Miscellaneous Isabelle/Isar examples

Makarius Wenzel

With contributions by Gertrud Bauer and Tobias Nipkow

June 9, 2019

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 Textbook-style reasoning: the Knaster-Tarski Theorem

theory *Knaster-Tarski*
  imports *Main HOL−Library.Lattice-Syntax*
begin

1.1 Prose version

According to the textbook [2, pages 93–94], the Knaster-Tarski fixpoint theorem is as follows.[1]

[1] We have dualized the argument, and tuned the notation a little bit.

The Knaster-Tarski Fixpoint Theorem. Let L be a complete lattice and $f\colon L \to L$ an order-preserving map. Then $\bigsqcap\{x \in L \mid f(x) \leq x\}$ is a fixpoint of f.

Proof. Let $H = \{x \in L \mid f(x) \leq x\}$ and $a = \bigsqcap H$. For all $x \in H$ we have $a \leq x$, so $f(a) \leq f(x) \leq x$. Thus $f(a)$ is a lower bound of H, whence $f(a) \leq a$. We now use this inequality to prove the reverse one (!) and thereby complete the proof that a is a fixpoint. Since f is order-preserving, $f(f(a)) \leq f(a)$. This says $f(a) \in H$, so $a \leq f(a)$.

1.2 Formal versions

The Isar proof below closely follows the original presentation. Virtually all of the prose narration has been rephrased in terms of formal Isar language elements. Just as many textbook-style proofs, there is a strong bias towards forward proof, and several bends in the course of reasoning.

```
theorem Knaster-Tarski:
  fixes f :: 'a::complete-lattice ⇒ 'a
  assumes mono f
  shows ∃ a. f a = a
proof
  let ?H = {u. f u ≤ u}
  let ?a = ⨅ ?H
  show f ?a = ?a
  proof −
    {
      fix x
      assume x ∈ ?H
      then have ?a ≤ x by (rule Inf-lower)
      with ‹mono f› have f ?a ≤ f x ..
      also from ‹x ∈ ?H› have . . . ≤ x ..
      finally have f ?a ≤ x .
    }
    then have f ?a ≤ ?a by (rule Inf-greatest)
    {
      also presume . . . ≤ f ?a
      finally (order-antisym) show ?thesis .
    }
    from ‹mono f› and ‹f ?a ≤ ?a› have f (f ?a) ≤ f ?a ..
    then have f ?a ∈ ?H ..
    then show ?a ≤ f ?a by (rule Inf-lower)
  qed
qed
```

Above we have used several advanced Isar language elements, such as explicit block structure and weak assumptions. Thus we have mimicked the particular way of reasoning of the original text.

In the subsequent version the order of reasoning is changed to achieve structured top-down decomposition of the problem at the outer level, while only the inner steps of reasoning are done in a forward manner. We are certainly more at ease here, requiring only the most basic features of the Isar language.

theorem *Knaster-Tarski′*:
  fixes f :: $'a$::*complete-lattice* $\Rightarrow$ $'a$
  assumes *mono f*
  shows $\exists\, a.\ f\ a = a$
proof
  let $?H = \{u.\ f\ u \le u\}$
  let $?a = \bigsqcap\ ?H$
  show $f\ ?a = ?a$
  proof (*rule order-antisym*)
    show $f\ ?a \le\ ?a$
    proof (*rule Inf-greatest*)
      fix x
      assume $x \in\ ?H$
      then have $?a \le x$ **by** (*rule Inf-lower*)
      with ‹*mono f*› **have** $f\ ?a \le f\ x$ **..**
      also from ‹$x \in\ ?H$› **have** $\ldots \le x$ **..**
      finally show $f\ ?a \le x$ **.**
    qed
    show $?a \le f\ ?a$
    proof (*rule Inf-lower*)
      from ‹*mono f*› **and** ‹$f\ ?a \le\ ?a$› **have** $f\ (f\ ?a) \le f\ ?a$ **..**
      then show $f\ ?a \in\ ?H$ **..**
    qed
  qed
qed

end

2 Peirce's Law

theory *Peirce*
  imports *Main*
begin

We consider Peirce's Law: $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$. This is an inherently non-intuitionistic statement, so its proof will certainly involve some form of classical contradiction.

The first proof is again a well-balanced combination of plain backward and forward reasoning. The actual classical step is where the negated goal may be introduced as additional assumption. This eventually leads to a contradiction.[2]

[2]The rule involved there is negation elimination; it holds in intuitionistic logic as well.

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

In the subsequent version the reasoning is rearranged by means of "weak assumptions" (as introduced by **presume**). Before assuming the negated goal $\neg A$, its intended consequence $A \longrightarrow B$ is put into place in order to solve the main problem. Nevertheless, we do not get anything for free, but have to establish $A \longrightarrow B$ later on. The overall effect is that of a logical *cut*.

Technically speaking, whenever some goal is solved by **show** in the context of weak assumptions then the latter give rise to new subgoals, which may be established separately. In contrast, strong assumptions (as introduced by **assume**) are solved immediately.

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

Note that the goals stemming from weak assumptions may be even left until qed time, where they get eventually solved "by assumption" as well. In that case there is really no fundamental difference between the two kinds of assumptions, apart from the order of reducing the individual parts of the proof configuration.

Nevertheless, the "strong" mode of plain assumptions is quite important in practice to achieve robustness of proof text interpretation. By forcing both the conclusion *and* the assumptions to unify with the pending goal to be solved, goal selection becomes quite deterministic. For example, decomposition with rules of the "case-analysis" type usually gives rise to several goals that only differ in there local contexts. With strong assumptions these may be still solved in any order in a predictable way, while weak ones would quickly lead to great confusion, eventually demanding even some backtracking.

end

3 The Drinker's Principle

theory *Drinker*
  imports *Main*
begin

Here is another example of classical reasoning: the Drinker's Principle says that for some person, if he is drunk, everybody else is drunk!

