Miscellaneous Isabelle/Isar examples

Makarius Wenzel

With contributions by Gertrud Bauer and Tobias Nipkow

March 13, 2025

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
  fix A B :: bool
  fix P :: 'a ⇒ bool

  have A ⟶ B
  proof
    show B if A using that ⟨proof⟩
  qed

  have ¬ A
  proof
    show False if A using that ⟨proof⟩
  qed

  have ∀ x. P x
  proof
    show P x for x ⟨proof⟩
  qed
end
```

1.2 If-and-only-if

```
notepad
begin
  fix A B :: bool

  have A ⟷ B
  proof
    show B if A ⟨proof⟩
    show A if B ⟨proof⟩
  qed
next
  fix A B :: bool

  have iff-comm: (A ∧ B) ⟷ (B ∧ A)
  proof
    show B ∧ A if A ∧ B
    proof
      show B using that ..
      show A using that ..
    qed
    show A ∧ B if B ∧ A
    proof
      show A using that ..
```

```
    show B using that ..
  qed
qed
```

Alternative proof, avoiding redundant copy of symmetric argument.

```
have iff-comm: (A ∧ B) ⟷ (B ∧ A)
proof
  show B ∧ A if A ∧ B for A B
  proof
    show B using that ..
    show A using that ..
  qed
  then show A ∧ B if B ∧ A
    by this (rule that)
qed
end
```

1.3 Elimination and cases

```
notepad
begin
  fix A B C D :: bool
  assume *: A ∨ B ∨ C ∨ D

  consider (a) A | (b) B | (c) C | (d) D
    using * by blast
  then have something
  proof cases
    case a thm ‹A›
    then show ?thesis ⟨proof⟩
  next
    case b thm ‹B›
    then show ?thesis ⟨proof⟩
  next
    case c thm ‹C›
    then show ?thesis ⟨proof⟩
  next
    case d thm ‹D›
    then show ?thesis ⟨proof⟩
  qed
next
  fix A :: 'a ⇒ bool
  fix B :: 'b ⇒ 'c ⇒ bool
  assume *: (∃x. A x) ∨ (∃y z. B y z)

  consider (a) x where A x | (b) y z where B y z
    using * by blast
  then have something
  proof cases
```

```
    case a  thm ‹A x›
    then show ?thesis ⟨proof⟩
  next
    case b  thm ‹B y z›
    then show ?thesis ⟨proof⟩
  qed
end
```

1.4 Induction

```
notepad
begin
  fix P :: nat ⇒ bool
  fix n :: nat

  have P n
  proof (induct n)
    show P 0 ⟨proof⟩
    show P (Suc n) if P n for n  thm ‹P n›
      using that ⟨proof⟩
  qed
end
```

1.5 Suffices-to-show

```
notepad
begin
  fix A B C
  assume r: A ⟹ B ⟹ C

  have C
  proof −
    show ?thesis when A (is ?A) and B (is ?B)
      using that by (rule r)
    show ?A ⟨proof⟩
    show ?B ⟨proof⟩
  qed
next
  fix a :: 'a
  fix A :: 'a ⇒ bool
  fix C

  have C
  proof −
    show ?thesis when A x (is ?A) for x :: 'a  — abstract x
      using that ⟨proof⟩
    show ?A a  — concrete a
      ⟨proof⟩
  qed
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 ⟶ A
proof
  assume A
  show A by fact
qed

lemma K: A ⟶ B ⟶ A
proof
  assume A
  show B ⟶ A
  proof
    show A by fact
  qed
qed

lemma S: (A ⟶ B ⟶ C) ⟶ (A ⟶ B) ⟶ A ⟶ C
proof
  assume A ⟶ B ⟶ C
  show (A ⟶ B) ⟶ A ⟶ C
  proof
    assume A ⟶ B
    show A ⟶ C
    proof
      assume A
      show C
      proof (rule mp)
        show B ⟶ C by (rule mp) fact+
        show B by (rule mp) fact+
      qed
    qed
  qed
qed
```

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
qed

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 $A \longrightarrow A$
  by *rule*

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

lemma $A \longrightarrow A$ **..**

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 $A \longrightarrow B \longrightarrow A$
proof (*intro impI*)
  assume A
  show A **by** *fact*
qed

Again, the body may be collapsed.

lemma $A \longrightarrow B \longrightarrow A$
  by (*intro impI*)

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.

---

[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.

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
  assume $A \wedge B$
  show $B \wedge A$
  proof
    show B **by** (*rule conjunct2*) *fact*
    show A **by** (*rule conjunct1*) *fact*
  qed
qed

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
  assume $A \wedge B$
  show $B \wedge A$
  proof
    from ‹$A \wedge B$› **show** B **..**
    from ‹$A \wedge B$› **show** A **..**
  qed
qed

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
  assume $A \land B$
  then show $B \land A$
  proof — rule *conjE* of $A \land B$
    assume B A
    then show *?thesis* **..** — rule *conjI* of $B \land A$
  qed
qed

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 $A \land B \longrightarrow B \land A$
proof
  assume $A \land B$
  from ‹$A \land B$› **have** A **..**
  from ‹$A \land B$› **have** B **..**
  from ‹B› ‹A› **show** $B \land A$ **..**
qed

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 $A \land B \longrightarrow B \land A$
proof $-$
  {
    assume $A \land B$
    from ‹$A \land B$› **have** A **..**
    from ‹$A \land B$› **have** B **..**
    from ‹B› ‹A› **have** $B \land A$ **..**
  }
  then show *?thesis* **..** — rule *impI*
qed

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 $A \land B \longrightarrow B \land A$
proof
  assume $A \land B$
  then show $B \land A$
  proof
    assume B A
    then show *?thesis* **..**
  qed
qed

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

lemma $P \lor P \longrightarrow P$
proof
  assume $P \lor P$
  then show P
  proof — rule *disjE*: $\dfrac{A \lor B \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}$
    assume P **show** P **by** *fact*
  next
    assume P **show** P **by** *fact*
  qed
qed

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 $P \lor P \longrightarrow P$
proof
  assume $P \lor P$
  then show P
  proof
    assume P
    show P **by** *fact*
    show P **by** *fact*
  qed
qed

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 $P \lor P \longrightarrow P$
proof
  assume $P \lor P$
  then show P **..**
qed

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
  assume $\exists x.\ P\ (f\ x)$
  then show $\exists y.\ P\ y$
  proof (*rule exE*) — rule *exE*: $\dfrac{\exists x.\ A(x) \qquad \begin{matrix}[A(x)]_x \\ \vdots \\ B\end{matrix}}{B}$
    fix a
    assume $P\ (f\ a)$ (**is** P *?witness*)
    then show *?thesis* **by** (*rule exI* [*of* P *?witness*])
  qed
qed

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
  assume $\exists\, x.\ P\ (f\ x)$
  then show $\exists\, y.\ P\ y$
  proof
    fix a
    assume $P\ (f\ a)$
    then show *?thesis* **..**
  qed
qed

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
  assume $\exists\, x.\ P\ (f\ x)$
  then obtain a **where** $P\ (f\ a)$ **..**
  then show $\exists\, y.\ P\ y$ **..**
qed

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 $-$
  assume $A \wedge B$
  assume r: $A \Longrightarrow B \Longrightarrow C$
  show C
  proof (*rule r*)
    show A **by** (*rule conjunct1*) *fact*
    show B **by** (*rule conjunct2*) *fact*
  qed
qed

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 $'val\ binop = 'val \Rightarrow 'val \Rightarrow 'val$

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 $eval :: ('adr,\ 'val)\ expr \Rightarrow ('adr \Rightarrow 'val) \Rightarrow 'val$
  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 $exec :: (('adr,\ 'val)\ instr)\ list \Rightarrow 'val\ list \Rightarrow ('adr \Rightarrow 'val) \Rightarrow 'val\ list$
  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 (induct xs arbitrary: stack)
  case Nil
  show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (induct x)
    case Const
    from Cons show ?case by simp
  next
    case Load
    from Cons show ?case by simp
  next
    case Apply
    from Cons show ?case by simp
  qed
qed
```

```
theorem correctness: execute (compile e) env = eval e env
proof −
  have ⋀stack. exec (compile e) stack env = eval e env # stack
  proof (induct e)
    case Variable
    show ?case by simp
  next
```

```
    case Constant
    show ?case by simp
  next
    case Binop
    then show ?case by (simp add: exec-append)
  qed
  then show ?thesis by (simp add: execute-def)
qed
```

