

Miscellaneous Isabelle/Isar examples

Makarius Wenzel

With contributions by Gertrud Bauer and Tobias Nipkow

February 17, 2016

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	3
1.1	Prose version	3
1.2	Formal versions	4
2	Peirce’s Law	5
3	The Drinker’s Principle	7
4	Cantor’s Theorem	8
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	Schröder-Bernstein Theorem	10
6	Structured statements within Isar proofs	11
6.1	Introduction steps	11
6.2	If-and-only-if	12
6.3	Elimination and cases	13
6.4	Induction	13
6.5	Suffices-to-show	14

7	Basic logical reasoning	14
7.1	Pure backward reasoning	15
7.2	Variations of backward vs. forward reasoning	17
7.3	A few examples from “Introduction to Isabelle”	19
7.3.1	A propositional proof	19
7.3.2	A quantifier proof	20
7.3.3	Deriving rules in Isabelle	21
8	Correctness of a simple expression compiler	21
8.1	Binary operations	22
8.2	Expressions	22
8.3	Machine	22
8.4	Compiler	23
9	Fib and Gcd commute	26
9.1	Fibonacci numbers	26
9.2	Fib and gcd commute	26
10	Basic group theory	29
10.1	Groups and calculational reasoning	29
10.2	Groups as monoids	31
10.3	More theorems of group theory	32
11	Some algebraic identities derived from group axioms – theory context version	33
12	Some algebraic identities derived from group axioms – proof notepad version	35
13	Hoare Logic	37
13.1	Abstract syntax and semantics	37
13.2	Primitive Hoare rules	38
13.3	Concrete syntax for assertions	40
13.4	Rules for single-step proof	42
13.5	Verification conditions	43
14	Using Hoare Logic	45
14.1	State spaces	45
14.2	Basic examples	45
14.3	Multiplication by addition	47
14.4	Summing natural numbers	48
14.5	Time	49

15 The Mutilated Checker Board Problem	50
15.1 Tilings	51
15.2 Basic properties of “below”	51
15.3 Basic properties of “evnodd”	52
15.4 Dominoes	52
15.5 Tilings of dominoes	54
15.6 Main theorem	55
16 An old chestnut	56
17 Summing natural numbers	58
17.1 Summation laws	58
18 A simple formulation of First-Order Logic	60
18.1 Abstract syntax	60
18.2 Propositional logic	60
18.3 Equality	62
18.4 Quantifiers	63
19 Foundations of HOL	63
19.1 Pure Logic	64
19.1.1 Basic logical connectives	64
19.1.2 Extensional equality	64
19.1.3 Derived connectives	65
19.2 Cantor’s Theorem	68
19.3 Classical logic	69

1 Textbook-style reasoning: the Knaster-Tarski Theorem

```
theory Knaster-Tarski
imports Main ~~/src/HOL/Library/Lattice-Syntax
begin
```

1.1 Prose version

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

The Knaster-Tarski Fixpoint Theorem. Let L be a complete lattice and $f: L \rightarrow L$ 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 $f(a)$

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

$\leq a$. We now use this inequality to prove the reverse one (!) and thereby complete the proof that a is a fixpoint. Since f is order-preserving, $f(f(a)) \leq f(a)$. This says $f(a) \in H$, so $a \leq f(a)$.

1.2 Formal versions

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

theorem *Knaster-Tarski*:

```

fixes  $f :: 'a::complete-lattice \Rightarrow 'a$ 
assumes mono  $f$ 
shows  $\exists a. f\ a = a$ 
proof
  let  $?H = \{u. f\ u \leq u\}$ 
  let  $?a = \bigcap ?H$ 
  show  $f\ ?a = ?a$ 
  proof -
    {
      fix  $x$ 
      assume  $x \in ?H$ 
      then have  $?a \leq x$  by (rule Inf-lower)
      with  $\langle mono\ f \rangle$  have  $f\ ?a \leq f\ x ..$ 
      also from  $\langle x \in ?H \rangle$  have  $\dots \leq x ..$ 
      finally have  $f\ ?a \leq x .$ 
    }
    then have  $f\ ?a \leq ?a$  by (rule Inf-greatest)
    {
      also presume  $\dots \leq f\ ?a$ 
      finally (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  $f :: 'a :: \text{complete-lattice} \Rightarrow 'a$ 
assumes  $\text{mono } f$ 
shows  $\exists a. f\ a = a$ 
proof
  let  $?H = \{u. f\ u \leq u\}$ 
  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  $\langle \text{mono } f \rangle$  have  $f\ ?a \leq f\ x$  ..
      also from  $\langle x \in ?H \rangle$  have  $\dots \leq x$  ..
      finally show  $f\ ?a \leq x$  .
    qed
  show  $?a \leq f\ ?a$ 
  proof (rule Inf-lower)
    from  $\langle \text{mono } f \rangle$  and  $\langle f\ ?a \leq ?a \rangle$  have  $f\ (f\ ?a) \leq f\ ?a$  ..
    then show  $f\ ?a \in ?H$  ..
  qed
qed
qed
end