We first prove a classical part of de-Morgan's law.

lemma *de-Morgan*:
  assumes $\neg (\forall x.\ P\ x)$
  shows $\exists x.\ \neg P\ x$
proof (*rule classical*)
  assume $\nexists x.\ \neg P\ x$
  have $\forall x.\ P\ x$
  proof
    fix x **show** $P\ x$
    proof (*rule classical*)
      assume $\neg P\ x$
      then have $\exists x.\ \neg P\ x$ **..**
      with $\langle \nexists x.\ \neg P\ x \rangle$ **show** *?thesis* **by** *contradiction*
    qed
  qed
  with $\langle \neg (\forall x.\ P\ x) \rangle$ **show** *?thesis* **by** *contradiction*
qed

theorem *Drinker's-Principle*: $\exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
proof *cases*
  assume $\forall x.\ drunk\ x$
  then have $drunk\ a \longrightarrow (\forall x.\ drunk\ x)$ **for** a **..**
  then show *?thesis* **..**
next
  assume $\neg (\forall x.\ drunk\ x)$
  then have $\exists x.\ \neg drunk\ x$ **by** (*rule de-Morgan*)
  then obtain a **where** $\neg drunk\ a$ **..**
  have $drunk\ a \longrightarrow (\forall x.\ drunk\ x)$

```
  proof
    assume drunk a
    with ‹¬ drunk a› show ∀x. drunk x by contradiction
  qed
  then show ?thesis ..
qed

end
```

4 Cantor's Theorem

```
theory Cantor
  imports Main
begin
```

4.1 Mathematical statement and proof

Cantor's Theorem states that there is no surjection from a set to its powerset. The proof works by diagonalization. E.g. see

- http://mathworld.wolfram.com/CantorDiagonalMethod.html
- https://en.wikipedia.org/wiki/Cantor's_diagonal_argument

```
theorem Cantor: ∄f :: 'a ⇒ 'a set. ∀A. ∃x. A = f x
proof
  assume ∃f :: 'a ⇒ 'a set. ∀A. ∃x. A = f x
  then obtain f :: 'a ⇒ 'a set where *: ∀A. ∃x. A = f x ..
  let ?D = {x. x ∉ f x}
  from * obtain a where ?D = f a by blast
  moreover have a ∈ ?D ⟷ a ∉ f a by blast
  ultimately show False by blast
qed
```

4.2 Automated proofs

These automated proofs are much shorter, but lack information why and how it works.

```
theorem ∄f :: 'a ⇒ 'a set. ∀A. ∃x. f x = A
  by best

theorem ∄f :: 'a ⇒ 'a set. ∀A. ∃x. f x = A
  by force
```

4.3 Elementary version in higher-order predicate logic

The subsequent formulation bypasses set notation of HOL; it uses elementary λ-calculus and predicate logic, with standard introduction and elimination rules. This also shows that the proof does not require classical reasoning.

lemma *iff-contradiction*:
  assumes $*$: $\neg A \longleftrightarrow A$
  shows *False*
proof (*rule notE*)
  show $\neg A$
  proof
    assume A
    with $*$ **have** $\neg A$ **..**
    from *this* **and** $\langle A \rangle$ **show** *False* **..**
  qed
  with $*$ **show** A **..**
qed

theorem *Cantor'*: $\nexists f :: 'a \Rightarrow 'a \Rightarrow bool.\ \forall A.\ \exists x.\ A = f\ x$
proof
  assume $\exists f :: 'a \Rightarrow 'a \Rightarrow bool.\ \forall A.\ \exists x.\ A = f\ x$
  then obtain $f :: 'a \Rightarrow 'a \Rightarrow bool$ **where** $*$: $\forall A.\ \exists x.\ A = f\ x$ **..**
  let $?D = \lambda x.\ \neg f\ x\ x$
  from $*$ **have** $\exists x.\ ?D = f\ x$ **..**
  then obtain a **where** $?D = f\ a$ **..**
  then have $?D\ a \longleftrightarrow f\ a\ a$ **by** (*rule arg-cong*)
  then have $\neg f\ a\ a \longleftrightarrow f\ a\ a$ **.**
  then show *False* **by** (*rule iff-contradiction*)
qed

4.4 Classic Isabelle/HOL example

The following treatment of Cantor's Theorem follows the classic example from the early 1990s, e.g. see the file `92/HOL/ex/set.ML` in Isabelle92 or [8, §18.7]. The old tactic scripts synthesize key information of the proof by refinement of schematic goal states. In contrast, the Isar proof needs to say explicitly what is proven.

Cantor's Theorem states that every set has more subsets than it has elements. It has become a favourite basic example in pure higher-order logic since it is so easily expressed:

$\forall f{::}\alpha \Rightarrow \alpha \Rightarrow bool.\ \exists S{::}\alpha \Rightarrow bool.\ \forall x{::}\alpha.\ f\ x \neq S$

Viewing types as sets, $\alpha \Rightarrow bool$ represents the powerset of α. This version of the theorem states that for every function from α to its powerset, some subset is outside its range. The Isabelle/Isar proofs below uses HOL's set theory, with the type α *set* and the operator $range :: (\alpha \Rightarrow \beta) \Rightarrow \beta\ set$.

theorem $\exists S.\ S \notin range\ (f :: 'a \Rightarrow 'a\ set)$

```
proof
  let ?S = {x. x ∉ f x}
  show ?S ∉ range f
  proof
    assume ?S ∈ range f
    then obtain y where ?S = f y ..
    then show False
    proof (rule equalityCE)
      assume y ∈ f y
      assume y ∈ ?S
      then have y ∉ f y ..
      with ‹y ∈ f y› show ?thesis by contradiction
    next
      assume y ∉ ?S
      assume y ∉ f y
      then have y ∈ ?S ..
      with ‹y ∉ ?S› show ?thesis by contradiction
    qed
  qed
qed
```

How much creativity is required? As it happens, Isabelle can prove this theorem automatically using best-first search. Depth-first search would diverge, but best-first search successfully navigates through the large search space. The context of Isabelle's classical prover contains rules for the relevant constructs of HOL's set theory.

```
theorem ∃S. S ∉ range (f :: 'a ⇒ 'a set)
  by best

end
```

5 Structured statements within Isar proofs

```
theory Structured-Statements
  imports Main
begin
```

5.1 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
```

5.2 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
```

5.3 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
```

5.4 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
```

5.5 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
```

6 Basic logical reasoning

```
theory Basic-Logic
  imports Main
begin
```

6.1 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
  assume A
  show A **by** *fact*
qed

lemma K: $A \longrightarrow B \longrightarrow A$
proof
  assume A
  show $B \longrightarrow A$
  proof
    show A **by** *fact*
  qed
qed

lemma S: $(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow A \longrightarrow C$
proof
  assume $A \longrightarrow B \longrightarrow C$
  show $(A \longrightarrow B) \longrightarrow A \longrightarrow C$
  proof
    assume $A \longrightarrow B$
    show $A \longrightarrow C$
    proof
      assume A
      show C
      proof (*rule mp*)
        show $B \longrightarrow C$ **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[3]. 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

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

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.[4]

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.[5]

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.

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*.

6.2 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 $A \wedge B \longrightarrow B \wedge A$.

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

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

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 $\langle A \wedge B \rangle$ **show** B **..**
    from $\langle A \wedge B \rangle$ **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 $A \wedge B$
  then show $B \wedge A$
  proof — rule *conjE* of $A \wedge B$
    assume B A
    then show *?thesis* **..** — rule *conjI* of $B \wedge A$
  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 $A \wedge B \longrightarrow B \wedge A$
proof

