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  
1.1 Prose version ........................................ 3  
1.2 Formal versions ...................................... 4  

2 Peirce’s Law  

3 The Drinker’s Principle  

4 Cantor’s Theorem  
4.1 Mathematical statement and proof ...................... 8  
4.2 Automated proofs .................................... 8  
4.3 Elementary version in higher-order predicate logic .... 8  
4.4 Classic Isabelle/HOL example ........................ 9  

5 Structured statements within Isar proofs  
5.1 Introduction steps .................................. 10  
5.2 If-and-only-if ....................................... 11  
5.3 Elimination and cases ............................... 12  
5.4 Induction .......................................... 12  
5.5 Suffices-to-show .................................... 13
6 Basic logical reasoning
   6.1 Pure backward reasoning .................................. 13
   6.2 Variations of backward vs. forward reasoning ........ 15
   6.3 A few examples from “Introduction to Isabelle” ...... 18
       6.3.1 A propositional proof ............................... 18
       6.3.2 A quantifier proof ................................. 19
       6.3.3 Deriving rules in Isabelle ......................... 20

7 Correctness of a simple expression compiler ....... 20
   7.1 Binary operations ...................................... 21
   7.2 Expressions .............................................. 21
   7.3 Machine .................................................. 21
   7.4 Compiler .................................................. 22

8 Fib and Gcd commute .................................... 24
   8.1 Fibonacci numbers ...................................... 25
   8.2 Fib and gcd commute ................................. 25

9 Basic group theory ..................................... 28
   9.1 Groups and calculational reasoning ................... 28
   9.2 Groups as monoids ...................................... 30
   9.3 More theorems of group theory ......................... 30

10 Some algebraic identities derived from group axioms – theory context version 32

11 Some algebraic identities derived from group axioms – proof notepad version 34

12 Hoare Logic .............................................. 36
   12.1 Abstract syntax and semantics ....................... 36
   12.2 Primitive Hoare rules .................................. 37
   12.3 Concrete syntax for assertions ...................... 39
   12.4 Rules for single-step proof ......................... 40
   12.5 Verification conditions .............................. 42

13 Using Hoare Logic .................................... 43
   13.1 State spaces ........................................... 44
   13.2 Basic examples ....................................... 44
   13.3 Multiplication by addition ........................... 46
   13.4 Summing natural numbers ............................. 46
   13.5 Time ................................................... 48
## 14 The Mutilated Checker Board Problem

14.1 Tilings .............................................. 49
14.2 Basic properties of “below” ......................... 50
14.3 Basic properties of “evnodd” ......................... 51
14.4 Dominoes ........................................... 51
14.5 Tilings of dominoes ................................. 53
14.6 Main theorem ....................................... 54

## 15 An old chestnut

## 16 Summing natural numbers

16.1 Summation laws ................................. 57

## 17 A simple formulation of First-Order Logic

17.1 Abstract syntax ................................. 59
17.2 Propositional logic .............................. 59
17.3 Equality ........................................ 61
17.4 Quantifiers .................................... 61

## 18 Foundations of HOL

## 19 HOL syntax within Pure

## 20 Minimal logic (axiomatization)

20.0.1 Derived connectives .......................... 63
20.0.2 Extensional equality ......................... 66
20.1 Cantor’s Theorem .............................. 67
20.2 Characterization of Classical Logic ........... 68

## 21 Peirce’s Law

## 22 Hilbert’s choice operator (axiomatization)

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 $f: L \to L$ an order-preserving map. Then $\bigcap \{x \in L \mid f(x) \leq x\}$ is a fixpoint of $f$.

**Proof.** Let $H = \{x \in L \mid f(x) \leq x\}$ and $a = \bigcap H$. 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* $f :: \text{complete-lattice} \Rightarrow \text{complete-lattice}$
  *assumes* $\text{mono } f$
  *shows* $\exists a. f\ a = a$

**proof**
  `let ?H = \{u. f\ u \leq u\}`
  `let ?a = d ?H`
  `show f ?a = ?a`
  `proof`
  `{ `fix x`
   `assume x \in ?H`
   `then have ?a \leq x` (rule Inf-lower)
   `with \text{mono } f` have $f\ ?a \leq f\ x$ ..
   `also from \langle x \in ?H, \text{have ... } \leq x \rangle`
   `finally have f\ ?a \leq x`
  `}`
  then have $f\ ?a \leq ?a$ (rule Inf-greatest)
  `{ `also presume \ldots \leq f\ ?a`
   `finally (\text{order-antisym}) show ?thesis`
  `}`
  from \langle mono f \rangle and \langle f\ ?a \leq ?a \rangle have $f\ (f\ ?a) \leq f\ ?a$ ..
  then have $f\ ?a \in ?H$ ..
  then show $?a \leq f\ ?a$ 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* \( f : \mathcal{A} \rightarrow \mathcal{A} :: \text{complete-lattice} \Rightarrow \mathcal{A} \)
- *assumes* \( \text{mono } f \)
- *shows* \( \exists a. f a = a \)

**proof**

- let \( ?H = \{ a. f u \leq u \} \)
- let \( ?a = \bigcap ?H \)
- show \( f ?a = ?a \)

**proof** (rule order-antisym)

- show \( f ?a \leq ?a \)

**proof** (rule Inf-greatest)

- fix \( x \)
- assume \( x \in ?H \)
- then have \( ?a \leq x \) by (rule Inf-lower)
- with (mono \( f \)) have \( f ?a \leq f x \)
- also from \( x \in ?H \) have \( \ldots \leq x \)
- finally show \( f ?a \leq x \).

qed

- show \( ?a \leq f ?a \)

**proof** (rule Inf-lower)

- from (mono \( f \)) and (f ?a \leq ?a) have \( f (f ?a) \leq f ?a \)
- then show \( f ?a \in ?H \)

qed

end

2 Peirce’s Law

**theory** Peirce

- *imports* Main

**begin**

We consider Peirce’s Law: \((A \rightarrow B) \rightarrow A\) \rightarrow 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** 
\[(A \rightarrow B) \rightarrow A\]

**Proof**

1. Assume \((A \rightarrow B) \rightarrow A\)
2. Show \(A\)
3. Prove \((\text{rule classical})\)
4. Assume \(\neg A\)
5. Have \(A \rightarrow B\)
6. Prove
   1. Assume \(A\)
   2. With \((\neg A)\) show \(B\) by contradiction
   3. QED
7. With \((A \rightarrow B) \rightarrow A\) show \(A\).
8. QED
9. 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 \(A \rightarrow B\) is put into place in order to solve the main problem. Nevertheless, we do not get anything for free, but have to establish \(A \rightarrow B\) 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** 
\[(A \rightarrow B) \rightarrow A\]

**Proof**

1. Assume \((A \rightarrow B) \rightarrow A\)
2. Show \(A\)
3. Prove \((\text{rule classical})\)
4. Presume \(A \rightarrow B\)
5. With \((A \rightarrow B) \rightarrow A\) show \(A\).
6. Next
   1. Assume \(\neg A\)
   2. Show \(A \rightarrow B\)
   3. Prove
      1. Assume \(A\)
      2. With \((\neg A)\) show \(B\) by contradiction
      3. QED
   4. QED
   5. 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 their 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.

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 ¬ (∀ x. P x)
  shows ∃ x. ¬ P x
proof (rule classical)
  assume ¬ (∀ x. P x)
  have ∀ x. P x
  proof
    fix x show P x
    proof (rule classical)
      assume ¬ P x
      then have ∃ x. ¬ P x ..
      with (¬ (∀ x. P x)) show ?thesis by contradiction
      qed
    qed
  with (¬ (∀ x. P x)) show ?thesis by contradiction
  qed

theorem Drinker’s-Principle: ∃ x. drunk x −→ (∀ x. drunk x)
proof cases
  assume ∀ x. drunk x
  then have drunk a −→ (∀ x. drunk x) for a ..
  then show ?thesis ..
next
  assume ¬ (∀ x. drunk x)
  then have ∃ x. ¬ drunk x by (rule de-Morgan)
  then obtain a where ¬ drunk a ..
  have drunk a −→ (∀ 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

- 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 elimi-
ination rules. This also shows that the proof does not require classical reasoning.

**lemma** *iff-contradiction:*

**assumes** $*: \neg A \leftrightarrow A$

**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*: $\forall f :: 'a \Rightarrow 'a \Rightarrow bool. \forall A. \exists x. A = f x$

**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 $?D a \leftrightarrow f a a$ by (rule arg-cong)

then have $\neg f a a \leftrightarrow f a a$

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 $\alpha$. This version of the theorem states that for every function from $\alpha$ to its powerset, some subset is outside its range. The Isabelle/Isar proofs below uses HOL’s set theory, with the type $\alpha$ set and the operator $\text{range} :: (\alpha \Rightarrow \beta) \Rightarrow \beta$ set.

**theorem** $\exists S. S \notin \text{range} (f :: 'a \Rightarrow 'a \text{ set})
proof
let \(?S = \{x. x \notin f \, x\}\}
show \(?S \notin \text{range } f\)
proof
assume \(?S \in \text{range } f\)
then obtain \(y\) where \(?S = f \, y\)
then show \(\text{False}\)
proof (rule equalityCE)
assume \(y \in f \, y\)
assume \(y \in ?S\)
then have \(y \notin f \, y\)
with \(\langle y \in f \, y\rangle\) show \(?\text{thesis}\) by contradiction
next
assume \(y \notin ?S\)
assume \(y \notin f \, y\)
then have \(y \in ?S\)
with \(\langle y \notin ?S\rangle\) show \(?\text{thesis}\) by contradiction
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.