```

2 Peirce's Law

```

theory Peirce
imports Main
begin

```

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

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

```

theorem  $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$ 
proof
  assume  $(A \longrightarrow B) \longrightarrow A$ 
  show  $A$ 
  proof (rule classical)
    assume  $\neg A$ 

```

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

```

have  $A \longrightarrow B$ 
proof
  assume  $A$ 
  with  $\langle \neg A \rangle$  show  $B$  by contradiction
qed
with  $\langle (A \longrightarrow B) \longrightarrow A \rangle$  show  $A$  ..
qed
qed

```

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

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

```

theorem  $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$ 
proof
  assume  $(A \longrightarrow B) \longrightarrow A$ 
  show  $A$ 
  proof (rule classical)
    presume  $A \longrightarrow B$ 
    with  $\langle (A \longrightarrow B) \longrightarrow A \rangle$  show  $A$  ..
  next
    assume  $\neg A$ 
    show  $A \longrightarrow B$ 
    proof
      assume  $A$ 
      with  $\langle \neg A \rangle$  show  $B$  by contradiction
    qed
  qed
qed

```

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

Nevertheless, the “strong” mode of plain assumptions is quite important in practice to achieve robustness of proof text interpretation. By forcing both the conclusion *and* the assumptions to unify with the pending goal to be solved, goal selection becomes quite deterministic. For example, decomposition with rules of the “case-analysis” type usually gives rise to several goals that only differ in 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.

end

3 The Drinker's Principle

```
theory Drinker
imports Main
begin
```

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

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

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

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

end

4 Cantor's Theorem

```
theory Cantor
imports Main
begin
```

4.1 Mathematical statement and proof

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

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

```
theorem Cantor:  $\neg (\exists f :: 'a \Rightarrow 'a \text{ set}. \forall A. \exists x. A = f x)$ 
proof
  assume  $\exists f :: 'a \Rightarrow 'a \text{ set}. \forall A. \exists x. A = f x$ 
  then obtain  $f :: 'a \Rightarrow 'a \text{ set}$  where  $*: \forall A. \exists x. A = f x ..$ 
  let  $?D = \{x. x \notin f x\}$ 
  from  $*$  obtain  $a$  where  $?D = f a$  by blast
  moreover have  $a \in ?D \longleftrightarrow a \notin f a$  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  $\neg (\exists f :: 'a \Rightarrow 'a \text{ set}. \forall A. \exists x. f x = A)$ 
  by best
```

```
theorem  $\neg (\exists f :: 'a \Rightarrow 'a \text{ set}. \forall A. \exists x. f x = A)$ 
  by force
```

4.3 Elementary version in higher-order predicate logic

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

```
lemma iff-contradiction:
  assumes  $*: \neg A \longleftrightarrow A$ 
```

```

  shows False
proof (rule notE)
  show  $\neg A$ 
  proof
    assume A
    with * have  $\neg A$  ..
    from this and  $\langle A \rangle$  show False ..
  qed
  with * show A ..
qed

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

```

4.4 Classic Isabelle/HOL example

The following treatment of Cantor's Theorem follows the classic example from the early 1990s, e.g. see the file `92/HOL/ex/set.ML` in Isabelle92 or [7, §18.7]. The old tactic scripts synthesize key information of the proof by refinement of schematic goal states. In contrast, the Isar proof needs to say explicitly what is proven.