```
  assume A ∧ B
  from ‹A ∧ B› have A ..
  from ‹A ∧ B› have B ..
  from ‹B› ‹A› show B ∧ 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 ∧ B ⟶ B ∧ A
proof −
  {
    assume A ∧ B
    from ‹A ∧ B› have A ..
    from ‹A ∧ B› have B ..
    from ‹B› ‹A› have B ∧ 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 ∧ B ⟶ B ∧ A
proof
  assume A ∧ B
  then show B ∧ A
  proof
    assume B A
    then show ?thesis ..
  qed
qed
```

6.3 A few examples from "Introduction to Isabelle"

We rephrase some of the basic reasoning examples of [7], using HOL rather than FOL.

6.3.1 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

6.3.2 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*: $\dfrac{\exists\, x.\ A(x) \qquad \begin{array}{c}[A(x)]_x\\ \vdots\\ B\end{array}}{B}$
    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 (∃x. P (f x)) ⟶ (∃y. P y)
proof
  assume ∃x. P (f x)
  then obtain a where P (f a) ..
  then show ∃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.

6.3.3 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 ∧ B ⟹ (A ⟹ B ⟹ C) ⟹ C
proof −
  assume A ∧ B
  assume r: A ⟹ B ⟹ C
  show C
  proof (rule r)
    show A by (rule conjunct1) fact
    show B by (rule conjunct2) fact
  qed
qed

end
```

7 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.

7.1 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*

7.2 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)
```

7.3 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)
```

7.4 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*: $\bigwedge$*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

8 Fib and Gcd commute

theory *Fibonacci*
imports *HOL−Computational-Algebra.Primes*

begin[6]

8.1 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*

8.2 Fib and gcd commute

A few laws taken from [4].

lemma *fib-add*: *fib* (*n* + *k* + *1*) = *fib* (*k* + *1*) ∗ *fib* (*n* + *1*) + *fib k* ∗ *fib n*
  (**is** *?P n*)
  — see [4, 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*)
  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*

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

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 ... = gcd (fib n) (fib (m mod n))
  also from n have ... = gcd (fib n) (fib m)
    by (rule gcd-fib-mod)
  also have ... = gcd (fib m) (fib n)
    by (rule gcd.commute)
  finally show fib (gcd m n) = gcd (fib m) (fib n) .
```

qed

end

9 Basic group theory

theory *Group*
  imports *Main*
begin

9.1 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 $\ldots = 1$
    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 $\ldots = x * inverse\ x * x$
  by (*simp only*: *group-assoc*)
also have $\ldots = 1 * x$
  by (*simp only*: *group-right-inverse*)
also have $\ldots = x$
  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[7] 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 $\ldots = x * inverse\ x * x$
    by (*simp only*: *group-assoc*)

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

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

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

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

have $\ldots = x$
  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.

9.2 Groups as monoids

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

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

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 $?x * (\mathit{1}::?'a) = ?x$), we may still instantiate *group* $\subseteq$ *monoid* properly as follows.

instance *group* $\subseteq$ *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.

9.3 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 $1 = x * inverse\ x$
    by (*simp only*: *group-right-inverse*)
  also have $\ldots = (e * x) * inverse\ x$
    by (*simp only*: *eq*)
  also have $\ldots = e * (x * inverse\ x)$
    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 $inverse\ x = x'$
proof −
  have $inverse\ x = 1 * inverse\ x$
    by (*simp only*: *group-left-one*)
  also have $\ldots = (x' * x) * inverse\ x$
    by (*simp only*: *eq*)
  also have $\ldots = x' * (x * inverse\ x)$
    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*: $inverse\ (x * y) = inverse\ y * inverse\ x$
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 $\ldots = (inverse\ y * 1) * y$
      by (*simp only*: *group-left-inverse*)
    also have $\ldots = inverse\ y * y$
      by (*simp only*: *group-right-one*)
    also have $\ldots = 1$
      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
```

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

```
theory Group-Context
  imports Main
begin
```

hypothetical group axiomatization

```
context
  fixes prod :: 'a ⇒ 'a ⇒ 'a  (infixl ⊙ 70)
    and one :: 'a
    and inverse :: 'a ⇒ 'a
  assumes assoc: (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)
    and left-one: one ⊙ x = x
    and left-inverse: inverse x ⊙ x = one
begin
```

some consequences

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

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

end

end

11 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* :: $'a \Rightarrow 'a$
  assume *assoc*: $(x \odot y) \odot z = x \odot (y \odot z)$
    and *left-one*: $one \odot x = x$
    and *left-inverse*: *inverse* $x \odot x = one$
    for $x\ y\ z$

some consequences

  have *right-inverse*: $x \odot$ *inverse* $x = one$ **for** x
  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 ... = 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 ... = $x' \odot (x \odot \textit{inverse } x)$
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

12 Hoare Logic

theory *Hoare*
imports *Main*
begin

12.1 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 [12, §6]. See also `~~/src/HOL/Hoare` and [6].

type-synonym $'a$ *bexp* = $'a$ *set*
type-synonym $'a$ *assn* = $'a$ *set*

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

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

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

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

primrec *Sem* :: $'a$ *com* $\Rightarrow$ $'a$ *sem*
where
$\textit{Sem } (\textit{Basic } f)\ s\ s' \longleftrightarrow s' = f\ s$
| $\textit{Sem } (c1;\ c2)\ s\ s' \longleftrightarrow (\exists s''.\ \textit{Sem } c1\ s\ s'' \wedge \textit{Sem } c2\ s''\ s')$

| $Sem\ (Cond\ b\ c1\ c2)\ s\ s' \longleftrightarrow (if\ s \in b\ then\ Sem\ c1\ s\ s'\ else\ Sem\ c2\ s\ s')$
| $Sem\ (While\ b\ x\ c)\ s\ s' \longleftrightarrow (\exists\, n.\ iter\ n\ b\ (Sem\ c)\ s\ s')$

definition *Valid* :: *'a bexp* ⇒ *'a com* ⇒ *'a bexp* ⇒ *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*)

12.2 Primitive Hoare rules

From the semantics defined above, we derive the standard set of primitive Hoare rules; e.g. see [12, §6]. Usually, variant forms of these rules are applied in actual proof, see also §12.4 and §12.5.

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 §12.4.