\textbf{theorem} \(\exists S. \, S \notin \text{range } (f :: 'a \Rightarrow 'a \, \text{set})\)

by \textit{best}

end

5 Structured statements within Isar proofs

\textbf{theory} \textit{Structured-Statements}
\textbf{imports} \textit{Main}
\textbf{begin}

5.1 Introduction steps

\textbf{notepad}
\textbf{begin}
\textbf{fix} \(A \, B :: \text{bool}\)
\textbf{fix} \(P :: 'a \Rightarrow \text{bool}\)

\textbf{have} \(A \rightarrow B\)
\textbf{proof}
\textbf{show} \(B\) if \(A\) using \textit{that} \(\langle\text{proof}\rangle\)
\textbf{qed}
have \neg A
proof
  show False if A using that \langle proof \rangle
qed

have \forall x. P x
proof
  show P x for x \langle proof \rangle
qed

end

5.2 If-and-only-if

\begin{notepad}
begin
  fix A B :: bool

  have A \iff B
  proof
    show B if A \langle proof \rangle
    show A if B \langle proof \rangle
  qed

next
  fix A B :: bool

  have iff-comm: (A \land B) \iff (B \land A)
  proof
    show B \land A if A \land B
    proof
      show B using that ..
      show A using that ..
    qed
    show A \land B if B \land A
    proof
      show A using that ..
      show B using that ..
    qed
  qed
\end{notepad}

Alternative proof, avoiding redundant copy of symmetric argument.

\begin{notepad}
begin
  fix A B :: bool

  have iff-comm: (A \land B) \iff (B \land A)
  proof
    show B \land A if A \land B for A B
    proof
      show B using that ..
      show A using that ..
    qed
    then show A \land B if B \land A
  qed
\end{notepad}
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 $P \, (\text{Suc } n)$ 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 $r: A \Rightarrow B \Rightarrow C$

  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
  end

next
  fix $a :: \, 'a$
  fix $A :: \, 'a \Rightarrow \text{bool}$
  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: \( A \rightarrow A \)
proof
assume \( A \)
show \( A \) by fact
qed

lemma K: \( A \rightarrow B \rightarrow A \)
proof
assume \( A \)
show \( B \rightarrow A \)
proof
show \( A \) by fact
qed
qed

lemma S: \((A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C\)
proof
assume \( A \rightarrow B \rightarrow C \)
show \((A \rightarrow B) \rightarrow A \rightarrow C \)
proof
assume \( A \rightarrow B \)
show \( A \rightarrow C \)
proof
assume \( A \)
show \( C \)
proof (rule mp)
show \( B \rightarrow C \) 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 \( A \rightarrow A \)
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** $A \rightarrow A$

*by rule*

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

**Lemma** $A \rightarrow A$

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** $A \rightarrow B \rightarrow A$

*proof (intro impI)*

*assume* $A$

*show* $A$ *by fact*

qed

Again, the body may be collapsed.

**Lemma** $A \rightarrow B \rightarrow A$

*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 $A \land B \rightarrow B \land A$.

\(^4\)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.

\begin{verbatim}
lemma A \land B \rightarrow B \land A
proof
  assume A \land B
  show B \land A
  proof
    show B by (rule conjunct2) fact
    show A by (rule conjunct1) fact
  qed
qed
\end{verbatim}

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

\begin{verbatim}
lemma A \land B \rightarrow B \land A
proof
  assume A \land B
  then show B \land A
  proof
    from (A \land B) show B ..
    from (A \land B) show A ..
  qed
qed
\end{verbatim}

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

\begin{verbatim}
lemma A \land B \rightarrow B \land A
proof
  assume A \land B
  then show B \land A
  proof
    — rule \texttt{conjE} of \( A \land B \)
    assume B A
    then show ?thesis .. — rule \texttt{conjI} of \( B \land A \)
  qed
qed
\end{verbatim}

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.

\begin{verbatim}
lemma A \land B \rightarrow B \land A
proof
\end{verbatim}
assume \( A \land B \)
from \( \langle A \land B \rangle \) have \( A \) ..
from \( \langle A \land B \rangle \) have \( B \) ..
from \( \langle B \rangle \langle A \rangle \) show \( B \land A \) ..
qed

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

lemma \( A \land B \implies B \land A \)
proof
\{
assume \( A \land B \)
from \( \langle A \land B \rangle \) have \( A \) ..
from \( \langle A \land B \rangle \) have \( B \) ..
from \( \langle B \rangle \langle A \rangle \) have \( B \land A \) ..
\}
then show \(?thesis\) .. — rule \( \text{impI} \)
qed