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

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

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

```

theorem  $\exists S. S \notin range\ (f :: 'a \Rightarrow 'a\ set)$ 
proof
  let ?S =  $\{x. x \notin f\ x\}$ 
  show  $?S \notin range\ f$ 
  proof
    assume  $?S \in range\ f$ 

```

```

then obtain  $y$  where  $?S = f\ y \ ..$ 
then show 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 ?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 ?thesis by contradiction
qed
qed
qed

```

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

```

theorem  $\exists S. S \notin \text{range } (f :: 'a \Rightarrow 'a \text{ set})$ 
  by best

```

end

5 Schröder-Bernstein Theorem

```

theory Schroeder-Bernstein
imports Main
begin

```

See also:

- `$ISABELLE_HOME/src/HOL/ex/Set_Theory.thy`
- <http://planetmath.org/proofofschroederbernsteintheoremusingtarskiknastertheorem>
- Springer LNCS 828 (cover page)

```

theorem Schroeder-Bernstein:
  fixes  $f :: 'a \Rightarrow 'b$ 
  and  $g :: 'b \Rightarrow 'a$ 
  assumes inj  $f$  and inj  $g$ 
  shows  $\exists h :: 'a \Rightarrow 'b. \text{inj } h \wedge \text{surj } h$ 
proof
  def  $A \equiv \text{lfp } (\lambda X. - (g \circ (f \circ X)))$ 
  def  $g' \equiv \text{inv } g$ 

```

```

let ?h =  $\lambda z. \text{if } z \in A \text{ then } f\ z \text{ else } g'\ z$ 

have A =  $\neg (g' \text{ ` } (\neg (f' A)))$ 
  unfolding A-def by (rule lfp-unfold) (blast intro: monoI)
then have A-compl:  $\neg A = g' \text{ ` } (\neg (f' A))$  by blast
then have *:  $g' \text{ ` } (\neg A) = \neg (f' A)$ 
  using g'-def <inj g> by auto

show inj ?h  $\wedge$  surj ?h
proof
  from * show surj ?h by auto
  have inj-on f A
    using <inj f> by (rule subset-inj-on) blast
  moreover
  have inj-on g' ( $\neg A$ )
    unfolding g'-def
  proof (rule inj-on-inv-into)
    have  $g' \text{ ` } (\neg (f' A)) \subseteq \text{range } g$  by blast
    then show  $\neg A \subseteq \text{range } g$  by (simp only: A-compl)
  qed
  moreover
  have False if eq:  $f\ a = g'\ b$  and a:  $a \in A$  and b:  $b \in \neg A$  for a b
  proof -
    from a have fa:  $f\ a \in f' A$  by (rule imageI)
    from b have gb:  $g'\ b \in g' \text{ ` } (\neg A)$  by (rule imageI)
    with * have gb:  $g'\ b \in \neg (f' A)$  by simp
    with eq fa show False by simp
  qed
  ultimately show inj ?h
    unfolding inj-on-def by (metis ComplI)
  qed
qed
end

```

6 Structured statements within Isar proofs

```

theory Structured-Statements
imports Main
begin

```

6.1 Introduction steps

```

notepad
begin
  fix A B :: bool
  fix P :: 'a  $\Rightarrow$  bool

  have A  $\longrightarrow$  B

```

```

proof
  show  $B$  if  $A$  using that  $\langle proof \rangle$ 
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

```

6.2 If-and-only-if

```

notepad
begin
  fix  $A\ B :: bool$ 

  have  $A \longleftrightarrow B$ 
  proof
    show  $B$  if  $A$   $\langle proof \rangle$ 
    show  $A$  if  $B$   $\langle proof \rangle$ 
  qed
next
  fix  $A\ B :: bool$ 

  have iff-comm:  $(A \wedge B) \longleftrightarrow (B \wedge A)$ 
  proof
    show  $B \wedge A$  if  $A \wedge B$ 
    proof
      show  $B$  using that ..
      show  $A$  using that ..
    qed
    show  $A \wedge B$  if  $B \wedge A$ 
    proof
      show  $A$  using that ..
      show  $B$  using that ..
    qed
  qed
qed

```

Alternative proof, avoiding redundant copy of symmetric argument.

```

have iff-comm:  $(A \wedge B) \longleftrightarrow (B \wedge A)$ 
proof
  show  $B \wedge A$  if  $A \wedge B$  for  $A\ B$ 
  proof
    show  $B$  using that ..