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 $\vdash P\ (While\ b\ X\ c)\ (P \cap -b)$
proof
fix $s\ s'$ **assume** *s*: $s \in P$
assume $Sem\ (While\ b\ X\ c)\ s\ s'$
then obtain n **where** $iter\ n\ b\ (Sem\ c)\ s\ s'$ **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*: *Sem c s* s''
and *iter*: *iter n b* (*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

12.3 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* $\Rightarrow 'b \Rightarrow$ *idt* $\Rightarrow 'a$ *bexp* (-[-'/´-] [*1000*] *999*)
-Assert :: $'a \Rightarrow 'a$ *set* ((⦃-⦄) [*0*] *1000*)
-Assign :: *idt* $\Rightarrow 'b \Rightarrow 'a$ *com* ((´- :=/ -) [*70*, *65*] *61*)
-Cond :: $'a$ *bexp* $\Rightarrow 'a$ *com* $\Rightarrow 'a$ *com* $\Rightarrow 'a$ *com*
((*0IF* -/ *THEN* -/ *ELSE* -/ *FI*) [*0*, *0*, *0*] *61*)
-While-inv :: $'a$ *bexp* $\Rightarrow 'a$ *assn* $\Rightarrow 'a$ *com* $\Rightarrow 'a$ *com*
((*0WHILE* -/ *INV* - //*DO* - /*OD*) [*0*, *0*, *0*] *61*)
-While :: $'a$ *bexp* $\Rightarrow 'a$ *com* $\Rightarrow 'a$ *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 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›)),
    (const-syntax ‹While›, K (bexp-tr' syntax-const ‹-While-inv›))]
  end
›
```

12.4 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*]:
  $\vdash \{\!|\acute{}P|\!\}\ c\ Q \Longrightarrow (\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \vdash \{\!|\acute{}P'|\!\}\ c\ Q$
  by (*simp add*: *Valid-def*)
lemma [*trans*]:
  $(\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \vdash \{\!|\acute{}P|\!\}\ c\ Q \Longrightarrow \vdash \{\!|\acute{}P'|\!\}\ c\ Q$
  by (*simp add*: *Valid-def*)

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

Identity and basic assignments.[8]

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

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

Note that above formulation of assignment corresponds to our preferred way to model state spaces, using (extensible) record types in HOL [5]. 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*.[9]

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

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

[8] The *hoare* method introduced in §12.5 is able to provide proper instances for any number of basic assignments, without producing additional verification conditions.

[9] 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 *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?*]:
$\vdash \{\!|{}^{\prime}P \wedge {}^{\prime}b|\!\}\ c1\ Q$
$\Longrightarrow \vdash \{\!|{}^{\prime}P \wedge \neg\ {}^{\prime}b|\!\}\ c2\ Q$
$\Longrightarrow \vdash \{\!|{}^{\prime}P|\!\}\ IF\ {}^{\prime}b\ THEN\ c1\ ELSE\ c2\ FI\ Q$
by (*rule cond*) (*simp-all add*: *Valid-def*)

While statements — with optional invariant.

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

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

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

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

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

lemma *Compl-Collect*: $-$ *Collect b* = $\{x.\ \neg\ b\ x\}$
  by *blast*

lemmas *AbortRule* = *SkipRule* — dummy version

ML-file ‹*~~/src/HOL/Hoare/hoare-tac.ML*›

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

end

13 Using Hoare Logic

theory *Hoare-Ex*
  imports *Hoare*
begin

13.1 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.

13.2 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 $\vdash \{\!|True|\!\}\ \acute{}M := a;\ \acute{}N := b\ \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$

by *hoare*

lemma
$\vdash \{\!|\,{}^\acute{}M = a \wedge {}^\acute{}N = b|\!\}$
$\quad {}^\acute{}I := {}^\acute{}M;\ {}^\acute{}M := {}^\acute{}N;\ {}^\acute{}N := {}^\acute{}I$
$\quad \{\!|\,{}^\acute{}M = b \wedge {}^\acute{}N = a|\!\}$
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
$\mathit{Valid}\ \{s.\ x\ s = a\}\ (\mathit{Basic}\ (\lambda s.\ \mathit{x\text{-}update}\ (x\ s)\ s))\ \{s.\ x\ s = n\}$
— 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 $\vdash \{\!|\,{}^\acute{}M = {}^\acute{}N|\!\}\ {}^\acute{}M := {}^\acute{}M + \mathit{1}\ \{\!|\,{}^\acute{}M \neq {}^\acute{}N|\!\}$
proof −
have $\{\!|\,{}^\acute{}M = {}^\acute{}N|\!\} \subseteq \{\!|\,{}^\acute{}M + \mathit{1} \neq {}^\acute{}N|\!\}$
by *auto*
also have $\vdash \ldots\ {}^\acute{}M := {}^\acute{}M + \mathit{1}\ \{\!|\,{}^\acute{}M \neq {}^\acute{}N|\!\}$
by *hoare*
finally show *?thesis* **.**
qed

lemma $\vdash \{\!|\,{}^\acute{}M = {}^\acute{}N|\!\}\ {}^\acute{}M := {}^\acute{}M + \mathit{1}\ \{\!|\,{}^\acute{}M \neq {}^\acute{}N|\!\}$
proof −
have $m = n \longrightarrow m + \mathit{1} \neq n$ **for** $m\ n :: \mathit{nat}$
— inclusion of assertions expressed in “pure” logic,
— without mentioning the state space
by *simp*
also have $\vdash \{\!|\,{}^\acute{}M + \mathit{1} \neq {}^\acute{}N|\!\}\ {}^\acute{}M := {}^\acute{}M + \mathit{1}\ \{\!|\,{}^\acute{}M \neq {}^\acute{}N|\!\}$
by *hoare*
finally show *?thesis* **.**
qed

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

13.3 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
  $\vdash \{\!|\,{}^\prime M = 0 \wedge {}^\prime S = 0|\!\}$
    $WHILE\ {}^\prime M \neq a$
    $DO\ {}^\prime S := {}^\prime S + b;\ {}^\prime M := {}^\prime M + 1\ OD$
    $\{\!|\,{}^\prime S = a * b|\!\}$
proof $-$
  let $\vdash$ - *?while* - = *?thesis*
  let $\{\!|\,{}^\prime ?inv|\!\} = \{\!|\,{}^\prime S = {}^\prime M * b|\!\}$

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

13.4 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*. $\sum j<k.\ j$
let *?inv* = $\lambda s\ i$::*nat*. s = *?sum* i
show *?thesis*
proof *hoare*
  show *?inv 0 1* **by** *simp*
  show *?inv* $(s + i)$ $(i + 1)$ **if** *?inv* $s\ i \wedge i \neq n$ **for** $s\ i$
    using *that* **by** *simp*
  show s = *?sum* n **if** *?inv* $s\ i \wedge \neg\ i \neq n$ **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
$\vdash$ {|*True*|}
  ´*S* := *0*; ´*I* := *1*;
  WHILE ´*I* $\neq$ *n*
  INV {|´*S* = ($\sum j<$´*I*. *j*)|}
  DO
    ´*S* := ´*S* + ´*I*;
    ´*I* := ´*I* + *1*
  OD
  {|´*S* = ($\sum j<n.\ j$)|}
by *hoare auto*

13.5 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* $\Rightarrow$ *'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 c*) = *While b iv* (*timeit c*)

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

lemma *lem*: (*0*::*nat*) $< n \Longrightarrow n + n \leq$ *Suc* $(n * n)$
by (*induct n*) *simp-all*

lemma
$\vdash \{\!|i = {}^\prime I \wedge {}^\prime time = 0|\!\}$
(*timeit*
(*WHILE* $'I \neq 0$
INV $\{\!|2 *' time + 'I * 'I + 5 * 'I = i * i + 5 * i|\!\}$
DO
$'J := 'I$;
WHILE $'J \neq 0$
INV $\{\!|0 < 'I \wedge 2 * 'time + 'I * 'I + 3 * 'I + 2 * 'J - 2 = i * i + 5 * i|\!\}$
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

14 The Mutilated Checker Board Problem

theory *Mutilated-Checkerboard*
imports *Main*
begin

The Mutilated Checker Board Problem, formalized inductively. See [9] for the original tactic script version.

14.1 Tilings

inductive-set *tiling* :: *'a set set* $\Rightarrow$ *'a set set* **for** *A* :: *'a set set*
where
empty: $\{\} \in$ *tiling A*
| *Un*: $a \cup t \in$ *tiling A* **if** $a \in A$ **and** $t \in$ *tiling A* **and** $a \subseteq - t$

The union of two disjoint tilings is a tiling.

lemma *tiling-Un*:
assumes $t \in$ *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
```