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 (induct xs arbitrary: stack)
  case (Nil s)
  have exec ([] @ ys) s env = exec ys s env
    by simp
  also have ... = exec ys (exec [] s env) env
    by simp
  finally show ?case .
next
  case (Cons x xs s)
  show ?case
  proof (induct x)
    case (Const val)
    have exec ((Const val # xs) @ ys) s env = exec (Const val # xs @ ys) s env
      by simp
    also have ... = exec (xs @ ys) (val # s) env
      by simp
    also from Cons have ... = exec ys (exec xs (val # s) env) env .
    also have ... = exec ys (exec (Const val # xs) s env) env
      by simp
    finally show ?case .
  next
    case (Load adr)
    from Cons show ?case
      by simp — same as above
  next
    case (Apply fn)
    have exec ((Apply fn # xs) @ ys) s env =
        exec (Apply fn # xs @ ys) s env by simp
    also have ... =
        exec (xs @ ys) (fn (hd s) (hd (tl s)) # (tl (tl s))) env
      by simp
    also from Cons have ... =
        exec ys (exec xs (fn (hd s) (hd (tl s)) # tl (tl s)) env) env .
```

```
    also have ... = exec ys (exec (Apply fn # xs) s env) env
      by simp
    finally show ?case .
  qed
qed

theorem correctness': execute (compile e) env = eval e env
proof -
  have exec-compile: ⋀stack. exec (compile e) stack env = eval e env # stack
  proof (induct e)
    case (Variable adr s)
    have exec (compile (Variable adr)) s env = exec [Load adr] s env
      by simp
    also have ... = env adr # s
      by simp
    also have env adr = eval (Variable adr) env
      by simp
    finally show ?case .
  next
    case (Constant val s)
    show ?case by simp — same as above
  next
    case (Binop fn e1 e2 s)
    have exec (compile (Binop fn e1 e2)) s env =
        exec (compile e2 @ compile e1 @ [Apply fn]) s env
      by simp
    also have ... = exec [Apply fn]
        (exec (compile e1) (exec (compile e2) s env) env) env
      by (simp only: exec-append)
    also have exec (compile e2) s env = eval e2 env # s
      by fact
    also have exec (compile e1) ... env = eval e1 env # ...
      by fact
    also have exec [Apply fn] ... env =
        fn (hd ...) (hd (tl ...)) # (tl (tl ...))
      by simp
    also have ... = fn (eval e1 env) (eval e2 env) # s
      by simp
    also have fn (eval e1 env) (eval e2 env) =
        eval (Binop fn e1 e2) env
      by simp
    finally show ?case .
  qed

  have execute (compile e) env = hd (exec (compile e) [] env)
    by (simp add: execute-def)
  also from exec-compile have exec (compile e) [] env = [eval e env] .
  also have hd ... = eval e env
    by simp
```

finally show *?thesis* **.**
qed

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*
by (*induct n rule*: *fib.induct*) *simp-all*

Alternative induction rule.

theorem *fib-induct*: $P\ 0 \Longrightarrow P\ 1 \Longrightarrow (\bigwedge n.\ P\ (n + 1) \Longrightarrow P\ n \Longrightarrow P\ (n + 2)) \Longrightarrow P\ n$
for *n* :: *nat*
by (*induct rule*: *fib.induct*) *simp-all*

4.2 Fib and gcd commute

A few 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 (*induct n rule*: *fib-induct*)
show *?P 0* **by** *simp*
show *?P 1* **by** *simp*
fix *n*
have *fib* (*n* + *2* + *k* + *1*)
= *fib* (*n* + *k* + *1*) + *fib* (*n* + *1* + *k* + *1*) **by** *simp*
also assume *fib* (*n* + *k* + *1*) = *fib* (*k* + *1*) ∗ *fib* (*n* + *1*) + *fib k* ∗ *fib n* (**is** - = *?R1*)
also assume *fib* (*n* + *1* + *k* + *1*) = *fib* (*k* + *1*) ∗ *fib* (*n* + *1* + *1*) + *fib k* ∗ *fib* (*n* + *1*)
(**is** - = *?R2*)

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

also have *?R1* + *?R2* = *fib* (*k* + *1*) ∗ *fib* (*n* + *2* + *1*) + *fib* *k* ∗ *fib* (*n* + *2*)
  by (*simp add*: *add-mult-distrib2*)
  finally show *?P* (*n* + *2*) **.**
qed

lemma *coprime-fib-Suc*: *coprime* (*fib* *n*) (*fib* (*n* + *1*))
  (**is** *?P n*)
proof (*induct n rule*: *fib-induct*)
  show *?P 0* **by** *simp*
  show *?P 1* **by** *simp*
  fix *n*
  assume *P*: *coprime* (*fib* (*n* + *1*)) (*fib* (*n* + *1* + *1*))
  have *fib* (*n* + *2* + *1*) = *fib* (*n* + *1*) + *fib* (*n* + *2*)
    by *simp*
  also have ... = *fib* (*n* + *2*) + *fib* (*n* + *1*)
    by *simp*
  also have *gcd* (*fib* (*n* + *2*)) ... = *gcd* (*fib* (*n* + *2*)) (*fib* (*n* + *1*))
    by (*rule gcd-add2*)
  also have ... = *gcd* (*fib* (*n* + *1*)) (*fib* (*n* + *1* + *1*))
    by (*simp add*: *gcd.commute*)
  also have ... = *1*
    using *P* **by** *simp*
  finally show *?P* (*n* + *2*)
    by (*simp add*: *coprime-iff-gcd-eq-1*)
qed

lemma *gcd-mult-add*: (*0*::*nat*) < *n* ⟹ *gcd* (*n* ∗ *k* + *m*) *n* = *gcd* *m* *n*
proof −
  assume *0* < *n*
  then have *gcd* (*n* ∗ *k* + *m*) *n* = *gcd* *n* (*m* *mod* *n*)
    by (*simp add*: *gcd-non-0-nat add.commute*)
  also from ‹*0* < *n*› **have** ... = *gcd* *m* *n*
    by (*simp add*: *gcd-non-0-nat*)
  finally show *?thesis* **.**
qed

lemma *gcd-fib-add*: *gcd* (*fib* *m*) (*fib* (*n* + *m*)) = *gcd* (*fib* *m*) (*fib* *n*)
proof (*cases m*)
  case *0*
  then show *?thesis* **by** *simp*
next
  case (*Suc k*)
  then have *gcd* (*fib* *m*) (*fib* (*n* + *m*)) = *gcd* (*fib* (*n* + *k* + *1*)) (*fib* (*k* + *1*))
    by (*simp add*: *gcd.commute*)
  also have *fib* (*n* + *k* + *1*) = *fib* (*k* + *1*) ∗ *fib* (*n* + *1*) + *fib* *k* ∗ *fib* *n*
    by (*rule fib-add*)
  also have *gcd* ... (*fib* (*k* + *1*)) = *gcd* (*fib* *k* ∗ *fib* *n*) (*fib* (*k* + *1*))
    by (*simp add*: *gcd-mult-add*)
  also have ... = *gcd* (*fib* *n*) (*fib* (*k* + *1*))

