,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\,2 * \acute{}N = 10\,|\!\}\ \acute{}N := 2 * \acute{}N\ \{\!|\,\acute{}N = 10\,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\,\acute{}N = 5\,|\!\}\ \acute{}N := 2 * \acute{}N\ \{\!|\,\acute{}N = 10\,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\,\acute{}N + 1 = a + 1\,|\!\}\ \acute{}N := \acute{}N + 1\ \{\!|\,\acute{}N = a + 1\,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\,\acute{}N = a\,|\!\}\ \acute{}N := \acute{}N + 1\ \{\!|\,\acute{}N = a + 1\,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\,a = a \wedge b = b\,|\!\}\ \acute{}M := a;\ \acute{}N := b\ \{\!|\,\acute{}M = a \wedge \acute{}N = b\,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\,True\,|\!\}\ \acute{}M := a;\ \acute{}N := b\ \{\!|\,\acute{}M = a \wedge \acute{}N = b\,|\!\}$
⟨*proof*⟩

lemma
$\vdash \{\!|\,\acute{}M = a \wedge \acute{}N = b\,|\!\}$
$\acute{}I := \acute{}M;\ \acute{}M := \acute{}N;\ \acute{}N := \acute{}I$
$\{\!|\,\acute{}M = b \wedge \acute{}N = a\,|\!\}$
⟨*proof*⟩

It is important to note that statements like the following one can only be proven for each individual program variable. Due to the extra-logical nature of record fields, we cannot formulate a theorem relating record selectors and updates schematically.

lemma $\vdash \{\!|\,\acute{}N = a\,|\!\}\ \acute{}N := \acute{}N\ \{\!|\,\acute{}N = a\,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\,\acute{}x = a\,|\!\}\ \acute{}x := \acute{}x\ \{\!|\,\acute{}x = a\,|\!\}$
⟨*proof*⟩

lemma
Valid $\{s.\ x\ s = a\}$ (*Basic* $(\lambda s.\ x\text{-}update\ (x\ s)\ s)$) $\{s.\ x\ s = n\}$
— same statement without concrete syntax
⟨*proof*⟩

In the following assignments we make use of the consequence rule in order to achieve the intended precondition. Certainly, the *hoare* method is able to handle this case, too.

lemma $\vdash \{\!|\, ´M = ´N \,|\!\}\ ´M := ´M + 1\ \{\!|\, ´M \neq ´N \,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\, ´M = ´N \,|\!\}\ ´M := ´M + 1\ \{\!|\, ´M \neq ´N \,|\!\}$
⟨*proof*⟩

lemma $\vdash \{\!|\, ´M = ´N \,|\!\}\ ´M := ´M + 1\ \{\!|\, ´M \neq ´N \,|\!\}$
⟨*proof*⟩

9.3 Multiplication by addition

We now do some basic examples of actual `WHILE` programs. This one is a loop for calculating the product of two natural numbers, by iterated addition. We first give detailed structured proof based on single-step Hoare rules.

lemma
$\vdash \{\!|\, ´M = 0 \wedge ´S = 0 \,|\!\}$
WHILE $´M \neq a$
DO $´S := ´S + b;\ ´M := ´M + 1$ *OD*
$\{\!|\, ´S = a * b \,|\!\}$
⟨*proof*⟩

The subsequent version of the proof applies the *hoare* method to reduce the Hoare statement to a purely logical problem that can be solved fully automatically. Note that we have to specify the `WHILE` loop invariant in the original statement.

lemma
$\vdash \{\!|\, ´M = 0 \wedge ´S = 0 \,|\!\}$
WHILE $´M \neq a$
INV $\{\!|\, ´S = ´M * b \,|\!\}$
DO $´S := ´S + b;\ ´M := ´M + 1$ *OD*
$\{\!|\, ´S = a * b \,|\!\}$
⟨*proof*⟩

9.4 Summing natural numbers

We verify an imperative program to sum natural numbers up to a given limit. First some functional definition for proper specification of the problem.