14.2 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 ⟹ below m × B = ({n} × B) ∪ (below n ×
B)
  by (simp add: below-def less-Suc-eq) blast

lemma Sigma-Suc2:
  m = n + 2 ⟹
    A × below m = (A × {n}) ∪ (A × {n + 1}) ∪ (A × below n)
  by (auto simp add: below-def)
```

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

14.3 Basic properties of "evnodd"

definition *evnodd* :: (*nat* × *nat*) *set* ⇒ *nat* ⇒ (*nat* × *nat*) *set*
  where *evnodd A b* = *A* ∩ {(*i*, *j*). (*i* + *j*) *mod 2* = *b*}

lemma *evnodd-iff*: (*i*, *j*) ∈ *evnodd A b* ⟷ (*i*, *j*) ∈ *A* ∧ (*i* + *j*) *mod 2* = *b*
  by (*simp add*: *evnodd-def*)

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

lemma *evnoddD*: *x* ∈ *evnodd A b* ⟹ *x* ∈ *A*
  by (*rule subsetD*) (*rule evnodd-subset*)

lemma *evnodd-finite*: *finite A* ⟹ *finite* (*evnodd A b*)
  by (*rule finite-subset*) (*rule evnodd-subset*)

lemma *evnodd-Un*: *evnodd* (*A* ∪ *B*) *b* = *evnodd A b* ∪ *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*)

14.4 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 (*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
```

14.5 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
```

14.6 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) ∈ ?e ?t' 0 by simp
      qed
      then show ?thesis by simp
    qed
    also have ... < card (?e ?t 0)
    proof −
      have (0, 0) ∈ ?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** ... = *card* (*?e ?t 1*)
by (*rule tiling-domino-01*)
also have *?e ?t 1* = *?e ?t'' 1* **by** *simp*
also from *t''* **have** *card* ... = *card* (*?e ?t'' 0*)
by (*rule tiling-domino-01* [*symmetric*])
finally have ... < ... **. then show** *False* **..**
qed
qed

end

15 An old chestnut

theory *Puzzle*
imports *Main*
begin[10]

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** $m \le f\ m$ **.**
also note ‹$\ldots < f\ n$›
finally have $m < f\ n$ **.**
then have $n \le f\ n$ **by** (*simp only*: *Suc*)
then show *?thesis* **.**
next
case *0*
then show *?thesis* **by** *simp*
qed

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