```
    using coprime-fib-Suc [of k] gcd-mult-left-right-cancel [of fib (k + 1) fib k fib n]
     by (simp add: ac-simps)
  also have ... = gcd (fib m) (fib n)
     using Suc by (simp add: gcd.commute)
  finally show ?thesis .
qed

lemma gcd-fib-diff: gcd (fib m) (fib (n − m)) = gcd (fib m) (fib n) if m ≤ n
proof −
  have gcd (fib m) (fib (n − m)) = gcd (fib m) (fib (n − m + m))
     by (simp add: gcd-fib-add)
  also from ‹m ≤ n› have n − m + m = n
     by simp
  finally show ?thesis .
qed

lemma gcd-fib-mod: gcd (fib m) (fib (n mod m)) = gcd (fib m) (fib n) if 0 < m
proof (induct n rule: nat-less-induct)
  case hyp: (1 n)
  show ?case
  proof −
    have n mod m = (if n < m then n else (n − m) mod m)
      by (rule mod-if)
    also have gcd (fib m) (fib ...) = gcd (fib m) (fib n)
    proof (cases n < m)
      case True
      then show ?thesis by simp
    next
      case False
      then have m ≤ n by simp
      from ‹0 < m› and False have n − m < n
        by simp
      with hyp have gcd (fib m) (fib ((n − m) mod m))
          = gcd (fib m) (fib (n − m)) by simp
      also have ... = gcd (fib m) (fib n)
        using ‹m ≤ n› by (rule gcd-fib-diff)
      finally have gcd (fib m) (fib ((n − m) mod m)) =
          gcd (fib m) (fib n) .
      with False show ?thesis by simp
    qed
    finally show ?thesis .
  qed
qed

theorem fib-gcd: fib (gcd m n) = gcd (fib m) (fib n)
  (is ?P m n)
proof (induct m n rule: gcd-nat-induct)
  fix m n :: nat
  show fib (gcd m 0) = gcd (fib m) (fib 0)
```

by *simp*
assume *n*: $0 < n$
then have $gcd\ m\ n = gcd\ n\ (m\ mod\ n)$
by (*simp add*: *gcd-non-0-nat*)
also assume *hyp*: $fib\ \ldots = gcd\ (fib\ n)\ (fib\ (m\ mod\ n))$
also from *n* **have** $\ldots = gcd\ (fib\ n)\ (fib\ m)$
by (*rule gcd-fib-mod*)
also have $\ldots = gcd\ (fib\ m)\ (fib\ n)$
by (*rule gcd.commute*)
finally show $fib\ (gcd\ m\ n) = gcd\ (fib\ m)\ (fib\ n)$ **.**
qed

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 −
have $x * inverse\ x = 1 * (x * inverse\ x)$
by (*simp only*: *group-left-one*)
also have $\ldots = 1 * x * inverse\ x$
by (*simp only*: *group-assoc*)
also have $\ldots = inverse\ (inverse\ x) * inverse\ x * x * inverse\ x$
by (*simp only*: *group-left-inverse*)
also have $\ldots = inverse\ (inverse\ x) * (inverse\ x * x) * inverse\ x$
by (*simp only*: *group-assoc*)
also have $\ldots = inverse\ (inverse\ x) * 1 * inverse\ x$
by (*simp only*: *group-left-inverse*)
also have $\ldots = inverse\ (inverse\ x) * (1 * inverse\ x)$
by (*simp only*: *group-assoc*)
also have $\ldots = inverse\ (inverse\ x) * inverse\ x$
by (*simp only*: *group-left-one*)

also have ... = *1*
by (*simp only*: *group-left-inverse*)
finally show *?thesis* **.**
qed

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

theorem (**in** *group*) *group-right-one*: $x * 1 = x$
proof −
have $x * 1 = x * (inverse\ x * x)$
by (*simp only*: *group-left-inverse*)
also have ... = $x * inverse\ x * x$
by (*simp only*: *group-assoc*)
also have ... = $1 * x$
by (*simp only*: *group-right-inverse*)
also have ... = x
by (*simp only*: *group-left-one*)
finally show *?thesis* **.**
qed

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 −
have $x * 1 = x * (inverse\ x * x)$
by (*simp only*: *group-left-inverse*)

note *calculation* = *this*
— first calculational step: init calculation register

have ... = $x * inverse\ x * x$

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

by (*simp only*: *group-assoc*)

note *calculation* = *trans* [*OF calculation this*]
— general calculational step: compose with transitivity rule

have ... = *1* ∗ *x*
by (*simp only*: *group-right-inverse*)

note *calculation* = *trans* [*OF calculation this*]
— general calculational step: compose with transitivity rule

have ... = *x*
by (*simp only*: *group-left-one*)

note *calculation* = *trans* [*OF calculation this*]
— final calculational step: compose with transitivity rule ...
from *calculation*
— ... and pick up the final result

show *?thesis* **.**
qed

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 $?x * 1 = ?x$), we may still instantiate *group* ⊆ *monoid* properly as follows.

instance *group* ⊆ *monoid*
by *intro-classes*
(*rule group-assoc*,
rule group-left-one,
rule group-right-one)

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 −
  have $1 = x * \mathit{inverse}\ x$
    by (*simp only*: *group-right-inverse*)
  also have $\ldots = (e * x) * \mathit{inverse}\ x$
    by (*simp only*: *eq*)
  also have $\ldots = e * (x * \mathit{inverse}\ x)$
    by (*simp only*: *group-assoc*)
  also have $\ldots = e * 1$
    by (*simp only*: *group-right-inverse*)
  also have $\ldots = e$
    by (*simp only*: *group-right-one*)
  finally show *?thesis* **.**
qed

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

theorem (**in** *group*) *group-inverse-equality*:
  assumes *eq*: $x' * x = 1$
  shows $\mathit{inverse}\ x = x'$
proof −
  have $\mathit{inverse}\ x = 1 * \mathit{inverse}\ x$
    by (*simp only*: *group-left-one*)
  also have $\ldots = (x' * x) * \mathit{inverse}\ x$
    by (*simp only*: *eq*)
  also have $\ldots = x' * (x * \mathit{inverse}\ x)$
    by (*simp only*: *group-assoc*)
  also have $\ldots = x' * 1$
    by (*simp only*: *group-right-inverse*)
  also have $\ldots = x'$
    by (*simp only*: *group-right-one*)
  finally show *?thesis* **.**
qed

The inverse operation has some further characteristic properties.

theorem (**in** *group*) *group-inverse-times*: $\mathit{inverse}\ (x * y) = \mathit{inverse}\ y * \mathit{inverse}\ x$

```
proof (rule group-inverse-equality)
  show (inverse y * inverse x) * (x * y) = 1
  proof -
    have (inverse y * inverse x) * (x * y) =
        (inverse y * (inverse x * x)) * y
      by (simp only: group-assoc)
    also have ... = (inverse y * 1) * y
      by (simp only: group-left-inverse)
    also have ... = inverse y * y
      by (simp only: group-right-one)
    also have ... = 1
      by (simp only: group-left-inverse)
    finally show ?thesis .
  qed
qed

theorem (in group) inverse-inverse: inverse (inverse x) = x
proof (rule group-inverse-equality)
  show x * inverse x = one
    by (simp only: group-right-inverse)
qed

theorem (in group) inverse-inject:
  assumes eq: inverse x = inverse y
  shows x = y
proof -
  have x = x * 1
    by (simp only: group-right-one)
  also have ... = x * (inverse y * y)
    by (simp only: group-left-inverse)
  also have ... = x * (inverse x * y)
    by (simp only: eq)
  also have ... = (x * inverse x) * y
    by (simp only: group-assoc)
  also have ... = 1 * y
    by (simp only: group-right-inverse)
  also have ... = y
    by (simp only: group-left-one)
  finally show ?thesis .
qed

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 −
  have $x \odot inverse\ x = one \odot (x \odot inverse\ x)$
    by (*simp only*: *left-one*)
  also have $\ldots = one \odot x \odot inverse\ x$
    by (*simp only*: *assoc*)
  also have $\ldots = inverse\ (inverse\ x) \odot inverse\ x \odot x \odot inverse\ x$
    by (*simp only*: *left-inverse*)
  also have $\ldots = inverse\ (inverse\ x) \odot (inverse\ x \odot x) \odot inverse\ x$
    by (*simp only*: *assoc*)
  also have $\ldots = inverse\ (inverse\ x) \odot one \odot inverse\ x$
    by (*simp only*: *left-inverse*)
  also have $\ldots = inverse\ (inverse\ x) \odot (one \odot inverse\ x)$
    by (*simp only*: *assoc*)
  also have $\ldots = inverse\ (inverse\ x) \odot inverse\ x$
    by (*simp only*: *left-one*)
  also have $\ldots = one$
    by (*simp only*: *left-inverse*)
  finally show *?thesis* **.**
qed

lemma *right-one*: $x \odot one = x$
proof −
  have $x \odot one = x \odot (inverse\ x \odot x)$
    by (*simp only*: *left-inverse*)
  also have $\ldots = x \odot inverse\ x \odot x$
    by (*simp only*: *assoc*)
  also have $\ldots = one \odot x$
    by (*simp only*: *right-inverse*)
  also have $\ldots = x$
    by (*simp only*: *left-one*)
  finally show *?thesis* **.**
qed