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

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

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

lemma \( A \land B \implies B \land A \)
proof
assume \( A \land B \)
then show \( B \land A \)
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 \( P \vee P \rightarrow P \). The proof below involves forward-chaining from \( P \vee P \), followed by an explicit case-analysis on the two identical cases.

```
lemma \( P \vee P \rightarrow P \)
proof
  assume \( P \vee P \)
  then show \( P \)
    proof
      -- rule disjE:
      \( A \vee B \) [A] [B] \\
      \( \vdots \) \( \vdots \) \\
      \( C \) \( C \)
    next
    assume \( P \)
    show \( P \) by fact
    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 \( P \vee P \rightarrow P \)
proof
  assume \( P \vee P \)
  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.

\textbf{lemma} \( P \lor P \rightarrow P \)
\textbf{proof}
\begin{itemize}
  \item assume \( P \lor P \)
  \item then show \( P \)
\end{itemize}
\textbf{qed}

\subsection{A quantifier proof}

To illustrate quantifier reasoning, let us prove \((\exists x. \ P (f \ x)) \rightarrow (\exists y. \ P y)\).
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 \texttt{fix} command augments the context by some new “arbitrary, but fixed” element; the \texttt{is} annotation binds term abbreviations by higher-order pattern matching.

\textbf{lemma} \((\exists x. \ P (f \ x)) \rightarrow (\exists y. \ P y)\)
\textbf{proof}
\begin{itemize}
  \item assume \( \exists x. \ P (f \ x) \)
  \item then show \( \exists y. \ P y \)
  \item proof (rule \texttt{exE}) \[ B \]
  \item fix \( a \)
  \item assume \( P (f \ a) \) (is \( P \) \texttt{?witness})
  \item then show \( \texttt{?thesis} \) by (rule \texttt{exI} [of \( P \) \texttt{?witness}])
\end{itemize}
\textbf{qed}
\textbf{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.

\textbf{lemma} \((\exists x. \ P (f \ x)) \rightarrow (\exists y. \ P y)\)
\textbf{proof}
\begin{itemize}
  \item assume \( \exists x. \ P (f \ x) \)
  \item then show \( \exists y. \ P y \)
  \item proof
    \begin{itemize}
      \item fix \( a \)
      \item assume \( P (f \ a) \)
    \end{itemize}
\end{itemize}

19
Explicit $\exists$-elimination as seen above can become quite cumbersome in practice. The derived Isar language element “obtain” provides a more handsome way to do generalized existence reasoning.

**lemma** $(\exists x. P (f x)) \rightarrow (\exists y. P y)$

**proof**

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

qed

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

### 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**: $A \land B \Rightarrow (A \Rightarrow B \Rightarrow C) \Rightarrow C$

**proof**

- assume $A \land B$
- assume $r$: $A \Rightarrow B \Rightarrow C$
- show $C$
  - proof (rule $r$)
    - show $A$ by (rule conjunct1) fact
    - show $B$ by (rule conjunct2) fact
  
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.

\[
\text{primrec } \text{compile} :: (\text{'adr, 'val} expr) \Rightarrow ((\text{'adr, 'val} instr) list)
\]

where

\[
\begin{align*}
\text{compile} \ (\text{Variable } x) &= [\text{Load } x] \\
\text{compile} \ (\text{Constant } c) &= [\text{Const } c] \\
\text{compile} \ (\text{Binop } f \ e1 \ e2) &= \text{compile } e2 @ \text{compile } e1 @ [\text{Apply } f]
\end{align*}
\]

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

\[
\text{lemma } \text{exec-append}:
\]

\[
\text{exec} \ (xs @ ys) \ stack \ env = \text{exec } ys \ (\text{exec } xs \ stack \ env) \ env
\]

\[
\text{proof} \ (\text{induct } xs \ \text{arbitrary: stack})
\]

\[
\begin{align*}
\text{case } \text{Nil} \\
\text{show } ?\text{case by simp}
\end{align*}
\]

next

\[
\begin{align*}
\text{case } (\text{Cons } x \ xs) \\
\text{show } ?\text{case}
\end{align*}
\]

proof (induct \(x\))

\[
\begin{align*}
\text{case } \text{Const} \\
\text{from } \text{Cons} \text{ show } ?\text{case by simp}
\end{align*}
\]

next

\[
\begin{align*}
\text{case } \text{Load} \\
\text{from } \text{Cons} \text{ show } ?\text{case by simp}
\end{align*}
\]

next

\[
\begin{align*}
\text{case } \text{Apply} \\
\text{from } \text{Cons} \text{ show } ?\text{case by simp}
\end{align*}
\]

qed

\[
\text{proof}
\]

\[
\begin{align*}
\text{have } \bigwedge \text{stack. } \text{exec} \ (\text{compile } e) \ stack \ env = \text{eval } e \ env \ # \ stack
\end{align*}
\]

\[
\text{proof} \ (\text{induct } e)
\]

\[
\begin{align*}
\text{case } \text{Variable} \\
\text{show } ?\text{case by simp}
\end{align*}
\]

next

\[
\begin{align*}
\text{case } \text{Constant} \\
\text{show } ?\text{case by simp}
\end{align*}
\]

next

\[
\begin{align*}
\text{case } \text{Binop} \\
\text{then show } ?\text{case by } (\text{simp add: exec-append})
\end{align*}
\]

qed

\[
\text{then show } ?\text{thesis by } (\text{simp add: execute-def})
\]

qed
In the proofs above, the simp method does quite a lot of work behind the
escenes (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
    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: \A stack. execute (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
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 ⇒ P 1 ⇒ (∀n. P (n + 1) ⇒ P n ⇒ P (n + 2))
⇒ 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

Isar version by Gertrud Bauer. Original tactic script by Larry Paulson. A few proofs of laws taken from [4].
\textbf{fix }n\\ \textbf{assume }P: \textit{coprime }\textit{(fib }\textit{(n }+\textit{1)}\textit{)} \textit{(fib }\textit{(n }+\textit{1 }+\textit{1)})\\ \textbf{have }fib (\textit{n }+\textit{2 }+\textit{1}) = fib (\textit{n }+\textit{1}) + fib (\textit{n }+\textit{2})\\ \textbf{also have }\ldots = fib (\textit{n }+\textit{2}) + fib (\textit{n }+\textit{1})\\ \textbf{also have }gcd (\textit{fib }\textit{(n }+\textit{2)}) \ldots = gcd (\textit{fib }\textit{(n }+\textit{2}}) (\textit{fib }\textit{(n }+\textit{1}) + fib (\textit{n }+\textit{1}) + \ldots = gcd (\textit{fib }\textit{(n }+\textit{2}) (\textit{fib }\textit{(n }+\textit{1}) + fib (\textit{n }+\textit{1}) + \ldots = \ldots = 1\\ \textbf{also have }\ldots = fib (\textit{n }+\textit{1}) (\textit{fib }\textit{(n }+\textit{1}) + fib (\textit{n }+\textit{1}) + \ldots = gcd (\textit{fib }\textit{(n }+\textit{1}) (\textit{fib }\textit{(n }+\textit{1}) + \ldots = \ldots = 1\\ \textbf{also have }\ldots = 1\\ \textbf{also have }\ldots = 1\\ \textbf{finally show }?P (\textit{n }+\textit{2})\\ \textbf{by }simp\\ qed\\

\textbf{lemma }gcd-mult-add: \textit{(0\cdot\textit{nat})} < \textit{n} \implies gcd (\textit{n }\cdot \textit{k }+ \textit{m}) \textit{n} = gcd \textit{m} \textit{n}\\ \textbf{proof }–\\ \textbf{assume }\textit{0} < \textit{n}\\ \textbf{then have }gcd (\textit{n }\cdot \textit{k }+ \textit{m}) \textit{n} = gcd \textit{n} \textit{m mod n}\\ \textbf{by }simp add: gcd-non-0-nat add.commute\\ \textbf{also from }\langle \textit{0} < \textit{n} \rangle \textbf{ have }\ldots = gcd \textit{m} \textit{n}\\ \textbf{by }simp add: gcd-non-0-nat\\ \textbf{finally show }?\textit{thesis}.\\ \textbf{qed}\\

\textbf{lemma }gcd-fib-add: gcd (\textit{fib }\textit{m}) (\textit{fib }\textit{(n }+\textit{m})) = gcd (\textit{fib }\textit{m}) (\textit{fib }\textit{n})\\ \textbf{proof }\langle \textit{cases }\textit{m} \rangle \langle \textit{cases }\langle \textit{Suc }\textit{k} \rangle \rangle\\ \textbf{case }\textit{0}\\ \textbf{then show }?\textit{thesis} \textbf{ by simp}\\ \textbf{next}\\ \textbf{case }\textit{(Suc }\textit{k})\\ \textbf{then have }gcd (\textit{fib }\textit{m}) (\textit{fib }\textit{(n }+\textit{m})) = gcd (\textit{fib }\textit{(n }+\textit{k }+\textit{1}) \textit{(fib }\textit{(k }+\textit{1}) + \ldots = gcd (\textit{fib }\textit{k }\cdot \textit{fib }\textit{n}) (\textit{fib }\textit{(k }+\textit{1}) + \ldots = gcd (\textit{fib }\textit{k }\cdot \textit{fib }\textit{n}) (\textit{fib }\textit{(k }+\textit{1}) + \ldots = \ldots = 1\\ \textbf{also have }\ldots = gcd (\textit{fib }\textit{k} \cdot \textit{fib }\textit{n}) (\textit{fib }\textit{(k }+\textit{1}) + \ldots = \ldots = 1\\ \textbf{also have }\ldots = 1\\ \textbf{using }\textit{Suc} \textbf{ by }simp add: gcd-commute\\ \textbf{finally show }?\textit{thesis}.\\ \textbf{qed}\\

\textbf{lemma }gcd-fib-diff: gcd (\textit{fib }\textit{m}) (\textit{fib }\textit{(n }-\textit{m})) = gcd (\textit{fib }\textit{m}) (\textit{fib }\textit{n}) \textit{if }\textit{m} \leq \textit{n}\\ \textbf{proof }–\\ \textbf{have }gcd (\textit{fib }\textit{m}) (\textit{fib }\textit{(n }-\textit{m})) = gcd (\textit{fib }\textit{m}) (\textit{fib }\textit{(n }-\textit{m }+\textit{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 (n mod m))
      = gcd (fib m) (fib (n - m)) by simp
    proof (cases n < m)
      case True
      then show ?thesis by simp
    next
      case False
      then have m ≤ n by simp
      from ⟨0 < m⟩ have n < m - n by simp
      with hyp have gcd (fib m) (fib (n mod m))
        = gcd (fib m) (fib (n - m)) by simp
      using m ≤ n by (rule gcd-fib-diff)
      finally have gcd (fib m) (fib (n mod m))
        = gcd (fib m) (fib n)
    with False show ?thesis by simp
  qed
finally show ?thesis .
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) .

27
9 Basic group theory

theory Group
  imports Main
begin

9.1 Groups and calculational reasoning

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

class group = times + one + inverse +
  assumes group-assoc: \((x \ast y) \ast z = x \ast (y \ast z)\)
  and group-left-one: \(I \ast x = x\)
  and group-left-inverse: \(\text{inverse} x \ast x = 1\)

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: \(x \ast \text{inverse} x = 1\)
proof
  have \(x \ast \text{inverse} x = 1 \ast (x \ast \text{inverse} x)\)
    by (simp only: group-left-one)
  also have \(= 1 \ast x \ast \text{inverse} x\)
    by (simp only: group-assoc)
  also have \(= \text{inverse} (\text{inverse} x) \ast \text{inverse} x \ast x \ast \text{inverse} x\)
    by (simp only: group-left-inverse)
  also have \(= \text{inverse} (\text{inverse} x) \ast (\text{inverse} x \ast x) \ast \text{inverse} x\)
    by (simp only: group-assoc)
  also have \(= \text{inverse} (\text{inverse} x) \ast 1 \ast \text{inverse} x\)
    by (simp only: group-left-inverse)
  also have \(= \text{inverse} (\text{inverse} x) \ast (1 \ast \text{inverse} x)\)
    by (simp only: group-assoc)
  also have \(= \text{inverse} (\text{inverse} x) \ast \text{inverse} x\)
    by (simp only: group-left-one)
  also have \(= 1\)
    by (simp only: group-left-inverse)
  finally show \(\text{thesis}\).
qed

With \(\text{group-right-inverse}\) already available, \(\text{group-right-one}\) is now established much easier.

theorem (in group) group-right-one: \(x \ast 1 = x\)
proof
have \( x \times 1 = x \times (\text{inverse } x \times x) \)
by \((\text{simp only: group-left-inverse})\)
also have \( \ldots = x \times \text{inverse } x \times x \)
by \((\text{simp only: group-assoc})\)
also have \( \ldots = 1 \times x \)
by \((\text{simp only: group-right-inverse})\)
also have \( \ldots = x \)
by \((\text{simp only: group-left-one})\)
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 “\(\ldots\)” is just a special term variable that is bound automatically to the argument of the last fact achieved by any local assumption or proven statement. In contrast to \(?thesis\), the “\(\ldots\)” variable is bound after the proof is finished.

There are only two separate Isar language elements for calculational proofs: “\textit{also}” for initial or intermediate calculational steps, and “\textit{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 \textit{also} and \textit{finally} derived language elements, calculations may be simulated by hand as demonstrated below.

\textbf{theorem (in group) } x \times I = x

proof –

have \( x \times 1 = x \times (\text{inverse } x \times x) \)
by \((\text{simp only: group-left-inverse})\)

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

have \( \ldots = x \times \text{inverse } x \times x \)
by \((\text{simp only: group-assoc})\)

note calculation = trans \[\text{OF calculation this}\]
— general calculational step: compose with transitivity rule

have \( \ldots = 1 \times x \)
by \((\text{simp only: group-right-inverse})\)

note calculation = trans \[\text{OF calculation this}\]
— general calculational step: compose with transitivity rule

\footnote{\textit{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 \( \ldots \)

from calculation
  — \( \ldots \) 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 \( (\cdot \colon \alpha \Rightarrow \alpha \Rightarrow \alpha, 1 \colon \alpha) \) are defined like this.

\begin{align*}
\text{class} & \text{ monoid } = \text{ times } + \text{ one } + \\
\text{ assumes} & \text{ monoid-assoc: } (x \cdot y) \cdot z = x \cdot (y \cdot z) \\
& \text{ and monoid-left-one: } 1 \cdot x = x \\
& \text{ and monoid-right-one: } x \cdot 1 = x
\end{align*}