```
  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
```

16 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.

16.1 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 = $ -)
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 "..." 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*:
$(\sum i{::}nat{=}0..{<}n.\ 2 * i + 1) = n\hat{}Suc\ (Suc\ 0)$
(**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 $?S\ n = n\hat{}Suc\ (Suc\ 0)$
also have $\ldots + 2 * n + 1 = (n + 1)\hat{}Suc\ (Suc\ 0)$
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*:
$6 * (\sum i{::}nat{=}0..n.\ i\hat{}Suc\ (Suc\ 0)) = n * (n + 1) * (2 * n + 1)$
(**is** $?P\ n$ **is** $?S\ n = -$)
proof (*induct n*)
show $?P\ 0$ **by** *simp*
next
fix n
have $?S\ (n + 1) = ?S\ n + 6 * (n + 1)\hat{}Suc\ (Suc\ 0)$
by (*simp add*: *distrib*)
also assume $?S\ n = n * (n + 1) * (2 * n + 1)$
also have $\ldots + 6 * (n + 1)\hat{}Suc\ (Suc\ 0) =$
$(n + 1) * (n + 2) * (2 * (n + 1) + 1)$
by (*simp add*: *distrib*)
finally show $?P\ (Suc\ n)$
by *simp*
qed

theorem *sum-of-cubes*:
$4 * (\sum i{::}nat{=}0..n.\ i\hat{}3) = (n * (n + 1))\hat{}Suc\ (Suc\ 0)$
(**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

17 A simple formulation of First-Order Logic

The subsequent theory development illustrates single-sorted intuitionistic first-order logic with equality, formulated within the Pure framework.

theory *First-Order-Logic*
  imports *Pure*
begin

17.1 Abstract syntax

typedecl *i*
typedecl *o*

judgment *Trueprop* :: $o \Rightarrow prop$ (- *5*)

17.2 Propositional logic

axiomatization *false* :: *o* (⊥)
  where *falseE* [*elim*]: $\bot \Longrightarrow A$

axiomatization *imp* :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\longrightarrow$ *25*)
  where *impI* [*intro*]: $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$
    and *mp* [*dest*]: $A \longrightarrow B \Longrightarrow A \Longrightarrow B$

axiomatization *conj* :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\wedge$ *35*)
  where *conjI* [*intro*]: $A \Longrightarrow B \Longrightarrow A \wedge B$
    and *conjD1*: $A \wedge B \Longrightarrow A$
    and *conjD2*: $A \wedge B \Longrightarrow B$

theorem *conjE* [*elim*]:

assumes $A \wedge B$
obtains A **and** B
proof
from $\langle A \wedge B\rangle$ **show** A
by (*rule conjD1*)
from $\langle A \wedge B\rangle$ **show** B
by (*rule conjD2*)
qed

axiomatization *disj* :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\vee$ *30*)
where *disjE* [*elim*]: $A \vee B \Longrightarrow (A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C$
and *disjI1* [*intro*]: $A \Longrightarrow A \vee B$
and *disjI2* [*intro*]: $B \Longrightarrow A \vee B$

definition *true* :: o ($\top$)
where $\top \equiv \bot \longrightarrow \bot$

theorem *trueI* [*intro*]: $\top$
unfolding *true-def* **..**

definition *not* :: $o \Rightarrow o$ ($\neg$ - [*40*] *40*)
where $\neg A \equiv A \longrightarrow \bot$

theorem *notI* [*intro*]: $(A \Longrightarrow \bot) \Longrightarrow \neg A$
unfolding *not-def* **..**

theorem *notE* [*elim*]: $\neg A \Longrightarrow A \Longrightarrow B$
unfolding *not-def*
proof $-$
assume $A \longrightarrow \bot$ **and** A
then have $\bot$ **..**
then show B **..**
qed

definition *iff* :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\longleftrightarrow$ *25*)
where $A \longleftrightarrow B \equiv (A \longrightarrow B) \wedge (B \longrightarrow A)$

theorem *iffI* [*intro*]:
assumes $A \Longrightarrow B$
and $B \Longrightarrow A$
shows $A \longleftrightarrow B$
unfolding *iff-def*
proof
from $\langle A \Longrightarrow B\rangle$ **show** $A \longrightarrow B$ **..**
from $\langle B \Longrightarrow A\rangle$ **show** $B \longrightarrow A$ **..**

qed

theorem *iff1* [*elim*]:
  assumes $A \longleftrightarrow B$ **and** A
  shows B
proof −
  from $\langle A \longleftrightarrow B\rangle$ **have** $(A \longrightarrow B) \wedge (B \longrightarrow A)$
    unfolding *iff-def* **.**
  then have $A \longrightarrow B$ **..**
  from *this* **and** $\langle A\rangle$ **show** B **..**
qed

theorem *iff2* [*elim*]:
  assumes $A \longleftrightarrow B$ **and** B
  shows A
proof −
  from $\langle A \longleftrightarrow B\rangle$ **have** $(A \longrightarrow B) \wedge (B \longrightarrow A)$
    unfolding *iff-def* **.**
  then have $B \longrightarrow A$ **..**
  from *this* **and** $\langle B\rangle$ **show** A **..**
qed

17.3 Equality

axiomatization *equal* :: $i \Rightarrow i \Rightarrow o$ (**infixl** = *50*)
  where *refl* [*intro*]: $x = x$
    and *subst*: $x = y \Longrightarrow P\ x \Longrightarrow P\ y$

theorem *trans* [*trans*]: $x = y \Longrightarrow y = z \Longrightarrow x = z$
  by (*rule subst*)

theorem *sym* [*sym*]: $x = y \Longrightarrow y = x$
proof −
  assume $x = y$
  from *this* **and** *refl* **show** $y = x$
    by (*rule subst*)
qed

17.4 Quantifiers

axiomatization *All* :: $(i \Rightarrow o) \Rightarrow o$ (**binder** ∀ *10*)
  where *allI* [*intro*]: $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
    and *allD* [*dest*]: $\forall x.\ P\ x \Longrightarrow P\ a$

axiomatization *Ex* :: $(i \Rightarrow o) \Rightarrow o$ (**binder** ∃ *10*)
  where *exI* [*intro*]: $P\ a \Longrightarrow \exists x.\ P\ x$
    and *exE* [*elim*]: $\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$

lemma $(\exists x.\ P\ (f\ x)) \longrightarrow (\exists y.\ P\ y)$

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

lemma $(\exists x.\ \forall y.\ R\ x\ y) \longrightarrow (\forall y.\ \exists x.\ R\ x\ y)$
proof
  assume $\exists x.\ \forall y.\ R\ x\ y$
  then obtain x **where** $\forall y.\ R\ x\ y$ **..**
  show $\forall y.\ \exists x.\ R\ x\ y$
  proof
    fix y
    from $\langle\forall y.\ R\ x\ y\rangle$ **have** $R\ x\ y$ **..**
    then show $\exists x.\ R\ x\ y$ **..**
  qed
qed

end

18 Foundations of HOL

theory *Higher-Order-Logic*
  imports *Pure*
begin

The following theory development illustrates the foundations of Higher-Order Logic. The "HOL" logic that is given here resembles [3] and its predecessor [1], but the order of axiomatizations and defined connectives has be adapted to modern presentations of λ-calculus and Constructive Type Theory. Thus it fits nicely to the underlying Natural Deduction framework of Isabelle/Pure and Isabelle/Isar.

19 HOL syntax within Pure

class *type*
default-sort *type*

typedecl *o*
instance *o* :: *type* **..**
instance *fun* :: (*type*, *type*) *type* **..**

judgment *Trueprop* :: $o \Rightarrow prop$ (- *5*)

20 Minimal logic (axiomatization)

axiomatization *imp* :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\longrightarrow$ *25*)

where *impI* [*intro*]: $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$
and *impE* [*dest*, *trans*]: $A \longrightarrow B \Longrightarrow A \Longrightarrow B$

axiomatization *All* :: $('a \Rightarrow o) \Rightarrow o$ (**binder** $\forall$ *10*)
where *allI* [*intro*]: $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
and *allE* [*dest*]: $\forall x.\ P\ x \Longrightarrow P\ a$

lemma *atomize-imp* [*atomize*]: $(A \Longrightarrow B) \equiv$ *Trueprop* $(A \longrightarrow B)$
by *standard* (*fact impI*, *fact impE*)

lemma *atomize-all* [*atomize*]: $(\bigwedge x.\ P\ x) \equiv$ *Trueprop* $(\forall x.\ P\ x)$
by *standard* (*fact allI*, *fact allE*)

20.0.1 Derived connectives

definition *False* :: *o*
where *False* $\equiv \forall A.\ A$

lemma *FalseE* [*elim*]:
assumes *False*
shows A
proof –
from ‹*False*› **have** $\forall A.\ A$ **by** (*simp only*: *False-def*)
then show A **..**
qed

definition *True* :: *o*
where *True* $\equiv$ *False* $\longrightarrow$ *False*

lemma *TrueI* [*intro*]: *True*
unfolding *True-def* **..**

definition *not* :: $o \Rightarrow o$ ($\neg$ - [*40*] *40*)
where *not* $\equiv \lambda A.\ A \longrightarrow$ *False*

lemma *notI* [*intro*]:
assumes $A \Longrightarrow$ *False*
shows $\neg\ A$
using *assms* **unfolding** *not-def* **..**

lemma *notE* [*elim*]:
assumes $\neg\ A$ **and** A
shows B
proof –
from ‹$\neg\ A$› **have** $A \longrightarrow$ *False* **by** (*simp only*: *not-def*)
from *this* **and** ‹A› **have** *False* **..**
then show B **..**

qed

lemma *notE'*: $A \Longrightarrow \neg\ A \Longrightarrow B$
  by (*rule notE*)

lemmas *contradiction* = *notE notE'* — proof by contradiction in any order

definition *conj* :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\wedge$ *35*)
  where $A \wedge B \equiv \forall\, C.\ (A \longrightarrow B \longrightarrow C) \longrightarrow C$

lemma *conjI* [*intro*]:
  assumes A **and** B
  shows $A \wedge B$
  unfolding *conj-def*
proof
  fix C
  show $(A \longrightarrow B \longrightarrow C) \longrightarrow C$
  proof
    assume $A \longrightarrow B \longrightarrow C$
    also note ‹A›
    also note ‹B›
    finally show C **.**
  qed
qed

lemma *conjE* [*elim*]:
  assumes $A \wedge B$
  obtains A **and** B
proof
  from ‹$A \wedge B$› **have** $*$: $(A \longrightarrow B \longrightarrow C) \longrightarrow C$ **for** C
    unfolding *conj-def* **..**
  show A
  proof $-$
    note $*$ [*of A*]
    also have $A \longrightarrow B \longrightarrow A$
    proof
      assume A
      then show $B \longrightarrow A$ **..**
    qed
    finally show *?thesis* **.**
  qed
  show B
  proof $-$
    note $*$ [*of B*]
    also have $A \longrightarrow B \longrightarrow B$
    proof
      show $B \longrightarrow B$ **..**
    qed