The following proof is quite explicit in the individual steps taken, with the *hoare* method only applied locally to take care of assignment and sequential composition. Note that we express intermediate proof obligation in pure logic, without referring to the state space.

theorem

$\vdash \{\!|\mathit{True}|\!\}$
$\acute{S} := 0;\ \acute{I} := 1;$
$\mathit{WHILE}\ \acute{I} \neq n$
DO
$\acute{S} := \acute{S} + \acute{I};$
$\acute{I} := \acute{I} + 1$
OD
$\{\!|\acute{S} = (\sum j{<}n.\ j)|\!\}$
(**is** $\vdash$ - (-; *?while*) -)
⟨*proof*⟩

The next version uses the *hoare* method, while still explaining the resulting proof obligations in an abstract, structured manner.

theorem
$\vdash \{\!|\mathit{True}|\!\}$
$\acute{S} := 0;\ \acute{I} := 1;$
$\mathit{WHILE}\ \acute{I} \neq n$
$\mathit{INV}\ \{\!|\acute{S} = (\sum j{<}\acute{I}.\ j)|\!\}$
DO
$\acute{S} := \acute{S} + \acute{I};$
$\acute{I} := \acute{I} + 1$
OD
$\{\!|\acute{S} = (\sum j{<}n.\ j)|\!\}$
⟨*proof*⟩

Certainly, this proof may be done fully automatic as well, provided that the invariant is given beforehand.

theorem
$\vdash \{\!|\mathit{True}|\!\}$
$\acute{S} := 0;\ \acute{I} := 1;$
$\mathit{WHILE}\ \acute{I} \neq n$
$\mathit{INV}\ \{\!|\acute{S} = (\sum j{<}\acute{I}.\ j)|\!\}$
DO
$\acute{S} := \acute{S} + \acute{I};$
$\acute{I} := \acute{I} + 1$
OD
$\{\!|\acute{S} = (\sum j{<}n.\ j)|\!\}$
⟨*proof*⟩

9.5 Time

A simple embedding of time in Hoare logic: function *timeit* inserts an extra variable to keep track of the elapsed time.

record *tstate* = *time* :: *nat*

type-synonym $'a$ *time* = $(\!|\mathit{time} :: \mathit{nat}, \ldots :: 'a|\!)$

primrec *timeit* :: $'a$ *time com* ⇒ $'a$ *time com*

where
timeit (*Basic f*) = (*Basic f*; *Basic*(λ*s*. *s*(|*time* := *Suc* (*time s*)|)))
| *timeit* (*c1*; *c2*) = (*timeit c1*; *timeit c2*)
| *timeit* (*Cond b c1 c2*) = *Cond b* (*timeit c1*) (*timeit c2*)
| *timeit* (*While b iv v c*) = *While b iv v* (*timeit c*)

record *tvars* = *tstate* +
I :: *nat*
J :: *nat*

lemma *lem*: (*0*::*nat*) < *n* ⟹ *n* + *n* ≤ *Suc* (*n* ∗ *n*)
⟨*proof*⟩

lemma
⊢ {|*i* = ´*I* ∧ ´*time* = *0*|}
(*timeit*
(*WHILE* ´*I* ≠ *0*
INV {|*2* ∗´ *time* + ´*I* ∗ ´*I* + *5* ∗ ´*I* = *i* ∗ *i* + *5* ∗ *i*|}
DO
´*J* := ´*I*;
WHILE ´*J* ≠ *0*
INV {|*0* < ´*I* ∧ *2* ∗ ´*time* + ´*I* ∗ ´*I* + *3* ∗ ´*I* + *2* ∗ ´*J* − *2* = *i* ∗ *i* + *5*
∗ *i*|}
DO ´*J* := ´*J* − *1* *OD*;
´*I* := ´*I* − *1*
OD))
{|*2* ∗ ´*time* = *i* ∗ *i* + *5* ∗ *i*|}
⟨*proof*⟩

end

10 The Mutilated Checker Board Problem

theory *Mutilated-Checkerboard*
imports *Main*
begin

The Mutilated Checker Board Problem, formalized inductively. See [5] for the original tactic script version.

10.1 Tilings

inductive-set *tiling* :: 'a *set set* ⇒ 'a *set set* **for** *A* :: 'a *set set*
where
empty: {} ∈ *tiling A*
| *Un*: *a* ∪ *t* ∈ *tiling A* **if** *a* ∈ *A* **and** *t* ∈ *tiling A* **and** *a* ⊆ − *t*

The union of two disjoint tilings is a tiling.

lemma *tiling-Un*:

assumes $t \in \mathit{tiling}\ A$
and $u \in \mathit{tiling}\ A$
and $t \cap u = \{\}$
shows $t \cup u \in \mathit{tiling}\ A$
⟨*proof*⟩