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 \( \exists x \cdot (1::\alpha) = x \)), we may still instantiate group \( \subseteq \) monoid properly as follows.

\begin{align*}
\text{instance} & \text{ group } \subseteq \text{ monoid} \\
\text{ by} & \text{ intro-classes } \\
& \text{ (rule group-assoc, } \\
& \text{ rule group-left-one, } \\
& \text{ rule group-right-one)}
\end{align*}

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 \( 1 = x * \text{inverse } x \)
  - By \( \text{simp only: group-right-inverse} \)
- Also have \( \ldots = (e * x) * \text{inverse } x \)
  - By \( \text{simp only: eq} \)
- Also have \( \ldots = e * (x * \text{inverse } x) \)
  - By \( \text{simp only: group-assoc} \)
- Also have \( \ldots = e * 1 \)
  - By \( \text{simp only: group-right-inverse} \)
- Also have \( \ldots = e \)
  - By \( \text{simp only: group-right-one} \)

Finally show \( \text{thesis} \).

**Qed**

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

**Theorem (in Group) Group-Inverse-Equality:**

**Assumes** \( eq : x' * x = 1 \)

**Shows** \( \text{inverse } x = x' \)

**Proof**

- Have \( \text{inverse } x = 1 * \text{inverse } x \)
  - By \( \text{simp only: group-left-one} \)
- Also have \( \ldots = (x' * x) * \text{inverse } x \)
  - By \( \text{simp only: eq} \)
- Also have \( \ldots = x' * (x * \text{inverse } x) \)
  - By \( \text{simp only: group-assoc} \)
- Also have \( \ldots = x' * 1 \)
  - By \( \text{simp only: group-right-inverse} \)
- Also have \( \ldots = x' \)
  - By \( \text{simp only: group-right-one} \)

Finally show \( \text{thesis} \).

**Qed**

The inverse operation has some further characteristic properties.

**Theorem (in Group) Group-Inverse-Times:** \( \text{inverse } (x * y) = \text{inverse } y * \text{inverse } x \)

**Proof** (rule group-inverse-equality)

**Show** \( (\text{inverse } y * \text{inverse } x) * (x * y) = 1 \)

**Proof**

- Have \( (\text{inverse } y * \text{inverse } x) * (x * y) = \)
  - \( (\text{inverse } y * (\text{inverse } x * x)) * y \)
  - By \( \text{simp only: group-assoc} \)
- Also have \( \ldots = (\text{inverse } y * 1) * y \)
  - By \( \text{simp only: group-left-inverse} \)
- Also have \( \ldots = \text{inverse } y * y \)
  - By \( \text{simp only: group-right-one} \)
- Also have \( \ldots = 1 \)
  - By \( \text{simp only: group-left-inverse} \)
finally show ?thesis.
sqed
sqed

\textbf{theorem (in group) inverse-inverse: }inverse \ (inverse \ x) = x
\textbf{proof (rule group-inverse-equality)}
show \ x \ast \ \text{inverse} \ \ x = \ \text{one}
by (simp only: group-right-inverse)
sqed

\textbf{theorem (in group) inverse-inject:}
assumes \ eq: \ inverse \ x = \ inverse \ y
shows \ x = y
\textbf{proof –}
\hspace{0.5em} have \ x = x \ast \ 1
by (simp only: group-right-one)
\hspace{0.5em} also have \ \ldots = \ x \ast (\text{inverse} \ y \ast \ y)
by (simp only: group-left-inverse)
\hspace{0.5em} also have \ \ldots = \ x \ast (\text{inverse} \ x \ast \ y)
by (simp only: eq)
\hspace{0.5em} also have \ \ldots = (x \ast \text{inverse} \ x) \ast \ y
by (simp only: group-assoc)
\hspace{0.5em} also have \ \ldots = 1 \ast y
by (simp only: group-right-inverse)
\hspace{0.5em} also have \ \ldots = y
by (simp only: group-left-one)
finally show \ ?thesis.
sqed

end

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

\textbf{theory Group-Context}
\textbf{imports Main}
\begin
hypothetical group axiomatization
\textbf{context}
\hspace{1em} fixes \ prod :: 'a \Rightarrow 'a \Rightarrow 'a \ (infixl \ \odot \ 70)
\hspace{1em} and \ one :: 'a
\hspace{1em} and \ inverse :: 'a \Rightarrow 'a
\hspace{1em} assumes \ assoc: \ (x \odot \ y) \odot \ z = x \odot (y \odot \ z)
\hspace{1em} and \ left-one: \ one \odot \ x = x
\hspace{1em} and \ left-inverse: \ \text{inverse} \ x \odot \ x = \ \text{one}
\begin
some consequences
lemma right-inverse: $x \odot \text{inverse } x = \text{one}$
proof
  have $x \odot \text{inverse } x = \text{one} \odot (x \odot \text{inverse } x)$
    by (simp only: left-one)
  also have $\ldots = \text{one} \odot x \odot \text{inverse } x$
    by (simp only: assoc)
  also have $\ldots = \text{inverse } (\text{inverse } x) \odot \text{inverse } x \odot x \odot \text{inverse } x$
    by (simp only: left-inverse)
  also have $\ldots = \text{inverse } (\text{inverse } x) \odot (\text{inverse } x \odot x) \odot \text{inverse } x$
    by (simp only: assoc)
  also have $\ldots = \text{inverse } (\text{inverse } x) \odot \text{one} \odot \text{inverse } x$
    by (simp only: left-inverse)
  also have $\ldots = \text{inverse } (\text{inverse } x) \odot \text{one} \odot \text{inverse } x$
    by (simp only: assoc)
  also have $\ldots = \text{one}$
    by (simp only: left-inverse)
  finally show $?\text{thesis }.$
qed

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

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

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 ⇒ 'a ⇒ 'a (infixl ⊙ 70)
  and one :: 'a
  and inverse :: 'a ⇒ 'a
assume assoc: (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)
  and left-one: one ⊙ x = x
  and left-inverse: inverse x ⊙ x = one
  for x y z
some consequences

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

qed

have right-one: \(x \odot one = x\) for \(x\)
proof –
  have \(x \odot one = x \odot (inverse x \odot x)\)
    by (simp only: left-inverse)
  also have \ldots = \(x \odot inverse x \odot x\)
    by (simp only: assoc)
  also have \ldots = \(one \odot x\)
    by (simp only: right-inverse)
  also have \ldots = \(x\)
    by (simp only: left-one)
finally show \?thesis.

qed

have one-equality: \(one = e\) if \(eq\): \(e \odot x = x\) for \(e\) \(x\)
proof –
  have \(one = x \odot inverse x\)
    by (simp only: right-inverse)
  also have \ldots = \((e \odot x) \odot inverse x\)
    by (simp only: eq)
  also have \ldots = \(e \odot (x \odot inverse x)\)
    by (simp only: assoc)
  also have \ldots = \(e \odot one\)
    by (simp only: right-inverse)
  also have \ldots = \(e\)
    by (simp only: right-one)
finally show \?thesis.

qed

have inverse-equality: \(inverse x = x'\) if \(eq\): \(x' \odot x = one\) for \(x\) \(x'\)
proof –
  have \(inverse x = one \odot inverse x\)
    by (simp only: left-one)
  also have \ldots = \((x' \odot x) \odot inverse x\)
also have \( \ldots = x' \circ (x \circ inverse x) \)
by (simp only: assoc)
also have \( \ldots = x' \circ one \)
by (simp only: right-inverse)
also have \( \ldots = x' \)
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 =\n  Basic \('a \Rightarrow 'a\)
  | Seq \('a com 'a com \((;)/-\) [60, 61] 60)\n  | Cond \('a bexp 'a com 'a com\)
  | While \('a bexp 'a assn 'a com\)

abbreviation Skip (SKIP)
where SKIP \equiv Basic id

type-synonym \('a sem = 'a \Rightarrow 'a \Rightarrow bool\)

primrec iter :: \(\text{nat} \Rightarrow 'a bexp \Rightarrow 'a sem \Rightarrow 'a sem\)
where
iter 0 b S s s' \iff s \notin b \land s = s'
| iter (Suc n) b S s s' \iff s \in b \land (\exists s''. S s s'' \land iter n b S s'' s')

primrec Sem :: \('a com \Rightarrow 'a sem\)
where
Sem (Basic f) s s' \iff s' = f s
| Sem (c1; c2) s s' \iff (\exists s''. Sem c1 s s'' \land Sem c2 s'' s')
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 straightforward manner as follows.

**theorem** seq: \( \vdash P c1 Q \implies \vdash Q c2 R \implies \vdash P (c1; c2) R \)

**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: \( P' \subseteq P \implies \vdash P c Q \implies Q \subseteq Q' \implies \vdash P' c Q' \)

**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 $c \cdot s \cdot s'$
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`: $\vdash (P \cap \neg b) c2 Q$
shows $\vdash P (\text{Cond } b c1 c2) Q$

**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 $s \in P \cap \neg b$ by simp
qed
qed

The *while* rule is slightly less trivial — it is the only one based on recur-
sion, 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 $\vdash P (\text{While } b X c) (P \cap \neg b)$

**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 $s' \in P \cap \neg b$
proof (induct n arbitrary: s)
  case 0
  then show ?case by auto
next
  case (Suc n)
  then obtain s'' where b: s ∈ b and sem: Sem c s s''
    and iter: iter n b (Sem c) s'' s' by auto
  from Suc and b have s ∈ P ∩ b by simp
  with body sem have s'' ∈ 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 ⇒ ('a ⇒ 'b)
-antiquote :: ('a ⇒ 'b) ⇒ 'b (''-/'' [1000] 1000)
-Subst :: 'a bexp ⇒ 'b ⇒ idt ⇒ 'a bexp ('''-/''-''/'' [1000] 999)
-Assert :: 'a ⇒ 'a set ("{|-|} [0] 1000)
-Assign :: idt ⇒ 'b ⇒ 'a com (('':=/-''''/''/'' [70, 65] 61)
-Cond :: 'a bexp ⇒ 'a com ⇒ 'a com ⇒ 'a com
  ("IF |- THEN |- ELSE |- FI" [0, 0, 0] 61)
-While-inv :: 'a bexp ⇒ 'a assn ⇒ 'a com ⇒ 'a com
  ("WHILE |- INV |- DO |- OD" [0, 0, 0] 61)
-While :: 'a bexp ⇒ 'a com ⇒ 'a com
  ("WHILE |- DO |- OD" [0, 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 \([\text{trans}]: \vdash P \land Q \Rightarrow P' \subseteq P \Rightarrow \vdash P' \land Q\) 
by \((\text{unfold Valid-def})\) blast

lemma \([\text{trans}]: P' \subseteq P \Rightarrow \vdash P \land Q \Rightarrow \vdash P' \land Q\)
by \((\text{unfold Valid-def})\) blast

lemma \([\text{trans}]: Q \subseteq Q' \Rightarrow \vdash P \land Q \Rightarrow \vdash P \land Q'\)
by \((\text{unfold Valid-def})\) blast

lemma \([\text{trans}]: \vdash P \land Q \Rightarrow Q \subseteq Q' \Rightarrow \vdash P \land Q'\)
by \((\text{unfold Valid-def})\) blast

lemma \([\text{trans}]: \vdash P \land Q \Rightarrow Q' \subseteq Q \Rightarrow \vdash P \land Q'\)
by \((\text{unfold Valid-def})\) blast

Identity and basic assignments.\(^8\)

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

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

Note that above formulation of assignment corresponds to our preferred way
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 \('x := 'a\) also contains \(x\)-update.\(^9\)

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

lemmas \([\text{trans, intro?}] = \text{seq}\)

---

\(^8\)The \(\text{hoare}\) 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]: ⊢ P c1;(c2;c3) Q ⟷ ⊢ P (c1;c2);c3 Q 
by (auto simp add: Valid-def)

Conditional statements.

lemmas [trans, intro?] = cond

lemma [trans, intro?]:
⊢ {`P ∧ `b} c1 Q
  ⟷ ⊢ {`P ∧ ¬ `b} c2 Q
  ⟷ ⊢ {`P} IF `b THEN c1 ELSE c2 FI Q
by (rule cond) (simp-all add: Valid-def)

While statements — with optional invariant.

lemma [intro?]: ⊢ (P ∩ b) c P ⟷ ⊢ P (While b P c) (P ∩ ¬b)
by (rule while)

lemma [intro?]: ⊢ (P ∩ b) c P ⟷ ⊢ P (While b undefined c) (P ∩ ¬b)
by (rule while)

lemma [intro?]:
⊢ {`P ∧ `b} c {`P}
  ⟷ ⊢ {`P} WHILE `b INV {`P} DO c OD {`P ∧ ¬ `b}
by (simp add: while Collect-conj-eq Collect-neg-eq)

lemma [intro?]:
⊢ {`P ∧ `b} c {`P}
  ⟷ ⊢ {`P} WHILE `b DO c OD {`P ∧ ¬ `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 ⊆ q ⟷ Valid p (Basic id) q
by (auto simp add: Valid-def)

lemma BasicRule: p ⊆ {s. f s ∈ q} ⟷ Valid p (Basic f) q
by (auto simp: Valid-def)
lemma SeqRule: \( \text{Valid } P \ c_1 \ Q \implies \text{Valid } Q \ c_2 \ R \implies \text{Valid } P \ (c_1; c_2) \ R \)
by \( \text{(auto simp: Valid-def)} \)

lemma CondRule:
\[
p \subseteq \{ (s, (s \in b \implies s \in w)) \land (s \notin b \implies s \in w') \} \implies \text{Valid } w \ c_1 \ q \implies \text{Valid } w' \ c_2 \ q \implies \text{Valid } p \ (\text{Cond } b \ c_1 \ c_2) \ q
\]
by \( \text{(auto simp: Valid-def)} \)

lemma iter-aux:
\[
\forall s s'. \ \text{Sem } c \ s \ s' \implies s \in I \land s \in b \implies s' \in I \implies (\forall s s'. s \in I \implies \text{iter } n \ b \ (\text{Sem } c) \ s \ s' \implies s' \in I \land s' \notin b)
\]
by \( \text{(induct } n \text{) auto} \)

lemma WhileRule:
\[
p \subseteq i \implies \text{Valid } (i \cap b) \ c \ i \implies i \cap (-b) \subseteq q \implies \text{Valid } p \ (\text{While } b \ i \ c) \ q
\]
apply \( \text{(clarsimp simp: Valid-def)} \)
apply \( \text{(drule iter-aux)} \)
prefer 2
apply assumption
apply blast
apply blast
done

lemma Compl-Collect: \(- \text{Collect } b = \{ x. \neg b x \}\)
by blast

lemmas AbortRule = SkipRule — dummy version

ML-file (~~/src/HOL/Hoare/hoare-tac.ML)

method-setup hoare =
\( \langle \text{Scan.succeed (fn ctxt =>} \)
\( \langle \text{SIMPLE-METHOD' } \)
\( \langle \text{(Hoare.hoare-tac ctxt} \)
\( \langle \text{(simp-tac (put-simpset HOL-basic-ss ctxt addsimps [\(\text{thm Record.K-record-comp}])} \}
\rangle)) \rangle \rangle \)

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 ⊢ { smelling (λ. (2 * ´N))) ∈ ⟨ ´N = 10 ⟩} ´N := 2 * ´N ⟨ ´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 ⊢ { True } ´N := 10 ⟨ ´N = 10 ⟩
  by hoare
```

```
lemma ⊢ { ´N = 5 } ´N := 2 * ´N ⟨ ´N = 10 ⟩
  by hoare simp
```

```
lemma ⊢ { ´N + 1 = a + 1 } ´N := ´N + 1 ⟨ ´N = a + 1 ⟩
  by hoare
```

```
lemma ⊢ { ´N = a } ´N := ´N + 1 ⟨ ´N = a + 1 ⟩
  by hoare simp
```

```
lemma ⊢ { a = a ∧ b = b } ´M := a; ´N := b ⟨ ´M = a ∧ ´N = b ⟩
  by hoare
```

```
lemma ⊢ { True } ´M := a; ´N := b ⟨ ´M = a ∧ ´N = b ⟩
```