```
    finally show ?thesis .
  qed
qed


definition disj :: o ⇒ o ⇒ o  (infixr ∨ 30)
  where A ∨ B ≡ ∀C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C

lemma disjI1 [intro]:
  assumes A
  shows A ∨ B
  unfolding disj-def
proof
  fix C
  show (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
  proof
    assume A ⟶ C
    from this and ‹A› have C ..
    then show (B ⟶ C) ⟶ C ..
  qed
qed

lemma disjI2 [intro]:
  assumes B
  shows A ∨ B
  unfolding disj-def
proof
  fix C
  show (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
  proof
    show (B ⟶ C) ⟶ C
    proof
      assume B ⟶ C
      from this and ‹B› show C ..
    qed
  qed
qed

lemma disjE [elim]:
  assumes A ∨ B
  obtains (a) A | (b) B
proof –
  from ‹A ∨ B› have (A ⟶ thesis) ⟶ (B ⟶ thesis) ⟶ thesis
    unfolding disj-def ..
  also have A ⟶ thesis
  proof
    assume A
    then show thesis by (rule a)
  qed
```

```
  also have B ⟶ thesis
  proof
    assume B
    then show thesis by (rule b)
  qed
  finally show thesis .
qed


definition Ex :: ('a ⇒ o) ⇒ o  (binder ∃ 10)
  where ∃x. P x ≡ ∀C. (∀x. P x ⟶ C) ⟶ C

lemma exI [intro]: P a ⟹ ∃x. P x
  unfolding Ex-def
proof
  fix C
  assume P a
  show (∀x. P x ⟶ C) ⟶ C
  proof
    assume ∀x. P x ⟶ C
    then have P a ⟶ C ..
    from this and ‹P a› show C ..
  qed
qed

lemma exE [elim]:
  assumes ∃x. P x
  obtains (that) x where P x
proof −
  from ‹∃x. P x› have (∀x. P x ⟶ thesis) ⟶ thesis
    unfolding Ex-def ..
  also have ∀x. P x ⟶ thesis
  proof
    fix x
    show P x ⟶ thesis
    proof
      assume P x
      then show thesis by (rule that)
    qed
  qed
  finally show thesis .
qed
```

20.0.2 Extensional equality