```

```

    show  $A$  using that ..
  qed
  then show  $A \wedge B$  if  $B \wedge A$ 
    by this (rule that)
  qed
end

```

6.3 Elimination and cases

```

notepad
begin
  fix  $A\ B\ C\ D :: \text{bool}$ 
  assume *:  $A \vee B \vee C \vee D$ 

  consider (a)  $A$  | (b)  $B$  | (c)  $C$  | (d)  $D$ 
    using * by blast
  then have something
  proof cases
    case a thm  $\langle A \rangle$ 
    then show ?thesis  $\langle \text{proof} \rangle$ 
  next
    case b thm  $\langle B \rangle$ 
    then show ?thesis  $\langle \text{proof} \rangle$ 
  next
    case c thm  $\langle C \rangle$ 
    then show ?thesis  $\langle \text{proof} \rangle$ 
  next
    case d thm  $\langle D \rangle$ 
    then show ?thesis  $\langle \text{proof} \rangle$ 
  qed
next
  fix  $A :: 'a \Rightarrow \text{bool}$ 
  fix  $B :: 'b \Rightarrow 'c \Rightarrow \text{bool}$ 
  assume *:  $(\exists x. A\ x) \vee (\exists y\ z. B\ y\ z)$ 

  consider (a)  $x$  where  $A\ x$  | (b)  $y\ z$  where  $B\ y\ z$ 
    using * by blast
  then have something
  proof cases
    case a thm  $\langle A\ x \rangle$ 
    then show ?thesis  $\langle \text{proof} \rangle$ 
  next
    case b thm  $\langle B\ y\ z \rangle$ 
    then show ?thesis  $\langle \text{proof} \rangle$ 
  qed
end

```

6.4 Induction

```

notepad

```

```

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

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

```

6.5 Suffices-to-show

```

notepad
begin
  fix A B C
  assume r: A ⇒ B ⇒ C

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

  have C
  proof –
    show ?thesis when A x (is ?A) for x :: 'a — abstract x
    using that ⟨proof⟩
    show ?A a — concrete a
    ⟨proof⟩
  qed
end

end

```

7 Basic logical reasoning

```

theory Basic-Logic
imports Main
begin

```

7.1 Pure backward reasoning

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

lemma I : $A \longrightarrow A$

proof

assume A

show A **by** *fact*

qed

lemma K : $A \longrightarrow B \longrightarrow A$

proof

assume A

show $B \longrightarrow A$

proof

show A **by** *fact*

qed

qed

lemma S : $(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow A \longrightarrow C$

proof

assume $A \longrightarrow B \longrightarrow C$

show $(A \longrightarrow B) \longrightarrow A \longrightarrow C$

proof

assume $A \longrightarrow B$

show $A \longrightarrow C$

proof

assume A

show C

proof (*rule mp*)

show $B \longrightarrow C$ **by** (*rule mp*) *fact*+

show B **by** (*rule mp*) *fact*+

qed

qed

qed

qed

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

Concluding any (sub-)proof already involves solving any remaining goals by assumption³. Thus we may skip the rather vacuous body of the above proof.

lemma $A \longrightarrow A$

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

```
proof
qed
```

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

```
lemma  $A \longrightarrow A$ 
by rule
```

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

```
lemma  $A \longrightarrow A$  ..
```

Thus we have arrived at an adequate representation of the proof of a tautology that holds by a single standard rule.⁴

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

The *intro* proof method repeatedly decomposes a goal's conclusion.⁵

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

Again, the body may be collapsed.

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

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

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

⁴Apparently, the rule here is implication introduction.

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

7.2 Variations of backward vs. forward reasoning

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

The first version is purely backward.

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

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

```
lemma  $A \wedge B \longrightarrow B \wedge A$ 
proof
  assume  $A \wedge B$ 
  show  $B \wedge A$ 
  proof
    from  $\langle A \wedge B \rangle$  show  $B$  ..
    from  $\langle A \wedge B \rangle$  show  $A$  ..
  qed
qed
```

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

```
lemma  $A \wedge B \longrightarrow B \wedge A$ 
proof
  assume  $A \wedge B$ 
  then show  $B \wedge A$ 
  proof — rule conjE of  $A \wedge B$ 
    assume  $B \ A$ 
    then show ?thesis .. — rule conjI of  $B \wedge A$ 
  qed
```

qed

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

```
lemma  $A \wedge B \longrightarrow B \wedge A$   
proof  
  assume  $A \wedge B$   
  from  $\langle A \wedge B \rangle$  have  $A$  ..  
  from  $\langle A \wedge B \rangle$  have  $B$  ..  
  from  $\langle B \rangle \langle A \rangle$  show  $B \wedge A$  ..  
qed
```