44
by hoare

**lemma**

\[ \vdash \{ \begin{array}{l} M = a \land N = b \\ I := M; M := N; N := I \\ M = b \land N = a \end{array} \} \]

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** \[ \vdash \{ N = a \} \]

by hoare

**lemma** \[ \vdash \{ x = a \} \]

oops

**lemma**

\[ \text{Valid } \{ s. x s = a \} \ (\text{Basic } (\lambda s. x\text{-update } (x s) s)) \{ s. x s = n \} \]

— 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** \[ \vdash \{ M = N \} \]

\[ \vdash \{ M := M + 1 \} \]

by hoare simp

---

by hoare

by simp

---

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**

\[ \vdash \{ \mathbf{M} = 0 \land \mathbf{S} = 0 \} \]

WHILE \( \mathbf{M} \neq a \)

\[ \mathbf{S} := \mathbf{S} + b; \mathbf{M} := \mathbf{M} + 1 \text{ OD} \]

\[ \{ \mathbf{S} = a \ast b \} \]

**Proof** —

let \( \vdash - \ \text{while} - = \ ?\text{thesis} \)

let \( \{ \ \text{"inv} \} = \{ \ \mathbf{S} = \mathbf{M} \ast b \} \)

\[
\begin{align*}
\text{have} & \ \{ \mathbf{M} = 0 \land \mathbf{S} = 0 \} \subseteq \{ \text{"inv} \} \text{ by auto} \\
\text{also have} & \ \vdash \ldots \ ?\text{while} \ \{ \ ?\text{inv} \land \neg (\mathbf{M} \neq a) \} \\
\text{proof} & \ \\
\text{let} & \ \text{"c} = \ \mathbf{S} := \mathbf{S} + b; \mathbf{M} := \mathbf{M} + 1 \\
\text{have} & \ \{ \ ?\text{inv} \land \mathbf{M} \neq a \} \subseteq \{ \mathbf{S} + b = (\mathbf{M} + 1) \ast b \} \\
& \ \text{by auto} \\
\text{also have} & \ \vdash \ldots \ ?\text{c} \ \{ \ ?\text{inv} \} \text{ by hoare} \\
\text{finally show} & \ \vdash \ \{ \ ?\text{inv} \land \mathbf{M} \neq a \} \ ?\text{c} \ \{ \ ?\text{inv} \} . \\
\text{qed} \\
\text{also have} & \ \ldots \subseteq \ \{ \mathbf{S} = a \ast b \} \text{ by auto} \\
\text{finally show} & \ ?\text{thesis} . \\
\text{qed}
\end{align*}
\]

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**

\[ \vdash \{ \mathbf{M} = 0 \land \mathbf{S} = 0 \} \]

WHILE \( \mathbf{M} \neq a \)

INV \( \{ \mathbf{S} = \mathbf{M} \ast b \} \)

\[ \mathbf{S} := \mathbf{S} + b; \mathbf{M} := \mathbf{M} + 1 \text{ OD} \]

\[ \{ \mathbf{S} = a \ast b \} \]

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.
\textbf{Theorem}

\[ \vdash \{ \mathit{True} \} \]
\[ 'S := 0; 'I := 1; \]
\[ \text{WHILE } 'I \neq n \]
\[ \text{DO} \]
\[ 'S := 'S + 'I; \]
\[ 'I := 'I + 1 \]
\[ \text{OD} \]
\[ \{ 'S = \left( \sum_{j<n} j \right) \} \]

\textbf{Proof} –

\begin{enumerate}
\item \textit{let } \sum = \lambda k :: \textbf{nat}. \sum j<k. j \\
\item \textit{let } \mathit{inv} = \lambda s i :: \textbf{nat}. s = \sum i \\
\end{enumerate}

\begin{enumerate}
\item \textit{have } \vdash \{ \mathit{True} \} \quad 'S := 0; 'I := 1 \quad \mathit{inv} \quad 'S \quad 'I \\
\item \textbf{proof} –
\item \textit{have } \mathit{True} \rightarrow 0 = \sum 1 \\
\item \textit{by } \texttt{simp} \\
\item \textit{also have } \vdash \{ \ldots \} \quad 'S := 0; 'I := 1 \quad \mathit{inv} \quad 'S \quad 'I \\
\item \textit{by } \texttt{hoare} \\
\item \textit{finally show } \mathit{thesis} . \\
\item \textbf{qed} \\
\item \textit{also have } \vdash \ldots \texttt{while} \quad \mathit{inv} \quad 'S \quad 'I \quad \neg 'I \neq n \\
\item \textbf{proof} –
\item \textit{let } \mathit{body} = \quad 'S := 'S + 'I; 'I := 'I + 1 \\
\item \textit{have } \mathit{inv} \quad s \quad i \quad \neg i \neq n \rightarrow \mathit{inv} \quad (s + i) \quad (i + 1) \quad \text{for } s \quad i \\
\item \textit{by } \texttt{simp} \\
\item \textit{also have } \vdash \{ 'S + 'I = \sum (i + 1) \} \quad \mathit{body} \quad \mathit{inv} \quad 'S \quad 'I \\
\item \textit{by } \texttt{hoare} \\
\item \textit{finally show } \vdash \{ \mathit{inv} \quad 'S \quad 'I \quad \neg 'I \neq n \} \quad \mathit{body} \quad \mathit{inv} \quad 'S \quad 'I . \\
\item \textbf{qed} \\
\item \textit{also have } s = \sum i \quad \neg i \neq n \rightarrow s = \sum n \quad \text{for } s \quad i \\
\item \textit{by } \texttt{simp} \\
\item \textit{finally show } \mathit{thesis} . \\
\item \textbf{qed} \\
\end{enumerate}

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

\textbf{Theorem}

\[ \vdash \{ \mathit{True} \} \]
\[ 'S := 0; 'I := 1; \]
\[ \text{WHILE } 'I \neq n \]
\[ \text{INV} \quad \{ 'S = \left( \sum_{j<n} 'I \cdot j \right) \} \]
\[ \text{DO} \]
\[ 'S := 'S + 'I; \]
\[ 'I := 'I + 1 \]
\[ \text{OD} \]
\[ \{ 'S = \left( \sum_{j<n} 'I \cdot j \right) \} \]

\textbf{Proof} –

\begin{enumerate}
\item \textit{let } ?\mathit{sum} = \lambda k :: \textbf{nat}. \sum j<k. j \\
\item \textit{let } ?\mathit{inv} = \lambda s i :: \textbf{nat}. s = ?\mathit{sum} i \\
\end{enumerate}
let \(?sum = \lambda k :: \text{nat}. \sum_{j < k} j\)
let \(?inv = \lambda s i :: \text{nat}. s = ?sum i\)
show \(?thesis\)
proof [hoare]
show \(?inv 0 1\) by simp
show \(?inv (s + i) (i + 1)\) if \(?inv s i \land i \neq n\) for \(s i\)
using that by simp
show \(s = ?sum n\) if \(?inv s i \land i \neq n\) 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
\(\vdash \{ | \text{True} \}\)
\('S := 0; 'I := 1;\)
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 \textit{timeit} inserts an extra variable to keep track of the elapsed time.