lemma *one-equality*:
  assumes *eq*: $e \odot x = x$

shows *one* = *e*
proof −
have *one* = $x \odot$ *inverse x*
by (*simp only*: *right-inverse*)
also have ... = $(e \odot x) \odot$ *inverse x*
by (*simp only*: *eq*)
also have ... = $e \odot (x \odot$ *inverse x*)
by (*simp only*: *assoc*)
also have ... = $e \odot$ *one*
by (*simp only*: *right-inverse*)
also have ... = *e*
by (*simp only*: *right-one*)
finally show *?thesis* **.**
qed

lemma *inverse-equality*:
assumes *eq*: $x' \odot x =$ *one*
shows *inverse* $x = x'$
proof −
have *inverse x* = *one* $\odot$ *inverse x*
by (*simp only*: *left-one*)
also have ... = $(x' \odot x) \odot$ *inverse x*
by (*simp only*: *eq*)
also have ... = $x' \odot (x \odot$ *inverse x*)
by (*simp only*: *assoc*)
also have ... = $x' \odot$ *one*
by (*simp only*: *right-inverse*)
also have ... = x'
by (*simp only*: *right-one*)
finally show *?thesis* **.**
qed

end

end

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

theory *Group-Notepad*
imports *Main*
begin

notepad
begin

hypothetical group axiomatization

fix *prod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹$\odot$› *70*)

```
    and one :: 'a
    and inverse :: 'a ⇒ 'a
  assume assoc: (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)
    and left-one: one ⊙ x = x
    and left-inverse: inverse x ⊙ x = one
    for x y z
```