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

```
lemma  $A \wedge B \longrightarrow B \wedge A$   
proof —  
  {  
    assume  $A \wedge B$   
    from  $\langle A \wedge B \rangle$  have  $A$  ..  
    from  $\langle A \wedge B \rangle$  have  $B$  ..  
    from  $\langle B \rangle \langle A \rangle$  have  $B \wedge A$  ..  
  }  
  then show ?thesis ..      — rule impI  
qed
```

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

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

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

```
lemma  $A \wedge B \longrightarrow B \wedge A$   
proof  
  assume  $A \wedge B$   
  then show  $B \wedge A$   
  proof
```

```

    assume B A
  then show ?thesis ..
qed
qed

```

7.3 A few examples from “Introduction to Isabelle”

We rephrase some of the basic reasoning examples of [6], using HOL rather than FOL.

7.3.1 A propositional proof

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

```

lemma P ∨ P ⟶ P
proof
  assume P ∨ P
  then show P
  proof
    — rule disjE:
    assume P show P by fact
  next
    assume P show P by fact
  qed
qed

```

$$\frac{A \vee B \quad \begin{array}{c} [A] \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ C \end{array}}{C}$$

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 ∨ P ⟶ P
proof
  assume P ∨ P

```

```

then show  $P$ 
proof
  assume  $P$ 
  show  $P$  by fact
  show  $P$  by fact
qed
qed

```

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

```
lemma  $P \vee P \longrightarrow P$ 
proof
  assume  $P \vee P$ 
  then show  $P$  ..
qed
```

7.3.2 A quantifier proof

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

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

$$\begin{array}{lcl}
\text{lemma } (\exists x. P(fx)) \longrightarrow (\exists y. P y) & & \\
\text{proof} & & [A(x)]_x \\
\text{assume } \exists x. P(fx) & & \vdots \\
\text{then show } \exists y. P y & & B \\
\text{proof (rule } exE) & \text{--- rule } exE: & \frac{\exists x. A(x) \quad B}{B} \\
\text{fix } a & & \\
\text{assume } P(f a) \text{ (is } P \text{ ?witness)} & & \\
\text{then show ?thesis by (rule } exI \text{ [of } P \text{ ?witness])} & & \\
\text{qed} & & \\
\text{qed} & &
\end{array}$$

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

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

```

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

```

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

```

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

```

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

7.3.3 Deriving rules in Isabelle

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

```

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

end

```

8 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.

8.1 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 $'val \text{ binop} = 'val \Rightarrow 'val \Rightarrow 'val$

8.2 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* $'val \text{ binop}$ ($'adr$, $'val$) *expr* ($'adr$, $'val$) *expr*

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

primrec *eval* :: ($'adr$, $'val$) *expr* $\Rightarrow$ ($'adr \Rightarrow 'val$) $\Rightarrow$ $'val$
where

eval (*Variable* x) *env* = *env* x
 | *eval* (*Constant* c) *env* = c
 | *eval* (*Binop* f $e1$ $e2$) *env* = f (*eval* $e1$ *env*) (*eval* $e2$ *env*)

8.3 Machine

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

datatype (*dead* $'adr$, *dead* $'val$) *instr* =
 Const $'val$
 | *Load* $'adr$
 | *Apply* $'val \text{ binop}$

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

primrec *exec* :: ($'adr$, $'val$) *instr* *list* $\Rightarrow$ $'val \text{ list}$ $\Rightarrow$ ($'adr \Rightarrow 'val$) $\Rightarrow$ $'val \text{ list}$
where

exec [] *stack env* = *stack*
 | *exec* (*instr* # *instrs*) *stack env* =
 (*case instr of*
 Const $c \Rightarrow$ *exec instrs* (c # *stack*) *env*
 | *Load* $x \Rightarrow$ *exec instrs* (*env* x # *stack*) *env*
 | *Apply* $f \Rightarrow$ *exec instrs* (f (*hd stack*) (*hd* (*tl stack*)))

$\# (tl (tl stack))) env)$

definition *execute* :: (('adr, 'val) instr) list $\Rightarrow$ ('adr $\Rightarrow$ 'val) $\Rightarrow$ 'val
where *execute instrs env* = *hd (exec instrs [] env)*

8.4 Compiler

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

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

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

lemma *exec-append*:
 exec (xs @ ys) stack env =
 exec ys (exec xs stack env) env
proof (*induct xs arbitrary: stack*)
 case *Nil*
 show ?*case* **by** *simp*
next
 case (*Cons x xs*)
 show ?*case*
 proof (*induct x*)
 case *Const*
 from *Cons* **show** ?*case* **by** *simp*
 next
 case *Load*
 from *Cons* **show** ?*case* **by** *simp*
 next
 case *Apply*
 from *Cons* **show** ?*case* **by** *simp*
 qed
qed

theorem *correctness*: *execute (compile e) env* = *eval e env*
proof –
 have $\bigwedge stack. exec (compile e) stack env = eval e env \# stack$
 proof (*induct e*)
 case *Variable*
 show ?*case* **by** *simp*
 next
 case *Constant*
 show ?*case* **by** *simp*
 next