record \(\text{tstate} = \text{time} :: \text{nat}\)

type-synonym \('a\ \text{time} = (| \text{time} :: \text{nat}, \ldots :: 'a|)\)

primrec \textit{timeit} :: \('a\ \text{time} \text{ com} \Rightarrow 'a\ \text{time} \text{ com}\)
where
\(\textit{timeit} (\text{Basic } f) = (\text{Basic } f; \text{Basic}(\lambda s. s|\text{time} := \text{Suc} \ (\text{time} s)|))\)
| \(\textit{timeit} (c1; c2) = (\textit{timeit} c1; \textit{timeit} c2)\)
| \(\textit{timeit} (\text{Cond } b \ c1 \ c2) = \text{Cond } b \ (\textit{timeit} c1) \ (\textit{timeit} c2)\)
| \(\textit{timeit} (\text{While } b \ iv \ c) = \text{While } b \ iv \ (\textit{timeit} c)\)

record \(\text{tvars} = \text{tstate} +\)
\(I :: \text{nat}\)
\(J :: \text{nat}\)

lemma \textit{lem}: \((0 :: \text{nat}) < n \implies n + n \leq \text{Suc} \ (n \ast n)\)
by [induct n] simp-all

48
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 ⇒ 'a set set for A :: 'a set set
  where
    empty: \{} \in tiling A
| Un: a ∪ t \in tiling A if a ∈ A and t ∈ tiling A and a ⊆ − t

The union of two disjoint tilings is a tiling.

lemma tiling-Un:
  assumes t ∈ tiling A
and \( u \in \text{tiling A} \)
and \( t \cap u = \{\} \)
shows \( t \cup u \in \text{tiling A} \)

proof

let \(?T = \text{tiling A} \)
from \( \langle t \in \?T \rangle \) and \( \langle t \cap u = \{\} \rangle \)
show \( t \cup u \in \?T \)

proof (induct \( t \))

case empty

with \( \langle u \in \?T \rangle \) show \( \{\} \cup u \in \?T \) by simp

next

case \( (\text{Un a t}) \)

show \( (a \cup t) \cup u \in \?T \)

proof

have \( a \cup (t \cup u) \in \?T \)
using \( \langle a \in A \rangle \)

proof (rule tiling.Un)

from \( \langle (a \cup t) \cap u = \{\} \rangle \) have \( t \cap u = \{\} \) by blast

then show \( t \cup u \in \?T \) by (rule Un)

from \( a \subseteq - t \) and \( \langle (a \cup t) \cap u = \{\} \rangle \)

show \( a \subseteq - (t \cup u) \) by blast

qed

also have \( a \cup (t \cup u) = (a \cup t) \cup u \)

by (simp only: Un-assoc)

finally show \( \simthesis \).

qed

qed

14.2 Basic properties of “below”

definition below :: \( \text{nat} \Rightarrow \text{nat set} \)

where below \( n = \{i.\  i < n\} \)

lemma below-less-iff [iff]: \( i \in \text{below k} \iff i < k \)

by (simp add: below-def)

lemma below-0: below 0 = \( \{\} \)

by (simp add: below-def)

lemma Sigma-Suc1: \( m = n + 1 \Rightarrow \text{below m} \times B = (\{n\} \times B) \cup (\text{below n} \times B) \)

by (simp add: below-def less-Suc-eq) blast

lemma Sigma-Suc2:

\( m = n + 2 \Rightarrow \)
\( A \times \text{below m} = (A \times \{n\}) \cup (A \times \{n + 1\}) \cup (A \times \text{below n}) \)

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 evnodd-D: 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 \ldots \in \mathcal{T}
proof (rule tiling.Un)
  have \{(i, 2 \times n), (i, 2 \times n + 1)\} \in \text{domino}
  by (rule domino.horiz)
  also have \{(i, 2 \times n), (i, 2 \times n + 1)\} = \{a\} by blast
finally show \ldots \in \text{domino}.
  show \mathcal{B} n \in \mathcal{T} by (rule Suc)
  show \{a\} \subseteq \mathcal{B} n by blast
qed
finally show ?case.
qed

lemma dominoes-tile-matrix:
  below m \times below (2 \times n) \subseteq \text{tiling domino}
(is \mathcal{B} m \in \mathcal{T})
proof (induct m)
  case 0
  show ?case by (simp add: below-0 tiling.empty)
next
  case (Suc m)
  let \mathcal{T} = \{m\} \times below (2 \times n)
  have \mathcal{B} (Suc m) = \mathcal{T} \cup \mathcal{B} m by (simp add: Sigma-Suc)
  also have \ldots \in \mathcal{T}
  proof (rule tiling-Un)
    show \mathcal{T} \in \mathcal{T} by (rule dominoes-tile-row)
    show \mathcal{B} m \in \mathcal{T} by (rule Suc)
    show \mathcal{T} \cap \mathcal{B} m = {} by blast
  qed
finally show ?case.
qed

lemma domino-singleton:
  assumes d \in \text{domino}
  and b < 2
  shows \exists i j. evnodd d b = \{(i, j)\} (is \mathcal{P} d)
using assms
proof induct
  from (b < 2) have b-cases: b = 0 \lor b = 1 by arith
  fix i j
  note [simp] = evnodd-empty evnodd-insert mod-Suc
from b-cases show \mathcal{P} \{(i, j), (i, j + 1)\} by rule auto
from b-cases show \mathcal{P} \{(i, j), (i + 1, j)\} by rule auto
qed

lemma domino-finite:
  assumes d \in \text{domino}
  shows finite d
using assms

52
proof \textit{induct}
  \begin{align*}
  \text{fix } & \ i, j : \text{nat} \\
  \text{show } & \ \text{finite } \{(i, j), (i, j + 1)\} \text{ by (intro finite.intros)} \\
  \text{show } & \ \text{finite } \{(i, j), (i + 1, j)\} \text{ by (intro finite.intros)} \\
  \end{align*}
qed

14.5 Tilings of dominoes

lemma \textit{tiling-domino-finite}:
  \begin{align*}
  \text{assumes } & \ t : \ t \in \text{tiling domino} \ \text{(is } t \in \mathcal{T} \text{)} \\
  \text{shows } & \ \text{finite } t \ \text{(is } \mathcal{F} t \text{)} \\
  \end{align*}
using \textit{t}
proof \textit{induct}
  \text{case empty}
  \begin{align*}
  \text{show } & \ \mathcal{F} \{} \text{ by (rule finite.emptyI)} \\
  \end{align*}
fix \ a \ \text{assume } \mathcal{F} t
proof (case \textit{Un a t})
  \begin{align*}
  \text{let } & \ ?e = \text{evnodd} \\
  \text{note hyp = } & \langle \text{card } (?e t 0) = \text{card } (?e t 1) \rangle \\
  \text{and at = } & \langle ?e \subseteq - t \rangle \\
  \text{have card-suc: } & \text{card } (?e a \cup b) = \text{Suc } (\text{card } (?e t b)) \text{ if } b < 2 \text{ for } b :: \text{nat} \\
  \end{align*}
proof –
  \begin{align*}
  \text{have } & ?e (a \cup b) = ?e a b \cup ?e t b \text{ by (rule evnodd-Un)} \\
  \text{also obtain } & i, j \text{ where } e : ?e a b = \{(i, j)\} \\
  \text{proof –} \\
  \text{from } & (a \in \text{domino}) \text{ and } (b < 2) \\
  \text{have } & \exists i, j. ?e a b = \{(i, j)\} \text{ by (rule domino-singleton)} \\
  \text{then show } & ?thesis \text{ by (blast intro: that)} \\
  \end{align*}
qed
also have \ldots \cup ?e t b = \text{insert } (i, j) (?e t b) \text{ by simp}
also have \text{card } \ldots = \text{Suc } (\text{card } (?e t b))
proof (rule card-insert-disjoint)
  \begin{align*}
  \text{from } & t \in \text{tiling domino} \text{ have finite } t \\
  \text{by (rule tiling-domino-finite)} \\
  \text{then show } & \text{finite } (?e t b) \\
  \text{by (rule evnodd-finite)} \\
  \text{from } e \ \text{have } (i, j) \in ?e a b \text{ by simp}
  \end{align*}
with at show \((i, j) \notin \mathcal{E} t b\) by (blast dest: evnoddD)
qed
finally show \(\text{thesis}\).
qed

then have \(\text{card} \ (\mathcal{E} e (a \cup t) 0) = \text{Suc} \ (\text{card} \ (\mathcal{E} e t 0))\) by simp
also from hyp have \(\text{card} \ (\mathcal{E} e t 0) = \text{card} \ (\mathcal{E} e t 1)\).
also from card-suc have \(\text{Suc} \ldots = \text{card} \ (\mathcal{E} e (a \cup t) 1)\)
by simp
finally show \(\text{case}\).
qed

14.6 Main theorem

definition mutilated-board :: \(\text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ set}\)
where mutilated-board \(m\ n = \)
\(\text{below} \ (2 \ast (m + 1)) \times \text{below} \ (2 \ast (n + 1))\) − \{(0, 0)\} − \{(2 \ast m + 1, 2 \ast n + 1)\}

theorem mutil-not-tiling: mutilated-board \(m\ n \notin\ \text{tiling domino}
proof (unfold mutilated-board-def)
let \(\mathcal{T} = \text{tiling domino}
let \(\mathcal{T}' = \text{below} \ (2 \ast (m + 1)) \times \text{below} \ (2 \ast (n + 1))\)
let \(\mathcal{T}' = \mathcal{T} - \{(0, 0)\}\)
let \(\mathcal{T}'' = \mathcal{T}' - \{(2 \ast m + 1, 2 \ast n + 1)\}\)

show \(\mathcal{T}'' \notin \mathcal{T}\)
proof
have \(t: \mathcal{T} \in \mathcal{T}\) by (rule dominoes-tile-matrix)
assume \(t': \mathcal{T}'' \in \mathcal{T}\)

let \(\mathcal{E} = \text{evnodd}\)
have \(\text{fin}: \text{finite} \ (\mathcal{E} e \mathcal{T} 0)\)
by (rule evnodd-finite, rule tiling-domino-finite, rule t)

note \(\text{simp} = \text{evnodd-iff}\ \text{evnodd-empty}\ \text{evnodd-insert}\ \text{evnodd-Diff}\)

have \(\text{card} \ (\mathcal{E} e \mathcal{T}' 0) < \text{card} \ (\mathcal{E} e \mathcal{T}' 0)\)
proof −
have \(\text{card} \ (\mathcal{E} e \mathcal{T}' 0) - \{(2 \ast m + 1, 2 \ast n + 1)\}\)
< \(\text{card} \ (\mathcal{E} e \mathcal{T}' 0)\)
proof (rule card-Diff1-less)
from - fin show \(\text{finite} \ (\mathcal{E} e \mathcal{T}' 0)\)
by (rule finite-subset) auto
show \(2 \ast m + 1, 2 \ast n + 1 \in \mathcal{E} e \mathcal{T}' 0\) by simp
qed
then show \(\text{thesis}\) by simp
qed
also have \(\ldots < \text{card} \ (\mathcal{E} e \mathcal{T} 0)\)
proof −
have \(0, 0 \in \mathcal{E} e \mathcal{T} 0\) by simp