some consequences

```
  have right-inverse: x ⊙ inverse x = one for x
  proof −
    have x ⊙ inverse x = one ⊙ (x ⊙ inverse x)
      by (simp only: left-one)
    also have ... = one ⊙ x ⊙ inverse x
      by (simp only: assoc)
    also have ... = inverse (inverse x) ⊙ inverse x ⊙ x ⊙ inverse x
      by (simp only: left-inverse)
    also have ... = inverse (inverse x) ⊙ (inverse x ⊙ x) ⊙ inverse x
      by (simp only: assoc)
    also have ... = inverse (inverse x) ⊙ one ⊙ inverse x
      by (simp only: left-inverse)
    also have ... = inverse (inverse x) ⊙ (one ⊙ inverse x)
      by (simp only: assoc)
    also have ... = inverse (inverse x) ⊙ inverse x
      by (simp only: left-one)
    also have ... = one
      by (simp only: left-inverse)
    finally show ?thesis .
  qed

  have right-one: x ⊙ one = x for x
  proof −
    have x ⊙ one = x ⊙ (inverse x ⊙ x)
      by (simp only: left-inverse)
    also have ... = x ⊙ inverse x ⊙ x
      by (simp only: assoc)
    also have ... = one ⊙ x
      by (simp only: right-inverse)
    also have ... = x
      by (simp only: left-one)
    finally show ?thesis .
  qed

  have one-equality: one = e if eq: e ⊙ x = x for e x
  proof −
    have one = x ⊙ inverse x
      by (simp only: right-inverse)
    also have ... = (e ⊙ x) ⊙ inverse x
      by (simp only: eq)
    also have ... = e ⊙ (x ⊙ inverse x)
```

```
    by (simp only: assoc)
  also have ... = e ⊙ one
    by (simp only: right-inverse)
  also have ... = e
    by (simp only: right-one)
  finally show ?thesis .
qed

have inverse-equality: inverse x = x′ if eq: x′ ⊙ x = one for x x′
proof −
  have inverse x = one ⊙ inverse x
    by (simp only: left-one)
  also have ... = (x′ ⊙ x) ⊙ inverse x
    by (simp only: eq)
  also have ... = x′ ⊙ (x ⊙ inverse x)
    by (simp only: assoc)
  also have ... = x′ ⊙ one
    by (simp only: right-inverse)
  also have ... = x′
    by (simp only: right-one)
  finally show ?thesis .
qed

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 $SKIP \equiv Basic\ id$

type-synonym $'a\ sem = 'a \Rightarrow 'a \Rightarrow bool$

primrec $iter :: nat \Rightarrow 'a\ bexp \Rightarrow 'a\ sem \Rightarrow 'a\ sem$
  where
    $iter\ 0\ b\ S\ s\ s' \longleftrightarrow s \notin b \wedge s = s'$
  $|\ iter\ (Suc\ n)\ b\ S\ s\ s' \longleftrightarrow s \in b \wedge (\exists s''.\ S\ s\ s'' \wedge iter\ n\ b\ S\ s''\ s')$

primrec $Sem :: 'a\ com \Rightarrow 'a\ sem$
  where
    $Sem\ (Basic\ f)\ s\ s' \longleftrightarrow s' = f\ s$
  $|\ Sem\ (c1;\ c2)\ s\ s' \longleftrightarrow (\exists s''.\ Sem\ c1\ s\ s'' \wedge Sem\ c2\ s''\ s')$
  $|\ Sem\ (Cond\ b\ c1\ c2)\ s\ s' \longleftrightarrow (if\ s \in b\ then\ Sem\ c1\ s\ s'\ else\ Sem\ c2\ s\ s')$
  $|\ Sem\ (While\ b\ x\ y\ c)\ s\ s' \longleftrightarrow (\exists n.\ iter\ n\ b\ (Sem\ c)\ s\ s')$

definition $Valid :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ bexp \Rightarrow bool$ (‹(3⊢ -/ (2-)/ -)› [*100*, *55*, *100*] *50*)
  where $\vdash P\ c\ Q \longleftrightarrow (\forall s\ s'.\ Sem\ c\ s\ s' \longrightarrow s \in P \longrightarrow s' \in Q)$

lemma *ValidI* [*intro?*]: $(\bigwedge s\ s'.\ Sem\ c\ s\ s' \Longrightarrow s \in P \Longrightarrow s' \in Q) \Longrightarrow \vdash P\ c\ Q$
  by (*simp add*: *Valid-def*)

lemma *ValidD* [*dest?*]: $\vdash P\ c\ Q \Longrightarrow Sem\ c\ s\ s' \Longrightarrow s \in P \Longrightarrow s' \in Q$
  by (*simp add*: *Valid-def*)

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*: $\vdash \{s.\ f\ s \in P\}\ (Basic\ f)\ P$
proof
  fix $s\ s'$
  assume *s*: $s \in \{s.\ f\ s \in P\}$
  assume $Sem\ (Basic\ f)\ s\ s'$
  then have $s' = f\ s$ **by** *simp*
  with *s* **show** $s' \in P$ **by** *simp*
qed

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

assume *cmd1*: $\vdash P\ c1\ Q$ **and** *cmd2*: $\vdash Q\ c2\ R$
fix $s\ s'$
assume *s*: $s \in P$
assume $Sem\ (c1;\ c2)\ s\ s'$
then obtain s'' **where** *sem1*: $Sem\ c1\ s\ s''$ **and** *sem2*: $Sem\ c2\ s''\ s'$
by *auto*
from *cmd1 sem1 s* **have** $s'' \in Q$ **..**
with *cmd2 sem2* **show** $s' \in R$ **..**
qed

theorem *conseq*: $P' \subseteq P \Longrightarrow \vdash P\ c\ Q \Longrightarrow Q \subseteq Q' \Longrightarrow \vdash P'\ c\ Q'$
proof
assume *P'P*: $P' \subseteq P$ **and** *QQ'*: $Q \subseteq Q'$
assume *cmd*: $\vdash P\ c\ Q$
fix $s\ s' :: 'a$
assume *sem*: $Sem\ c\ s\ s'$
assume $s \in P'$ **with** *P'P* **have** $s \in P$ **..**
with *cmd sem* **have** $s' \in Q$ **..**
with *QQ'* **show** $s' \in Q'$ **..**
qed

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 $\vdash P\ (Cond\ b\ c1\ c2)\ Q$
proof
fix $s\ s'$
assume *s*: $s \in P$
assume *sem*: $Sem\ (Cond\ b\ c1\ c2)\ s\ s'$
show $s' \in Q$
proof *cases*
assume *b*: $s \in b$
from *case-b* **show** *?thesis*
proof
from *sem b* **show** $Sem\ c1\ s\ s'$ **by** *simp*
from *s b* **show** $s \in P \cap b$ **by** *simp*
qed
next
assume *nb*: $s \notin b$
from *case-nb* **show** *?thesis*
proof
from *sem nb* **show** $Sem\ c2\ s\ s'$ **by** *simp*
from *s nb* **show** $s \in P \cap -b$ **by** *simp*
qed
qed
qed

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
  fix $s\ s'$ **assume** *s*: $s \in P$
  assume $\mathit{Sem}\ (\mathit{While}\ b\ X\ Y\ c)\ s\ s'$
  then obtain *n* **where** $\mathit{iter}\ n\ b\ (\mathit{Sem}\ c)\ s\ s'$ **by** *auto*
  from *this* **and** *s* **show** $s' \in P \cap -b$
  proof (*induct n arbitrary*: *s*)
    case *0*
    then show *?case* **by** *auto*
  next
    case (*Suc n*)
    then obtain s'' **where** *b*: $s \in b$ **and** *sem*: $\mathit{Sem}\ c\ s\ s''$
      and *iter*: $\mathit{iter}\ n\ b\ (\mathit{Sem}\ c)\ s''\ s'$ **by** *auto*
    from *Suc* **and** *b* **have** $s \in P \cap b$ **by** *simp*
    with *body sem* **have** $s'' \in P$ **..**
    with *iter* **show** *?case* **by** (*rule Suc*)
  qed
qed

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 :: $'b \Rightarrow ('a \Rightarrow 'b)$
  -antiquote :: $('a \Rightarrow 'b) \Rightarrow 'b$ (‹´-› [*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*

parse-translation ‹
let
fun quote-tr [*t*] = *Syntax-Trans.quote-tr* ***syntax-const*** ‹*-antiquote*› *t*
| *quote-tr ts* = *raise TERM* (*quote-tr*, *ts*);
in [(***syntax-const*** ‹*-quote*›, *K quote-tr*)] *end*
›

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.

print-translation ‹
let
fun quote-tr' f (*t* :: *ts*) =
Term.list-comb (*f* $ *Syntax-Trans.quote-tr'* ***syntax-const*** ‹*-antiquote*› *t*, *ts*)
| *quote-tr'* - - = *raise Match*;

val assert-tr' = *quote-tr'* (*Syntax.const* ***syntax-const*** ‹*-Assert*›);

fun bexp-tr' name ((*Const* (***const-syntax*** ‹*Collect*›, -) $ *t*) :: *ts*) =
quote-tr' (*Syntax.const name*) (*t* :: *ts*)
| *bexp-tr'* - - = *raise Match*;

fun assign-tr' (*Abs* (*x*, -, *f* $ *k* $ *Bound 0*) :: *ts*) =
quote-tr' (*Syntax.const* ***syntax-const*** ‹*-Assign*› $ *Syntax-Trans.update-name-tr' f*)
(*Abs* (*x*, *dummyT*, *Syntax-Trans.const-abs-tr' k*) :: *ts*)
| *assign-tr'* - = *raise Match*;
in
[(***const-syntax*** ‹*Collect*›, *K assert-tr'*),
(***const-syntax*** ‹*Basic*›, *K assign-tr'*),
(***const-syntax*** ‹*Cond*›, *K* (*bexp-tr'* ***syntax-const*** ‹*-Cond*›)),

$(\textit{\textbf{const-syntax}}\ \langle\textit{While}\rangle,\ K\ (\textit{bexp-tr}'\ \textit{\textbf{syntax-const}}\ \langle\textit{-While-inv}\rangle))]$
  end
›

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*]: $\vdash P\ c\ Q \Longrightarrow P' \subseteq P \Longrightarrow \vdash P'\ c\ Q$
  by (*unfold Valid-def*) *blast*
lemma [*trans*] : $P' \subseteq P \Longrightarrow \vdash P\ c\ Q \Longrightarrow \vdash P'\ c\ Q$
  by (*unfold Valid-def*) *blast*

lemma [*trans*]: $Q \subseteq Q' \Longrightarrow \vdash P\ c\ Q \Longrightarrow \vdash P\ c\ Q'$
  by (*unfold Valid-def*) *blast*
lemma [*trans*]: $\vdash P\ c\ Q \Longrightarrow Q \subseteq Q' \Longrightarrow \vdash P\ c\ Q'$
  by (*unfold Valid-def*) *blast*

lemma [*trans*]:
  $\vdash \{\!|\,{}'P|\!\}\ c\ Q \Longrightarrow (\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \vdash \{\!|\,{}'P'|\!\}\ c\ Q$
  by (*simp add*: *Valid-def*)
lemma [*trans*]:
  $(\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \vdash \{\!|\,{}'P|\!\}\ c\ Q \Longrightarrow \vdash \{\!|\,{}'P'|\!\}\ c\ Q$
  by (*simp add*: *Valid-def*)

lemma [*trans*]:
  $\vdash P\ c\ \{\!|\,{}'Q|\!\} \Longrightarrow (\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \vdash P\ c\ \{\!|\,{}'Q'|\!\}$
  by (*simp add*: *Valid-def*)
lemma [*trans*]:
  $(\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \vdash P\ c\ \{\!|\,{}'Q|\!\} \Longrightarrow \vdash P\ c\ \{\!|\,{}'Q'|\!\}$
  by (*simp add*: *Valid-def*)

Identity and basic assignments.[6]

lemma *skip* [*intro?*]: $\vdash P\ \textit{SKIP}\ P$
proof −
  have $\vdash \{s.\ \textit{id}\ s \in P\}\ \textit{SKIP}\ P$ **by** (*rule basic*)
  then show *?thesis* **by** *simp*
qed

lemma *assign*: $\vdash P\ [{}'a/{}'x{::}'a]\ {}'x := {}'a\ P$
  by (*rule basic*)

[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.

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 $\acute{}x := \acute{}a$ 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*]: $\vdash P\ c1;(c2;c3)\ Q \longleftrightarrow \vdash P\ (c1;c2);c3\ Q$
  by (*auto simp add*: *Valid-def*)

Conditional statements.

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

lemma [*trans*, *intro?*]:
  $\vdash \{\!|\acute{}P \wedge \acute{}b|\!\}\ c1\ Q$
      $\Longrightarrow \vdash \{\!|\acute{}P \wedge \neg\ \acute{}b|\!\}\ c2\ Q$
      $\Longrightarrow \vdash \{\!|\acute{}P|\!\}\ IF\ \acute{}b\ THEN\ c1\ ELSE\ c2\ FI\ Q$
    by (*rule cond*) (*simp-all add*: *Valid-def*)

While statements — with optional invariant.

lemma [*intro?*]: $\vdash (P \cap b)\ c\ P \Longrightarrow \vdash P\ (While\ b\ P\ V\ c)\ (P \cap -b)$
  by (*rule while*)

lemma [*intro?*]: $\vdash (P \cap b)\ c\ P \Longrightarrow \vdash P\ (While\ b\ undefined\ V\ c)\ (P \cap -b)$
  by (*rule while*)

lemma [*intro?*]:
  $\vdash \{\!|\acute{}P \wedge \acute{}b|\!\}\ c\ \{\!|\acute{}P|\!\}$
    $\Longrightarrow \vdash \{\!|\acute{}P|\!\}\ WHILE\ \acute{}b\ INV\ \{\!|\acute{}P|\!\}\ DO\ c\ OD\ \{\!|\acute{}P \wedge \neg\ \acute{}b|\!\}$
  by (*simp add*: *while Collect-conj-eq Collect-neg-eq*)

lemma [*intro?*]:
  $\vdash \{\!|\acute{}P \wedge \acute{}b|\!\}\ c\ \{\!|\acute{}P|\!\}$
    $\Longrightarrow \vdash \{\!|\acute{}P|\!\}\ WHILE\ \acute{}b\ DO\ c\ OD\ \{\!|\acute{}P \wedge \neg\ \acute{}b|\!\}$
  by (*simp add*: *while Collect-conj-eq Collect-neg-eq*)

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

[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.

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*: $p \subseteq q \Longrightarrow Valid\ p\ (Basic\ id)\ q$
  by (*auto simp add*: *Valid-def*)

lemma *BasicRule*: $p \subseteq \{s.\ f\ s \in q\} \Longrightarrow Valid\ p\ (Basic\ f)\ q$
  by (*auto simp*: *Valid-def*)

lemma *SeqRule*: $Valid\ P\ c1\ Q \Longrightarrow Valid\ Q\ c2\ R \Longrightarrow Valid\ P\ (c1;c2)\ R$
  by (*auto simp*: *Valid-def*)

lemma *CondRule*:
  $p \subseteq \{s.\ (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}$
    $\Longrightarrow Valid\ w\ c1\ q \Longrightarrow Valid\ w'\ c2\ q \Longrightarrow Valid\ p\ (Cond\ b\ c1\ c2)\ q$
  by (*auto simp*: *Valid-def*)

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

lemma *WhileRule*:
    $p \subseteq i \Longrightarrow Valid\ (i \cap b)\ c\ i \Longrightarrow i \cap (-b) \subseteq q \Longrightarrow Valid\ p\ (While\ b\ i\ v\ c)\ q$
  apply (*clarsimp simp*: *Valid-def*)
  apply (*drule iter-aux*)
    prefer *2*
    apply *assumption*
   apply *blast*
  apply *blast*
  done

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*]

method-setup *hoare* =
  ‹*Scan.succeed* (*fn ctxt* =>
    (*SIMPLE-METHOD′*
      (*Hoare-Tac.hoare-tac ctxt*
      (*simp-tac* (*put-simpset HOL-basic-ss ctxt addsimps* [@{*thm Record.K-record-comp*}]
)))))›
  verification condition generator for Hoare logic

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 $\vdash \{\!|\acute{}(N\text{-}update\ (\lambda\text{-}.\ (2 * \acute{}N))) \in \{\!|\acute{}N = 10|\!\}|\!\}\ \acute{}N := 2 * \acute{}N\ \{\!|\acute{}N = 10|\!\}$
  by (*rule assign*)

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 $\vdash \{\!|True|\!\}\ \acute{}N := 10\ \{\!|\acute{}N = 10|\!\}$
  by *hoare*

lemma $\vdash \{\!|2 * \acute{}N = 10|\!\}\ \acute{}N := 2 * \acute{}N\ \{\!|\acute{}N = 10|\!\}$
  by *hoare*

lemma $\vdash \{\!|\acute{}N = 5|\!\}\ \acute{}N := 2 * \acute{}N\ \{\!|\acute{}N = 10|\!\}$
  by *hoare simp*

lemma $\vdash \{\!|\acute{}N + 1 = a + 1|\!\}\ \acute{}N := \acute{}N + 1\ \{\!|\acute{}N = a + 1|\!\}$
  by *hoare*

lemma $\vdash \{\!|\acute{}N = a|\!\}\ \acute{}N := \acute{}N + 1\ \{\!|\acute{}N = a + 1|\!\}$
  by *hoare simp*

lemma $\vdash \{\!|a = a \wedge b = b|\!\}\ \acute{}M := a;\ \acute{}N := b\ \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$
  by *hoare*

lemma $\vdash \{\!|\mathit{True}|\!\}\ \acute{}M := a;\ \acute{}N := b\ \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$
  by *hoare*

lemma
  $\vdash \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$
    $\acute{}I := \acute{}M;\ \acute{}M := \acute{}N;\ \acute{}N := \acute{}I$
    $\{\!|\acute{}M = b \wedge \acute{}N = a|\!\}$
  by *hoare simp*

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 $\vdash \{\!|\acute{}N = a|\!\}\ \acute{}N := \acute{}N\ \{\!|\acute{}N = a|\!\}$
  by *hoare*

lemma $\vdash \{\!|\acute{}x = a|\!\}\ \acute{}x := \acute{}x\ \{\!|\acute{}x = a|\!\}$
  oops

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

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 $\vdash \{\!|\acute{}M = \acute{}N|\!\}\ \acute{}M := \acute{}M + 1\ \{\!|\acute{}M \neq \acute{}N|\!\}$
proof −
  have $\{\!|\acute{}M = \acute{}N|\!\} \subseteq \{\!|\acute{}M + 1 \neq \acute{}N|\!\}$
    by *auto*
  also have $\vdash \ldots\ \acute{}M := \acute{}M + 1\ \{\!|\acute{}M \neq \acute{}N|\!\}$
    by *hoare*
  finally show *?thesis* **.**
qed

lemma $\vdash \{\!|\acute{}M = \acute{}N|\!\}\ \acute{}M := \acute{}M + 1\ \{\!|\acute{}M \neq \acute{}N|\!\}$
proof −
  have $m = n \longrightarrow m + 1 \neq n$ **for** $m\ n :: \mathit{nat}$

— inclusion of assertions expressed in "pure" logic,
— without mentioning the state space
by *simp*
also have $\vdash \{\!|\,\acute{}M + 1 \neq \acute{}N|\!\}\ \acute{}M := \acute{}M + 1\ \{\!|\,\acute{}M \neq \acute{}N|\!\}$
by *hoare*
finally show *?thesis* **.**
qed

lemma $\vdash \{\!|\,\acute{}M = \acute{}N|\!\}\ \acute{}M := \acute{}M + 1\ \{\!|\,\acute{}M \neq \acute{}N|\!\}$
by *hoare simp*

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
$\vdash \{\!|\,\acute{}M = 0 \wedge \acute{}S = 0|\!\}$
$WHILE\ \acute{}M \neq a$
$DO\ \acute{}S := \acute{}S + b;\ \acute{}M := \acute{}M + 1\ OD$
$\{\!|\,\acute{}S = a * b|\!\}$
proof $-$
let $\vdash$ - *?while* - = *?thesis*
let $\{\!|\,\acute{}?inv|\!\} = \{\!|\,\acute{}S = \acute{}M * b|\!\}$

have $\{\!|\,\acute{}M = 0 \wedge \acute{}S = 0|\!\} \subseteq \{\!|\,\acute{}?inv|\!\}$ **by** *auto*
also have $\vdash \ldots\ ?while\ \{\!|\,\acute{}?inv \wedge \neg\ (\acute{}M \neq a)|\!\}$
proof
let $?c = \acute{}S := \acute{}S + b;\ \acute{}M := \acute{}M + 1$
have $\{\!|\,\acute{}?inv \wedge \acute{}M \neq a|\!\} \subseteq \{\!|\,\acute{}S + b = (\acute{}M + 1) * b|\!\}$
by *auto*
also have $\vdash \ldots\ ?c\ \{\!|\,\acute{}?inv|\!\}$ **by** *hoare*
finally show $\vdash \{\!|\,\acute{}?inv \wedge \acute{}M \neq a|\!\}\ ?c\ \{\!|\,\acute{}?inv|\!\}$ **.**
qed
also have $\ldots \subseteq \{\!|\,\acute{}S = a * b|\!\}$ **by** *auto*
finally show *?thesis* **.**
qed

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

by *hoare auto*

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
  ⊢ {|True|}
      ´S := 0; ´I := 1;
      WHILE ´I ≠ n
      DO
        ´S := ´S + ´I;
        ´I := ´I + 1
      OD
      {|´S = (∑j<n. j)|}
  (is ⊢ - (-; ?while) -)
proof −
  let ?sum = λk::nat. ∑j<k. j
  let ?inv = λs i::nat. s = ?sum i

  have ⊢ {|True|} ´S := 0; ´I := 1 {|?inv ´S ´I|}
  proof −
    have True ⟶ 0 = ?sum 1
      by simp
    also have ⊢ {|...|} ´S := 0; ´I := 1 {|?inv ´S ´I|}
      by hoare
    finally show ?thesis .
  qed
  also have ⊢ ... ?while {|?inv ´S ´I ∧ ¬ ´I ≠ n|}
  proof
    let ?body = ´S := ´S + ´I; ´I := ´I + 1
    have ?inv s i ∧ i ≠ n ⟶ ?inv (s + i) (i + 1) for s i
      by simp
    also have ⊢ {|´S + ´I = ?sum (´I + 1)|} ?body {|?inv ´S ´I|}
      by hoare
    finally show ⊢ {|?inv ´S ´I ∧ ´I ≠ n|} ?body {|?inv ´S ´I|} .
  qed
  also have s = ?sum i ∧ ¬ i ≠ n ⟶ s = ?sum n for s i
    by simp
  finally show ?thesis .
qed
```