```
axiomatization equal :: 'a ⇒ 'a ⇒ o  (infixl = 50)
  where refl [intro]: x = x
    and subst: x = y ⟹ P x ⟹ P y
```

abbreviation *not-equal* :: $'a \Rightarrow 'a \Rightarrow o$ (**infixl** $\neq$ *50*)
where $x \neq y \equiv \neg (x = y)$

abbreviation *iff* :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\longleftrightarrow$ *25*)
where $A \longleftrightarrow B \equiv A = B$

axiomatization
where *ext* [*intro*]: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow f = g$
and *iff* [*intro*]: $(A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A \longleftrightarrow B$

lemma *sym* [*sym*]: $y = x$ **if** $x = y$
using *that* **by** (*rule subst*) (*rule refl*)

lemma [*trans*]: $x = y \Longrightarrow P\ y \Longrightarrow P\ x$
by (*rule subst*) (*rule sym*)

lemma [*trans*]: $P\ x \Longrightarrow x = y \Longrightarrow P\ y$
by (*rule subst*)

lemma *arg-cong*: $f\ x = f\ y$ **if** $x = y$
using *that* **by** (*rule subst*) (*rule refl*)

lemma *fun-cong*: $f\ x = g\ x$ **if** $f = g$
using *that* **by** (*rule subst*) (*rule refl*)

lemma *trans* [*trans*]: $x = y \Longrightarrow y = z \Longrightarrow x = z$
by (*rule subst*)

lemma *iff1* [*elim*]: $A \longleftrightarrow B \Longrightarrow A \Longrightarrow B$
by (*rule subst*)

lemma *iff2* [*elim*]: $A \longleftrightarrow B \Longrightarrow B \Longrightarrow A$
by (*rule subst*) (*rule sym*)

20.1 Cantor's Theorem

Cantor's Theorem states that there is no surjection from a set to its powerset. The subsequent formulation uses elementary λ-calculus and predicate logic, with standard introduction and elimination rules.

lemma *iff-contradiction*:
assumes $*$: $\neg A \longleftrightarrow A$
shows C
proof (*rule notE*)
show $\neg A$
proof
assume A
with $*$ **have** $\neg A$ **..**
from *this* **and** $\langle A \rangle$ **show** *False* **..**
qed

with $*$ **show** A **..**
qed

theorem *Cantor*: $\neg\ (\exists f :: 'a \Rightarrow 'a \Rightarrow o.\ \forall A.\ \exists x.\ A = f\ x)$
proof
assume $\exists f :: 'a \Rightarrow 'a \Rightarrow o.\ \forall A.\ \exists x.\ A = f\ x$
then obtain $f :: 'a \Rightarrow 'a \Rightarrow o$ **where** $*$: $\forall A.\ \exists x.\ A = f\ x$ **..**
let $?D = \lambda x.\ \neg\ f\ x\ x$
from $*$ **have** $\exists x.\ ?D = f\ x$ **..**
then obtain a **where** $?D = f\ a$ **..**
then have $?D\ a \longleftrightarrow f\ a\ a$ **using** *refl* **by** (*rule subst*)
then have $\neg\ f\ a\ a \longleftrightarrow f\ a\ a$ **.**
then show *False* **by** (*rule iff-contradiction*)
qed

20.2 Characterization of Classical Logic

The subsequent rules of classical reasoning are all equivalent.

locale *classical* $=$
assumes *classical*: $(\neg\ A \Longrightarrow A) \Longrightarrow A$
— predicate definition and hypothetical context
begin

lemma *classical-contradiction*:
assumes $\neg\ A \Longrightarrow$ *False*
shows A
proof (*rule classical*)
assume $\neg\ A$
then have *False* **by** (*rule assms*)
then show A **..**
qed

lemma *double-negation*:
assumes $\neg\ \neg\ A$
shows A
proof (*rule classical-contradiction*)
assume $\neg\ A$
with $\langle\neg\ \neg\ A\rangle$ **show** *False* **by** (*rule contradiction*)
qed

lemma *tertium-non-datur*: $A \lor \neg\ A$
proof (*rule double-negation*)
show $\neg\ \neg\ (A \lor \neg\ A)$
proof
assume $\neg\ (A \lor \neg\ A)$
have $\neg\ A$
proof
assume A **then have** $A \lor \neg\ A$ **..**
with $\langle\neg\ (A \lor \neg\ A)\rangle$ **show** *False* **by** (*rule contradiction*)

```
    qed
    then have A ∨ ¬ A ..
    with ‹¬ (A ∨ ¬ A)› show False by (rule contradiction)
  qed
qed

lemma classical-cases:
  obtains A | ¬ A
  using tertium-non-datur
proof
  assume A
  then show thesis ..
next
  assume ¬ A
  then show thesis ..
qed

end

lemma classical-if-cases: classical
  if cases: ⋀A C. (A ⟹ C) ⟹ (¬ A ⟹ C) ⟹ C
proof
  fix A
  assume *: ¬ A ⟹ A
  show A
  proof (rule cases)
    assume A
    then show A .
  next
    assume ¬ A
    then show A by (rule *)
  qed
qed
```

21 Peirce's Law

Peirce's Law is another characterization of classical reasoning. Its statement only requires implication.

```
theorem (in classical) Peirce's-Law: ((A ⟶ B) ⟶ A) ⟶ A
proof
  assume *: (A ⟶ B) ⟶ A
  show A
  proof (rule classical)
    assume ¬ A
    have A ⟶ B
    proof
      assume A
      with ‹¬ A› show B by (rule contradiction)
```

```
    qed
    with * show A ..
  qed
qed
```

22 Hilbert's choice operator (axiomatization)

```
axiomatization Eps :: ('a ⇒ o) ⇒ 'a
  where someI: P x ⟹ P (Eps P)

syntax -Eps :: pttrn ⇒ o ⇒ 'a  ((3SOME -./ -) [0, 10] 10)
translations SOME x. P ⇌ CONST Eps (λx. P)
```

It follows a derivation of the classical law of tertium-non-datur by means of Hilbert's choice operator (due to Berghofer, Beeson, Harrison, based on a proof by Diaconescu).

```
theorem Diaconescu: A ∨ ¬ A
proof −
  let ?P = λx. (A ∧ x) ∨ ¬ x
  let ?Q = λx. (A ∧ ¬ x) ∨ x

  have a: ?P (Eps ?P)
  proof (rule someI)
    have ¬ False ..
    then show ?P False ..
  qed
  have b: ?Q (Eps ?Q)
  proof (rule someI)
    have True ..
    then show ?Q True ..
  qed

  from a show ?thesis
  proof
    assume A ∧ Eps ?P
    then have A ..
    then show ?thesis ..
  next
    assume ¬ Eps ?P
    from b show ?thesis
    proof
      assume A ∧ ¬ Eps ?Q
      then have A ..
      then show ?thesis ..
    next
      assume Eps ?Q
      have neq: ?P ≠ ?Q
```

```
      proof
        assume ?P = ?Q
        then have Eps ?P ⟷ Eps ?Q by (rule arg-cong)
        also note ‹Eps ?Q›
        finally have Eps ?P .
        with ‹¬ Eps ?P› show False by (rule contradiction)
      qed
      have ¬ A
      proof
        assume A
        have ?P = ?Q
        proof (rule ext)
          show ?P x ⟷ ?Q x for x
          proof
            assume ?P x
            then show ?Q x
            proof
              assume ¬ x
              with ‹A› have A ∧ ¬ x ..
              then show ?thesis ..
            next
              assume A ∧ x
              then have x ..
              then show ?thesis ..
            qed
          next
            assume ?Q x
            then show ?P x
            proof
              assume A ∧ ¬ x
              then have ¬ x ..
              then show ?thesis ..
            next
              assume x
              with ‹A› have A ∧ x ..
              then show ?thesis ..
            qed
          qed
        qed
        with neq show False by (rule contradiction)
      qed
      then show ?thesis ..
    qed
  qed
qed
```

This means, the hypothetical predicate *classical* always holds unconditionally (with all consequences).

interpretation *classical*

proof (*rule classical-if-cases*)
  fix *A C*
  assume $*$: $A \Longrightarrow C$
    and $**$: $\neg\ A \Longrightarrow C$
  from *Diaconescu* [*of A*] **show** *C*
  proof
    assume *A*
    then show *C* **by** (*rule* $*$)
  next
    assume $\neg\ A$
    then show *C* **by** (*rule* $**$)
  qed
qed

thm *classical*
  classical-contradiction
  double-negation
  tertium-non-datur
  classical-cases
  Peirce's-Law

end

References

[1] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[3] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

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

[5] 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.

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

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

[8] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[9] 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.

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

[11] 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.

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