54
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

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: \( \forall n. f (f n) < f (Suc n) \)
  shows \( f n = n \)
proof (rule order-antisym)
  show ge: \( n \leq f n \) for \( n \)
    proof (induct f n arbitrary: \( n \) rule: less-induct)
      case less
      show \( n \leq f n \)
    proof (cases n)
      case (Suc m)
      from f-ax have \( f (f m) < f n \) by (simp only: Suc)
      with less have \( f m \leq f (f m) \).
      also from f-ax have \( ... < f n \) by (simp only: Suc)
      finally have \( f m < f n \).
      with less have \( m \leq f m \).
      also note \( (... < f n) \)
      finally have \( m < f n \).
      then have \( n \leq f n \) by (simp only: Suc)
      then show ?thesis .
    next
      case 0
      then show ?thesis by simp
    qed

\footnote{A question from “Bundeswettbewerb Mathematik”. Original pen-and-paper proof
  due to Herbert Ehler; Isabelle tactic script by Tobias Nipkow.}
qed

have mono: \( m \leq n \implies f \cdot m \leq f \cdot n \) for \( m, n :: \text{nat} \)
proof (induct n)
  case 0
  then have \( m = 0 \) by simp
  then show \(?case\) by simp
next
  case (Suc n)
  from Suc.prems show \( f \cdot m \leq f \cdot (\text{Suc} \cdot n) \)
  proof (rule le-SucE)
    assume \( m \leq n \)
    with Suc.hyps have \( f \cdot m \leq f \cdot n \).
    also from \( \text{ge f-ax} \) have \( \ldots < f \cdot (\text{Suc} \cdot n) \)
    by (rule le-less-trans)
    finally show \(?thesis\) by simp
  next
    assume \( m = \text{Suc} \cdot n \)
    then show \(?thesis\) by simp
qed
qed

show \( f \cdot n \leq n \)
proof =
  have \( \neg \cdot n < f \cdot n \)
  proof
    assume \( n < f \cdot n \)
    then have \( \text{Suc} \cdot n \leq f \cdot n \) by simp
    then have \( f \cdot (\text{Suc} \cdot n) \leq f \cdot (f \cdot n) \) by (rule mono)
    also have \( \ldots < f \cdot (\text{Suc} \cdot n) \) by (rule f-ax)
    finally have \( \ldots < \ldots \cdot \) 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:

$2 \times (\sum i::\text{nat}=0..n. \ i) = n \times (n + 1)$

(is $?P \ n$ is $?S \ n = -$)

**Proof** (induct $n$)

show $?P \ 0$ by simp

next

fix $n$ have $?S \ (n + 1) = ?S \ n + 2 \times (n + 1)$

by simp

also assume $?S \ n = n \times (n + 1)$

also have $\ldots + 2 \times (n + 1) = (n + 1) \times (n + 2)$

by simp

finally show $?P \ (\text{Suc} \ n)$

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 $?P \ n \Rightarrow ?P \ (\text{Suc} \ n)$ 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:**
\[
(\sum i :: \text{nat} = 0..<n. 2 * i + 1) = n \cdot \text{Suc} \ (\text{Suc} \ 0)
\]
(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** \(?S\ n = n \cdot \text{Suc} \ (\text{Suc} \ 0)\)
- **Also have** \(\ldots + 2 * n + 1 = (n + 1) \cdot \text{Suc} \ (\text{Suc} \ 0)\)
  - by simp
- **Finally show** \(?P\ (\text{Suc} \ n)\)
  - 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:**
\[
6 \cdot (\sum i :: \text{nat} = 0..n. i^2) = n \cdot (n + 1)^2 \cdot (2 * n + 1)
\]
(is \(?P\ n\ is\ ?S\ n = -\))

**Proof (induct n)**
- **Show** \(?P\ 0\) by simp

**Next**
- **Fix** \(n\)
- **Have** \(?S\ (n + 1) = ?S\ n + 6 \cdot (n + 1) \cdot \text{Suc} \ (\text{Suc} \ 0)\)
  - by (simp add: distrib)
- **Also assume** \(?S\ n = n \cdot (n + 1)^2 \cdot (2 * n + 1)\)
- **Also have** \(\ldots + 6 \cdot (n + 1) \cdot \text{Suc} \ (\text{Suc} \ 0) = \)
  \[(n + 1) \cdot (n + 2) \cdot (2 \cdot (n + 1) + 1)\]
  - by (simp add: distrib)
- **Finally show** \(?P\ (\text{Suc} \ n)\)
  - by simp

**Qed**

**Theorem sum-of-cubes:**
\[
4 \cdot (\sum i :: \text{nat} = 0..n. i^3) = (n \cdot (n + 1)) \cdot \text{Suc} \ (\text{Suc} \ 0)
\]
(is \(?P\ n\ is\ ?S\ n = -\))

**Proof (induct n)**
- **Show** \(?P\ 0\) by (simp add: power-eq-if)

**Next**
- **Fix** \(n\)
- **Have** \(?S\ (n + 1) = ?S\ n + 4 \cdot (n + 1) \cdot 3\)
  - by (simp add: power-eq-if distrib)
- **Also assume** \(?S\ n = (n \cdot (n + 1)) \cdot \text{Suc} \ (\text{Suc} \ 0)\)
- **Also have** \(\ldots + 4 \cdot (n + 1) \cdot 3 = ((n + 1) \cdot ((n + 1) + 1)) \cdot \text{Suc} \ (\text{Suc} \ 0)\)
  - by (simp add: power-eq-if distrib)
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 ⇒ prop (- 5)

17.2 Propositional logic

axiomatization false :: o (⊥)
  where falseE [elim]: ⊥ ⇒ A

axiomatization imp :: o ⇒ o ⇒ o (infixr → 25)
  where impI [intro]: (A ⇒ B) ⇒ A → B
    and mp [dest]: A → B ⇒ A ⇒ B

axiomatization conj :: o ⇒ o ⇒ o (infixr ∧ 35)
  where conjI [intro]: A ⇒ B ⇒ A ∧ B
    and conjD1: A ∧ B ⇒ A
    and conjD2: A ∧ B ⇒ B

theorem conjE [elim]:
assumes $A \land B$
obreak\hspace{2em} obtains $A$ and $B$

proof

from $(A \land B)$ show $A$
  by (rule conjD1)
from $(A \land B)$ show $B$
  by (rule conjD2)

qed

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

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

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

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

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

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

definition iff :: $o \Rightarrow o \Rightarrow o$ (infixr $\leftrightarrow$ 25)
  where $A \leftrightarrow B \equiv (A \Rightarrow B) \land (B \Rightarrow A)$

theorem iffI [intro]:
  assumes $A \Rightarrow B$
  and $B \Rightarrow A$
  shows $A \leftrightarrow B$
  unfolding iff-def
proof
  from $(A \Rightarrow B)$ show $A \Rightarrow B$ ..
  from $(B \Rightarrow A)$ show $B \Rightarrow A$ ..
theorem iff1 [elim]:
  assumes \( A \leftrightarrow B \) and \( A \)
  shows \( B \)
proof
  from \( A \leftrightarrow B \) have \( (A \rightarrow B) \land (B \rightarrow A) \)
  unfolding iff-def .
  then have \( A \rightarrow B \) ..
  from this and \( \langle A \rangle \) show \( B \) ..
qed

theorem iff2 [elim]:
  assumes \( A \leftrightarrow B \) and \( B \)
  shows \( A \)
proof
  from \( A \leftrightarrow B \) have \( (A \rightarrow B) \land (B \rightarrow A) \)
  unfolding iff-def .
  then have \( B \rightarrow A \) ..
  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: \( x = y \Rightarrow P x \Rightarrow P y \)

theorem trans [trans]: \( x = y \Rightarrow y = z \Rightarrow x = z \)
  by (rule subst)

theorem sym [sym]: \( x = y \Rightarrow y = x \)
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 \( \forall \) 10)
  where allI [intro]: \( \langle \forall x. P x \rangle \Rightarrow \forall x. P x \)
  and allD [dest]: \( \forall x. P x \Rightarrow P a \)

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

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

lemma \((\exists x. \forall y. R x y) \rightarrow (\forall y. \exists x. R x y)\)
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 \( \forall y. R x y \) 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 been adapted to modern presentations of \( \lambda \)-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 \Rightarrow 25)
where \( \text{impI} \) [intro]: \((A \Rightarrow B) \Rightarrow A \rightarrow B\)
and \( \text{impE} \) [dest, trans]: \(A \rightarrow B \Rightarrow A \rightarrow B\)

axiomatization \( \text{All} :: (\forall \ a \Rightarrow o) \Rightarrow o \) (binder \( \forall \ 10 \))
where \( \text{allI} \) [intro]: \((\forall x. P x) \Rightarrow \forall x. P x\)
and \( \text{allE} \) [dest]: \(\forall x. P x \Rightarrow P a\)

lemma \( \text{atomize-imp} \) [atomize]: \((A \Rightarrow B) \equiv \text{Trueprop} (A \rightarrow B)\)
by standard (fact impI, fact impE)

lemma \( \text{atomize-all} \) [atomize]: \((\forall x. P x) \equiv \text{Trueprop} (\forall x. P x)\)
by standard (fact allI, fact allE)

20.0.1 Derived connectives

definition \( \text{False} :: o \)
where \( \text{False} \equiv \forall A. \ A\)

lemma \( \text{FalseE} \) [elim]:
assumes \( \text{False} \)
shows \( A\)
proof –
from \( \langle \text{False} \rangle \) have \( \forall A. \ A\) by \( \text{simp only: False-def} \)
then show \( A\) ..
qed

definition \( \text{True} :: o \)
where \( \text{True} \equiv \text{False} \rightarrow \text{False}\)

lemma \( \text{TrueI} \) [intro]: True
unfolding \( \text{True-def} ..\)

definition \( \text{not} :: o \Rightarrow o \) (\( \neg - \ [40] \ 40 \))
where \( \text{not} \equiv \lambda A. \ A \rightarrow \text{False}\)

lemma \( \text{notI} \) [intro]:
assumes \( \neg A \Rightarrow \text{False} \)
shows \( \neg A\)
using \( \text{assms unfolding not-def} ..\)

lemma \( \text{notE} \) [elim]:
assumes \( \neg A \) and \( A\)
shows \( B\)
proof –
from \( \langle \neg A \rangle\) have \( A \rightarrow \text{False} \) by \( \text{simp only: not-def} \)
from this and \( \langle A \rangle\) have \( \text{False} ..\)
then show \( B\) ..