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

```
theorem
  ⊢ {|True|}
      ´S := 0; ´I := 1;
      WHILE ´I ≠ n
      INV {|´S = (∑j<´I. j)|}
      DO
        ´S := ´S + ´I;
        ´I := ´I + 1
      OD
      {|´S = (∑j<n. j)|}
proof −
  let ?sum = λk::nat. ∑j<k. j
  let ?inv = λs i::nat. s = ?sum i
  show ?thesis
  proof hoare
    show ?inv 0 1 by simp
    show ?inv (s + i) (i + 1) if ?inv s i ∧ i ≠ n for s i
      using that by simp
    show s = ?sum n if ?inv s i ∧ ¬ i ≠ n for s i
      using that by simp
  qed
qed
```

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

```
theorem
  ⊢ {|True|}
      ´S := 0; ´I := 1;
      WHILE ´I ≠ n
      INV {|´S = (∑j<´I. j)|}
      DO
        ´S := ´S + ´I;
        ´I := ´I + 1
      OD
      {|´S = (∑j<n. j)|}
  by hoare auto
```

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 = (|time :: nat, . . . :: 'a|)

primrec timeit :: 'a time com ⇒ 'a time com
  where
    timeit (Basic f) = (Basic f; Basic(λs. 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)
  by (induct n) simp-all

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|}
  apply simp
  apply hoare
      apply simp
     apply clarsimp
    apply clarsimp
   apply arith
   prefer 2
   apply clarsimp
  apply (clarsimp simp: nat-distrib)
  apply (frule lem)
  apply arith
  done

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 ∈ tiling A
    and u ∈ tiling A
    and t ∩ u = {}
  shows t ∪ u ∈ tiling A
proof −
  let ?T = tiling A
  from ‹t ∈ ?T› and ‹t ∩ u = {}›
  show t ∪ u ∈ ?T
  proof (induct t)
    case empty
    with ‹u ∈ ?T› show {} ∪ u ∈ ?T by simp
  next
    case (Un a t)
    show (a ∪ t) ∪ u ∈ ?T
    proof −
      have a ∪ (t ∪ u) ∈ ?T
        using ‹a ∈ A›
      proof (rule tiling.Un)
        from ‹(a ∪ t) ∩ u = {}› have t ∩ u = {} by blast
        then show t ∪ u ∈ ?T by (rule Un)
        from ‹a ⊆ − t› and ‹(a ∪ t) ∩ u = {}›
        show a ⊆ − (t ∪ u) by blast
      qed
      also have a ∪ (t ∪ u) = (a ∪ t) ∪ u
        by (simp only: Un-assoc)
      finally show ?thesis .
    qed
  qed
qed
```

10.2 Basic properties of "below"

```
definition below :: nat ⇒ nat set
  where below n = {i. i < n}

lemma below-less-iff [iff]: i ∈ below k ⟷ i < k
  by (simp add: below-def)

lemma below-0: below 0 = {}
  by (simp add: below-def)
```

lemma *Sigma-Suc1*: $m = n + 1 \Longrightarrow below\ m \times B = (\{n\} \times B) \cup (below\ n \times B)$
  by (*simp add*: *below-def less-Suc-eq*) *blast*

lemma *Sigma-Suc2*:
  $m = n + 2 \Longrightarrow$
    $A \times below\ m = (A \times \{n\}) \cup (A \times \{n + 1\}) \cup (A \times below\ n)$
  by (*auto simp add*: *below-def*)

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

10.3 Basic properties of "evnodd"

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

lemma *evnodd-iff*: $(i, j) \in evnodd\ A\ b \longleftrightarrow (i, j) \in A\ \wedge (i + j)\ mod\ 2 = b$
  by (*simp add*: *evnodd-def*)

lemma *evnodd-subset*: $evnodd\ A\ b \subseteq A$
  unfolding *evnodd-def* **by** (*rule Int-lower1*)

lemma *evnoddD*: $x \in evnodd\ A\ b \Longrightarrow x \in A$
  by (*rule subsetD*) (*rule evnodd-subset*)

lemma *evnodd-finite*: $finite\ A \Longrightarrow finite\ (evnodd\ A\ b)$
  by (*rule finite-subset*) (*rule evnodd-subset*)

lemma *evnodd-Un*: $evnodd\ (A \cup B)\ b = evnodd\ A\ b \cup evnodd\ B\ b$
  unfolding *evnodd-def* **by** *blast*

lemma *evnodd-Diff*: $evnodd\ (A - B)\ b = evnodd\ A\ b - evnodd\ B\ b$
  unfolding *evnodd-def* **by** *blast*

lemma *evnodd-empty*: $evnodd\ \{\}\ b = \{\}$
  by (*simp add*: *evnodd-def*)

lemma *evnodd-insert*: $evnodd\ (insert\ (i, j)\ C)\ b =$
    $(if\ (i + j)\ mod\ 2 = b$
      $then\ insert\ (i, j)\ (evnodd\ C\ b)\ else\ evnodd\ C\ b)$
  by (*simp add*: *evnodd-def*)

10.4 Dominoes

inductive-set *domino* :: $(nat \times nat)\ set\ set$
  where
    horiz: $\{(i, j), (i, j + 1)\} \in domino$
  | *vertl*: $\{(i, j), (i + 1, j)\} \in domino$

lemma *dominoes-tile-row*:
  $\{i\} \times below\ (2 * n) \in tiling\ domino$

```
  (is ?B n ∈ ?T)
proof (induct n)
  case 0
  show ?case by (simp add: below-0 tiling.empty)
next
  case (Suc n)
  let ?a = {i} × {2 ∗ n + 1} ∪ {i} × {2 ∗ n}
  have ?B (Suc n) = ?a ∪ ?B n
    by (auto simp add: Sigma-Suc Un-assoc)
  also have ... ∈ ?T
  proof (rule tiling.Un)
    have {(i, 2 ∗ n), (i, 2 ∗ n + 1)} ∈ domino
      by (rule domino.horiz)
    also have {(i, 2 ∗ n), (i, 2 ∗ n + 1)} = ?a by blast
    finally show ... ∈ domino .
    show ?B n ∈ ?T by (rule Suc)
    show ?a ⊆ − ?B n by blast
  qed
  finally show ?case .
qed

lemma dominoes-tile-matrix:
  below m × below (2 ∗ n) ∈ tiling domino
  (is ?B m ∈ ?T)
proof (induct m)
  case 0
  show ?case by (simp add: below-0 tiling.empty)
next
  case (Suc m)
  let ?t = {m} × below (2 ∗ n)
  have ?B (Suc m) = ?t ∪ ?B m by (simp add: Sigma-Suc)
  also have ... ∈ ?T
  proof (rule tiling-Un)
    show ?t ∈ ?T by (rule dominoes-tile-row)
    show ?B m ∈ ?T by (rule Suc)
    show ?t ∩ ?B m = {} by blast
  qed
  finally show ?case .
qed