10.2 Basic properties of "below"

definition $\mathit{below} :: \mathit{nat} \Rightarrow \mathit{nat}\ \mathit{set}$
where $\mathit{below}\ n = \{i.\ i < n\}$

lemma *below-less-iff* [*iff*]: $i \in \mathit{below}\ k \longleftrightarrow i < k$
⟨*proof*⟩

lemma *below-0*: $\mathit{below}\ 0 = \{\}$
⟨*proof*⟩

lemma *Sigma-Suc1*: $m = n + 1 \Longrightarrow \mathit{below}\ m \times B = (\{n\} \times B) \cup (\mathit{below}\ n \times B)$
⟨*proof*⟩

lemma *Sigma-Suc2*:
$m = n + 2 \Longrightarrow$
$A \times \mathit{below}\ m = (A \times \{n\}) \cup (A \times \{n + 1\}) \cup (A \times \mathit{below}\ n)$
⟨*proof*⟩

lemmas *Sigma-Suc* = *Sigma-Suc1 Sigma-Suc2*

10.3 Basic properties of "evnodd"

definition $\mathit{evnodd} :: (\mathit{nat} \times \mathit{nat})\ \mathit{set} \Rightarrow \mathit{nat} \Rightarrow (\mathit{nat} \times \mathit{nat})\ \mathit{set}$
where $\mathit{evnodd}\ A\ b = A \cap \{(i, j).\ (i + j)\ \mathit{mod}\ 2 = b\}$

lemma *evnodd-iff*: $(i, j) \in \mathit{evnodd}\ A\ b \longleftrightarrow (i, j) \in A\ \wedge (i + j)\ \mathit{mod}\ 2 = b$
⟨*proof*⟩

lemma *evnodd-subset*: $\mathit{evnodd}\ A\ b \subseteq A$
⟨*proof*⟩

lemma *evnoddD*: $x \in \mathit{evnodd}\ A\ b \Longrightarrow x \in A$
⟨*proof*⟩

lemma *evnodd-finite*: $\mathit{finite}\ A \Longrightarrow \mathit{finite}\ (\mathit{evnodd}\ A\ b)$
⟨*proof*⟩

lemma *evnodd-Un*: $\mathit{evnodd}\ (A \cup B)\ b = \mathit{evnodd}\ A\ b \cup \mathit{evnodd}\ B\ b$
⟨*proof*⟩

lemma *evnodd-Diff*: $\mathit{evnodd}\ (A - B)\ b = \mathit{evnodd}\ A\ b - \mathit{evnodd}\ B\ b$
⟨*proof*⟩

lemma *evnodd-empty*: *evnodd* {} *b* = {}
⟨*proof*⟩

lemma *evnodd-insert*: *evnodd* (*insert* (*i*, *j*) *C*) *b* =
(*if* (*i* + *j*) *mod* 2 = *b*
then insert (*i*, *j*) (*evnodd C b*) *else evnodd C b*)
⟨*proof*⟩

10.4 Dominoes

inductive-set *domino* :: (*nat* × *nat*) *set set*
where
horiz: {(*i*, *j*), (*i*, *j* + 1)} ∈ *domino*
| *vertl*: {(*i*, *j*), (*i* + 1, *j*)} ∈ *domino*

lemma *dominoes-tile-row*:
{*i*} × *below* (2 ∗ *n*) ∈ *tiling domino*
(**is** *?B n* ∈ *?T*)
⟨*proof*⟩

lemma *dominoes-tile-matrix*:
below m × *below* (2 ∗ *n*) ∈ *tiling domino*
(**is** *?B m* ∈ *?T*)
⟨*proof*⟩

lemma *domino-singleton*:
assumes *d* ∈ *domino*
and *b* < 2
shows ∃ *i j*. *evnodd d b* = {(*i*, *j*)} (**is** *?P d*)
⟨*proof*⟩

lemma *domino-finite*:
assumes *d* ∈ *domino*
shows *finite d*
⟨*proof*⟩

10.5 Tilings of dominoes

lemma *tiling-domino-finite*:
assumes *t*: *t* ∈ *tiling domino* (**is** *t* ∈ *?T*)
shows *finite t* (**is** *?F t*)
⟨*proof*⟩

lemma *tiling-domino-01*:
assumes *t*: *t* ∈ *tiling domino* (**is** *t* ∈ *?T*)
shows *card* (*evnodd t* 0) = *card* (*evnodd t* 1)
⟨*proof*⟩

10.6 Main theorem

definition *mutilated-board* :: *nat* ⇒ *nat* ⇒ (*nat* × *nat*) *set*
  where *mutilated-board m n* =
    below (*2* ∗ (*m* + *1*)) × *below* (*2* ∗ (*n* + *1*)) − {(*0*, *0*)} − {(*2* ∗ *m* + *1*, *2* ∗ *n* + *1*)}

theorem *mutil-not-tiling*: *mutilated-board m n* ∉ *tiling domino*
⟨*proof*⟩

end

11 An old chestnut

theory *Puzzle*
  imports *Main*
begin[8]

Problem. Given some function $f\colon \mathbb{N} \to \mathbb{N}$ such that $f\ (f\ n) < f\ (Suc\ n)$ for all n. Demonstrate that f is the identity.

theorem
  assumes *f-ax*: ⋀*n*. *f* (*f n*) < *f* (*Suc n*)
  shows *f n* = *n*
⟨*proof*⟩

end

12 Summing natural numbers

theory *Summation*
  imports *Main*
begin

Subsequently, we prove some summation laws of natural numbers (including odds, squares, and cubes). These examples demonstrate how plain natural deduction (including induction) may be combined with calculational proof.