63
qed

lemma notE': \( A \implies \neg A \implies B \)
by (rule notE)

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

definition conj :: \( o \implies o \implies o \) (infixr \( \land \) 35)
where \( A \land B \equiv \forall C. (A \implies B \implies C) \implies C \)

lemma conjI [intro]:
assumes \( A \) and \( B \)
shows \( A \land B \)
unfolding conj-def
proof
fix \( C \)
show \( (A \implies B \implies C) \implies C \)
proof
assume \( A \implies B \implies C \)
also note (\( A \))
also note (\( B \))
finally show \( C \).
qed

qed

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

show \( B \)
proof –
note * [of \( B \)]
also have \( A \implies B \implies B \)
proof
show \( B \implies B \).. 
qed
finally show \( ?\text{thesis} \).

qed

qed

definition \text{disj} :: \( o \Rightarrow o \Rightarrow o \) (infixr \( \lor \))
where \( A \lor B \equiv \forall C. \ (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C \)

lemma \text{disjI1} [intro]:
  assumes \( A \)
  shows \( A \lor B \)
  unfolding \text{disj-def}
proof
  fix \( C \)
  show \( (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C \)
  proof
    assume \( A \rightarrow C \)
    from this and \( \langle A \rangle \) have \( C \).
    then show \( (B \rightarrow C) \rightarrow C \).
  qed
qed

lemma \text{disjI2} [intro]:
  assumes \( B \)
  shows \( A \lor B \)
  unfolding \text{disj-def}
proof
  fix \( C \)
  show \( (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C \)
  proof
    show \( (B \rightarrow C) \rightarrow C \)
    proof
      assume \( B \rightarrow C \)
      from this and \( \langle B \rangle \) show \( C \).
    qed
  qed
qed

lemma \text{disjE} [elim]:
  assumes \( A \lor B \)
  obtains \( \langle a \rangle \) \( A \) | \( \langle b \rangle \) \( B \)
proof
  from \( \langle A \lor B \rangle \) have \( (A \rightarrow \text{thesis}) \rightarrow (B \rightarrow \text{thesis}) \rightarrow \text{thesis} \)
  unfolding \text{disj-def}.
  also have \( A \rightarrow \text{thesis} \)
  proof
    assume \( A \)
    then show \( \text{thesis} \) by (rule \( a \))
  qed
also have $B \rightarrow \text{thesis}$
proof
  assume $B$
  then show thesis by (rule b)
qed
finally show thesis .
qed

definition $\text{Ex} :: ('a \Rightarrow o) \Rightarrow o$ (binder $\exists$ 10)
where $\exists x. P x \equiv \forall C. (\forall x. P x \rightarrow C) \rightarrow C$

lemma $\text{exI}$ [intro]: $P a \implies \exists x. P x$
  unfolding $\text{Ex-def}$
proof
  fix $C$
  assume $P a$
  show $(\forall x. P x \rightarrow C) \rightarrow C$
  proof
    assume $\forall x. P x \rightarrow C$
    then have $P a \rightarrow C$ ..
    from this and $(P a)$ show $C$ ..
  qed
qed

lemma $\text{exE}$ [elim]:
  assumes $\exists x. P x$
  obtains (that) $x$ where $P x$
proof
  from $(\exists x. P x)$ have $(\forall x. P x \rightarrow \text{thesis}) \rightarrow \text{thesis}$
  unfolding $\text{Ex-def}$ ..
  also have $\forall x. P x \rightarrow \text{thesis}$
  proof
    fix $x$
    show $P x \rightarrow \text{thesis}$
    proof
      assume $P x$
      then show thesis by (rule that)
    qed
  qed
  finally show thesis .
qed

20.0.2 Extensional equality

axiomatization $\text{equal} :: 'a \Rightarrow 'a \Rightarrow o$ (infixl = 50)
where $\text{refl}$ [intro]: $x = x$
and $\text{subst}$: $x = y \implies P x \Rightarrow P y$
abbreviation not-equal :: 'a ⇒ 'a ⇒ o (infixl ≠ 50)
  where x ≠ y ≡ ¬(x = y)

abbreviation iff :: o ⇒ o ⇒ o (infixr ⇔ 25)
  where A ⇔ B ≡ A = B

axiomatization
  where ext [intro]: (∀x. f x = g x) ⇒ f = g
    and iff [intro]: (A ⇒ B) ⇒ (B ⇒ A) ⇒ A ⇔ B

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

lemma [trans]: x = y ⇒ P y ⇒ P x
  by (rule subst) (rule sym)

lemma [trans]: P x ⇒ x = y ⇒ 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 ⇒ y = z ⇒ x = z
  by (rule subst)

lemma iff1 [elim]: A ⇔ B ⇒ A ⇒ B
  by (rule subst)

lemma iff2 [elim]: A ⇔ B ⇒ B ⇒ 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 *: ¬ A ⇔ A
  shows C
proof (rule notE)
  show ¬ A
  proof
    assume A
    with * have ¬ A ..
    from this and ⟨A⟩ show False ..
  qed

67
with * show A ..

qed

**Theorem Cantor:** ¬ (∃ f :: 'a ⇒ 'a ⇒ o. ∀ A. ∃ x. A = f x)

**Proof**

assume ∃ f :: 'a ⇒ 'a ⇒ o. ∀ A. ∃ x. A = f x
then obtain f :: 'a ⇒ 'a ⇒ o where *: ∀ A. ∃ x. A = f x ..
let ?D = λx. ¬ f x x
from * have ∃ x. ?D = f x ..
then obtain a where ?D = f a ..
then have ?D a ←→ f a a using refl by (rule subst)
then have ¬ f a a ←→ f a a .
then show False by (rule iff-contradiction)

**20.2 Characterization of Classical Logic**

The subsequent rules of classical reasoning are all equivalent.

**locale classical =**

assumes classical: (~ A ⇒ A) ⇒ A
— predicate definition and hypothetical context

**begin**

**lemma classical-contradiction:**

assumes ¬ A ⇒ False
shows A

**proof** (rule classical)

assume ¬ A
then have False by (rule assms)
then show A ..

qed

**lemma double-negation:**

assumes ¬¬ A
shows A

**proof** (rule classical-contradiction)

assume ¬¬ A
with (~ ¬ A. show False by (rule contradiction)

qed

**lemma tertium-non-datur:** A ∨ ¬ A

**proof** (rule double-negation)

show ¬ (A ∨ ¬ A)

**proof**

assume ¬ (A ∨ ¬ A)
have ¬ A

**proof**

assume A then have A ∨ ¬ A ..
with (~ (A ∨ ¬ A). show False by (rule contradiction)
then have \( A \lor \neg A \).
with \( \neg (A \lor \neg A) \) show \text{False} by (rule contradiction).

\[
\begin{align*}
\text{lemma classical-cases:} \\
\text{obtains } A \mid \neg A \\
\text{using } \text{tertium-non-datur}
\end{align*}
\]
proof
\begin{align*}
\text{assume } A \\
\text{then show } \text{thesis} .. \\
\text{next} \\
\text{assume } \neg A \\
\text{then show } \text{thesis} ..
\end{align*}
qed

\[
\begin{align*}
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{lemma classical-if-cases: classical} \\
\text{if cases: } \bigwedge A C, (A \implies C) \implies (\neg A \implies C) \implies C
\end{align*}
\]
proof
\begin{align*}
\text{fix } A \\
\text{assume } \ast: \neg A \implies A \\
\text{show } A \\
\text{proof (rule cases)} \\
\quad \text{assume } A \\
\quad \text{then show } A. \\
\text{next} \\
\quad \text{assume } \neg A \\
\quad \text{then show } A \text{ by (rule } \ast) \\
\text{qed}
\end{align*}
qed

\section{21 Peirce’s Law}

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

\[
\begin{align*}
\text{theorem (in classical) Peirce’s-Law: } ((A \implies B) \implies A) \implies A
\end{align*}
\]
proof
\begin{align*}
\text{assume } \ast: (A \implies B) \implies A \\
\text{show } A \\
\text{proof (rule classical)} \\
\quad \text{assume } \neg A \\
\quad \text{have } A \implies B \\
\text{proof} \\
\quad \text{assume } A \\
\quad \text{with } (\neg A) \text{ show } B \text{ by (rule contradiction)}
\end{align*}
\]
qed
with show A ..
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

70
proof
  assume \( ?P = ?Q \)
  then have \( \text{Eps } ?P \iff \text{Eps } ?Q \) by (rule arg-cong)
  also note \( \langle \text{Eps } ?Q \rangle \)
  finally have \( \text{Eps } ?P \).
  with \( \langle \neg \text{Eps } ?P \rangle \) show False by (rule contradiction)
qed
have \( \neg A \)
proof
  assume \( A \)
  have \( ?P = ?Q \)
proof (rule ext)
    show \( ?P x \iff ?Q x \) for \( x \)
    proof
      assume \( ?P x \)
      then show \( ?Q x \)
      proof
        assume \( \neg x \)
        with \( \langle A \rangle \) have \( A \land \neg x \) ..
        then show \( ?\text{thesis} \) ..
      next
        assume \( A \land x \)
        then have \( x \) ..
        then show \( ?\text{thesis} \) ..
      qed
    next
    assume \( ?Q x \)
    then show \( ?P x \)
    proof
      assume \( A \land \neg x \)
      then have \( \neg x \) ..
      then show \( ?\text{thesis} \) ..
    next
    assume \( x \)
    with \( \langle A \rangle \) have \( A \land x \) ..
    then show \( ?\text{thesis} \) ..
    qed
    qed
    qed
    with \( \text{neq} \) show False by (rule contradiction)
    qed
    then show \( ?\text{thesis} \) ..
    qed
    qed
    qed

This means, the hypothetical predicate \textit{classical} always holds unconditionally (with all consequences).

interpretation \textit{classical}
proof (rule classical-if-cases)
fix A C
assume ∗: A ⇒ C
and ∗∗: ¬ A ⇒ C
from Diaconescu [of A] show C
proof
assume A
then show C by (rule ∗)
next
assume ¬ A
then show C by (rule ∗∗)
qed
qed

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

end

References