lemma domino-singleton:
  assumes d ∈ domino
    and b < 2
  shows ∃ i j. evnodd d b = {(i, j)}  (is ?P d)
  using assms
proof induct
  from ‹b < 2› have b-cases: b = 0 ∨ b = 1 by arith
  fix i j
  note [simp] = evnodd-empty evnodd-insert mod-Suc
```

  from *b-cases* **show** *?P* {(*i*, *j*), (*i*, *j* + *1*)} **by** *rule auto*
  from *b-cases* **show** *?P* {(*i*, *j*), (*i* + *1*, *j*)} **by** *rule auto*
qed

lemma *domino-finite*:
  assumes *d* ∈ *domino*
  shows *finite d*
  using *assms*
proof *induct*
  fix *i j* :: *nat*
  show *finite* {(*i*, *j*), (*i*, *j* + *1*)} **by** (*intro finite.intros*)
  show *finite* {(*i*, *j*), (*i* + *1*, *j*)} **by** (*intro finite.intros*)
qed

10.5 Tilings of dominoes

lemma *tiling-domino-finite*:
  assumes *t*: *t* ∈ *tiling domino* (**is** *t* ∈ *?T*)
  shows *finite t* (**is** *?F t*)
  using *t*
proof *induct*
  show *?F* {} **by** (*rule finite.emptyI*)
  fix *a t* **assume** *?F t*
  assume *a* ∈ *domino*
  then have *?F a* **by** (*rule domino-finite*)
  from *this* **and** ‹*?F t*› **show** *?F* (*a* ∪ *t*) **by** (*rule finite-UnI*)
qed

lemma *tiling-domino-01*:
  assumes *t*: *t* ∈ *tiling domino* (**is** *t* ∈ *?T*)
  shows *card* (*evnodd t 0*) = *card* (*evnodd t 1*)
  using *t*
proof *induct*
  case *empty*
  show *?case* **by** (*simp add*: *evnodd-def*)
next
  case (*Un a t*)
  let *?e* = *evnodd*
  note *hyp* = ‹*card* (*?e t 0*) = *card* (*?e t 1*)›
    and *at* = ‹*a* ⊆ − *t*›
  have *card-suc*: *card* (*?e* (*a* ∪ *t*) *b*) = *Suc* (*card* (*?e t b*)) **if** *b* < *2* **for** *b* :: *nat*
  proof −
    have *?e* (*a* ∪ *t*) *b* = *?e a b* ∪ *?e t b* **by** (*rule evnodd-Un*)
    also obtain *i j* **where** *e*: *?e a b* = {(*i*, *j*)}
    proof −
      from ‹*a* ∈ *domino*› **and** ‹*b* < *2*›
      have ∃ *i j*. *?e a b* = {(*i*, *j*)} **by** (*rule domino-singleton*)
      then show *?thesis* **by** (*blast intro*: *that*)
    qed

also have ... ∪ *?e t b* = *insert* (*i*, *j*) (*?e t b*) **by** *simp*
also have *card* ... = *Suc* (*card* (*?e t b*))
proof (*rule card-insert-disjoint*)
from ‹*t* ∈ *tiling domino*› **have** *finite t*
by (*rule tiling-domino-finite*)
then show *finite* (*?e t b*)
by (*rule evnodd-finite*)
from *e* **have** (*i*, *j*) ∈ *?e a b* **by** *simp*
with *at* **show** (*i*, *j*) ∉ *?e t b* **by** (*blast dest*: *evnoddD*)
qed
finally show *?thesis* **.**
qed
then have *card* (*?e* (*a* ∪ *t*) *0*) = *Suc* (*card* (*?e t 0*)) **by** *simp*
also from *hyp* **have** *card* (*?e t 0*) = *card* (*?e t 1*) **.**
also from *card-suc* **have** *Suc* ... = *card* (*?e* (*a* ∪ *t*) *1*)
by *simp*
finally show *?case* **.**
qed

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 (*unfold mutilated-board-def*)
let *?T* = *tiling domino*
let *?t* = *below* (*2* ∗ (*m* + *1*)) × *below* (*2* ∗ (*n* + *1*))
let *?t′* = *?t* − {(*0*, *0*)}
let *?t″* = *?t′* − {(*2* ∗ *m* + *1*, *2* ∗ *n* + *1*)}

show *?t″* ∉ *?T*
proof
have *t*: *?t* ∈ *?T* **by** (*rule dominoes-tile-matrix*)
assume *t″*: *?t″* ∈ *?T*

let *?e* = *evnodd*
have *fin*: *finite* (*?e ?t 0*)
by (*rule evnodd-finite*, *rule tiling-domino-finite*, *rule t*)

note [*simp*] = *evnodd-iff evnodd-empty evnodd-insert evnodd-Diff*
have *card* (*?e ?t″ 0*) < *card* (*?e ?t′ 0*)
proof −
have *card* (*?e ?t′ 0* − {(*2* ∗ *m* + *1*, *2* ∗ *n* + *1*)})
< *card* (*?e ?t′ 0*)
proof (*rule card-Diff1-less*)
from - *fin* **show** *finite* (*?e ?t′ 0*)

by (*rule finite-subset*) *auto*
show $(2 * m + 1, 2 * n + 1) \in ?e\ ?t'\ 0$ **by** *simp*
qed
then show *?thesis* **by** *simp*
qed
also have $\ldots < card\ (?e\ ?t\ 0)$
proof −
have $(0, 0) \in ?e\ ?t\ 0$ **by** *simp*
with *fin* **have** $card\ (?e\ ?t\ 0 - \{(0, 0)\}) < card\ (?e\ ?t\ 0)$
by (*rule card-Diff1-less*)
then show *?thesis* **by** *simp*
qed
also from *t* **have** $\ldots = card\ (?e\ ?t\ 1)$
by (*rule tiling-domino-01*)
also have $?e\ ?t\ 1 = ?e\ ?t''\ 1$ **by** *simp*
also from t'' **have** $card\ \ldots = card\ (?e\ ?t''\ 0)$
by (*rule tiling-domino-01* [*symmetric*])
finally have $\ldots < \ldots$ **. then show** *False* **..**
qed
qed

end

11 An old chestnut

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

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

theorem
assumes *f-ax*: $\bigwedge n.\ f\ (f\ n) < f\ (Suc\ n)$
shows $f\ n = n$
proof (*rule order-antisym*)
show *ge*: $n \le f\ n$ **for** n
proof (*induct f n arbitrary*: *n rule*: *less-induct*)
case *less*
show $n \le f\ n$
proof (*cases n*)
case (*Suc m*)
from *f-ax* **have** $f\ (f\ m) < f\ n$ **by** (*simp only*: *Suc*)
with *less* **have** $f\ m \le f\ (f\ m)$ **.**
also from *f-ax* **have** $\ldots < f\ n$ **by** (*simp only*: *Suc*)
finally have $f\ m < f\ n$ **.**
with *less* **have** $m \le f\ m$ **.**

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

```
      also note ‹... < f n›
      finally have m < f n .
      then have n ≤ f n by (simp only: Suc)
      then show ?thesis .
    next
      case 0
      then show ?thesis by simp
    qed
  qed

  have mono: m ≤ n ⟹ f m ≤ f n for m n :: nat
  proof (induct n)
    case 0
    then have m = 0 by simp
    then show ?case by simp
  next
    case (Suc n)
    from Suc.prems show f m ≤ f (Suc n)
    proof (rule le-SucE)
      assume m ≤ n
      with Suc.hyps have f m ≤ f n .
      also from ge f-ax have ... < f (Suc n)
        by (rule le-less-trans)
      finally show ?thesis by simp
    next
      assume m = Suc n
      then show ?thesis by simp
    qed
  qed

  show f n ≤ n
  proof −
    have ¬ n < f n
    proof
      assume n < f n
      then have Suc n ≤ f n by simp
      then have f (Suc n) ≤ f (f n) by (rule mono)
      also have ... < f (Suc n) by (rule f-ax)
      finally have ... < ... . then show False ..
    qed
    then show ?thesis by simp
  qed
qed

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*:
  $2 * (\sum i::nat=0..n.\ i) = n * (n + 1)$
  (**is** $?P\ n$ **is** $?S\ n = \text{-}$)
proof ($induct\ n$)
  show $?P\ 0$ **by** *simp*
next
  fix n **have** $?S\ (n + 1) = ?S\ n + 2 * (n + 1)$
    by *simp*
  also assume $?S\ n = n * (n + 1)$
  also have $\ldots + 2 * (n + 1) = (n + 1) * (n + 2)$
    by *simp*
  finally show $?P\ (Suc\ n)$
    by *simp*
qed

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 "$\ldots$" 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 (induct n)
  show ?P 0 by simp
next
  fix n
  have ?S (n + 1) = ?S n + 2 * n + 1
    by simp
  also assume ?S n = n^Suc (Suc 0)
  also have ... + 2 * n + 1 = (n + 1)^Suc (Suc 0)
    by simp
  finally show ?P (Suc n)
    by simp
qed
```

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 (induct n)
  show ?P 0 by simp
next
  fix n
  have ?S (n + 1) = ?S n + 6 * (n + 1)^Suc (Suc 0)
    by (simp add: distrib)
  also assume ?S n = n * (n + 1) * (2 * n + 1)
  also have ... + 6 * (n + 1)^Suc (Suc 0) =
      (n + 1) * (n + 2) * (2 * (n + 1) + 1)
    by (simp add: distrib)
  finally show ?P (Suc n)
    by simp
qed

theorem sum-of-cubes:
  4 * (∑ i::nat=0..n. i^3) = (n * (n + 1))^Suc (Suc 0)
  (is ?P n is ?S n = -)
proof (induct n)
  show ?P 0 by (simp add: power-eq-if)
```

next
  fix n
  have $?S\ (n + 1) = ?S\ n + 4 * (n + 1)\hat{\ }3$
    by (*simp add*: *power-eq-if distrib*)
  also assume $?S\ n = (n * (n + 1))\hat{\ }Suc\ (Suc\ 0)$
  also have $\ldots + 4 * (n + 1)\hat{\ }3 = ((n + 1) * ((n + 1) + 1))\hat{\ }Suc\ (Suc\ 0)$
    by (*simp add*: *power-eq-if distrib*)
  finally show $?P\ (Suc\ n)$
    by *simp*
qed

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.