12.1 Summation laws

The sum of natural numbers $0 + \cdots + n$ equals $n \times (n + 1)/2$. Avoiding formal reasoning about division we prove this equation multiplied by 2.

theorem *sum-of-naturals*:
  2 ∗ (∑ *i*::*nat*=*0*..*n*. *i*) = *n* ∗ (*n* + *1*)
  (**is** *?P n* **is** *?S n* = -)

[8] A question from "Bundeswettbewerb Mathematik". Original pen-and-paper proof due to Herbert Ehler; Isabelle tactic script by Tobias Nipkow.

⟨*proof*⟩

The above proof is a typical instance of mathematical induction. The main statement is viewed as some *?P n* that is split by the induction method into base case *?P 0*, and step case *?P n* $\Longrightarrow$ *?P* (*Suc n*) for arbitrary *n*.

The step case is established by a short calculation in forward manner. Starting from the left-hand side *?S* (*n + 1*) of the thesis, the final result is achieved by transformations involving basic arithmetic reasoning (using the Simplifier). The main point is where the induction hypothesis *?S n = n* $\times$ (*n + 1*) is introduced in order to replace a certain subterm. So the "transitivity" rule involved here is actual *substitution*. Also note how the occurrence of "..." in the subsequent step documents the position where the right-hand side of the hypothesis got filled in.

A further notable point here is integration of calculations with plain natural deduction. This works so well in Isar for two reasons.

1. Facts involved in **also** / **finally** calculational chains may be just anything. There is nothing special about **have**, so the natural deduction element **assume** works just as well.

2. There are two *separate* primitives for building natural deduction contexts: **fix** *x* and **assume** *A*. Thus it is possible to start reasoning with some new "arbitrary, but fixed" elements before bringing in the actual assumption. In contrast, natural deduction is occasionally formalized with basic context elements of the form *x*:*A* instead.

We derive further summation laws for odds, squares, and cubes as follows. The basic technique of induction plus calculation is the same as before.

theorem *sum-of-odds*:
(∑ *i*::*nat*=*0*..<*n*. *2 * i + 1*) = *n*^*Suc* (*Suc 0*)
(**is** *?P n* **is** *?S n* = -)
⟨*proof*⟩

Subsequently we require some additional tweaking of Isabelle built-in arithmetic simplifications, such as bringing in distributivity by hand.

lemmas *distrib* = *add-mult-distrib add-mult-distrib2*

theorem *sum-of-squares*:
*6 * (*∑ *i*::*nat*=*0*..*n*. *i*^*Suc* (*Suc 0*)) = n * (n + 1) * (2 * n + 1)*
(**is** *?P n* **is** *?S n* = -)
⟨*proof*⟩

theorem *sum-of-cubes*:
*4 * (*∑ *i*::*nat*=*0*..*n*. *i*^*3*) = (n * (n + 1))^*Suc* (*Suc 0*)*
(**is** *?P n* **is** *?S n* = -)

⟨*proof*⟩

Note that in contrast to older traditions of tactical proof scripts, the structured proof applies induction on the original, unsimplified statement. This allows to state the induction cases robustly and conveniently. Simplification (or other automated) methods are then applied in terminal position to solve certain sub-problems completely.

As a general rule of good proof style, automatic methods such as *simp* or *auto* should normally be never used as initial proof methods with a nested sub-proof to address the automatically produced situation, but only as terminal ones to solve sub-problems.

end

References

[1] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[2] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in Higher-Order Logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *LNCS*, 1998.

[3] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[4] L. C. Paulson. *Introduction to Isabelle*.

[5] L. C. Paulson. A simple formalization and proof for the mutilated chess board. Technical Report 394, Comp. Lab., Univ. Camb., 1996. http://www.cl.cam.ac.uk/users/lcp/papers/Reports/mutil.pdf.

[6] M. Wenzel. *The Isabelle/Isar Reference Manual*.

[7] 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.

[8] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.