```

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

```

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

lemma *exec-append'*:

exec (xs @ ys) stack env = exec ys (exec xs stack env) env

proof (*induct xs arbitrary: stack*)

case (*Nil s*)

have *exec ([] @ ys) s env = exec ys s env*

by *simp*

also have ... = *exec ys (exec [] s env) env*

by *simp*

finally show ?*case* .

next

case (*Cons x xs s*)

show ?*case*

proof (*induct x*)

case (*Const val*)

have *exec ((Const val # xs) @ ys) s env = exec (Const val # xs @ ys) s env*

by *simp*

also have ... = *exec (xs @ ys) (val # s) env*

by *simp*

also from *Cons* have ... = *exec ys (exec xs (val # s) env) env* .

also have ... = *exec ys (exec (Const val # xs) s env) env*

by *simp*

finally show ?*case* .

next

case (*Load adr*)

from *Cons* show ?*case*

by *simp* — same as above

next

case (*Apply fn*)

have *exec ((Apply fn # xs) @ ys) s env =*

exec (Apply fn # xs @ ys) s env by *simp*

also have ... =

exec (xs @ ys) (fn (hd s) (hd (tl s)) # (tl (tl s))) env

by *simp*

also from *Cons* have ... =

exec ys (exec xs (fn (hd s) (hd (tl s)) # tl (tl s)) env) env .

also have ... = *exec ys (exec (Apply fn # xs) s env) env*

by *simp*

finally show ?*case* .

qed
qed

theorem *correctness'*: $\text{execute } (\text{compile } e) \text{ env} = \text{eval } e \text{ env}$

proof —

have *exec-compile*: $\bigwedge \text{stack}. \text{exec } (\text{compile } e) \text{ stack env} = \text{eval } e \text{ env} \# \text{stack}$

proof (*induct e*)

case (*Variable adr s*)

have $\text{exec } (\text{compile } (\text{Variable } \text{adr})) \text{ s env} = \text{exec } [\text{Load } \text{adr}] \text{ s env}$

by *simp*

also have $\dots = \text{env } \text{adr} \# \text{s}$

by *simp*

also have $\text{env } \text{adr} = \text{eval } (\text{Variable } \text{adr}) \text{ env}$

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 $\text{exec } (\text{compile } (\text{Binop } \text{fn } e1 \text{ } e2)) \text{ s env} =$

$\text{exec } (\text{compile } e2 \text{ @ compile } e1 \text{ @ } [\text{Apply } \text{fn}]) \text{ s env}$

by *simp*

also have $\dots = \text{exec } [\text{Apply } \text{fn}]$

$(\text{exec } (\text{compile } e1) (\text{exec } (\text{compile } e2) \text{ s env}) \text{ env}) \text{ env}$

by (*simp only: exec-append*)

also have $\text{exec } (\text{compile } e2) \text{ s env} = \text{eval } e2 \text{ env} \# \text{s}$

by *fact*

also have $\text{exec } (\text{compile } e1) \dots \text{ env} = \text{eval } e1 \text{ env} \# \dots$

by *fact*

also have $\text{exec } [\text{Apply } \text{fn}] \dots \text{ env} =$

$\text{fn } (\text{hd } \dots) (\text{hd } (\text{tl } \dots)) \# (\text{tl } (\text{tl } \dots))$

by *simp*

also have $\dots = \text{fn } (\text{eval } e1 \text{ env}) (\text{eval } e2 \text{ env}) \# \text{s}$

by *simp*

also have $\text{fn } (\text{eval } e1 \text{ env}) (\text{eval } e2 \text{ env}) =$

$\text{eval } (\text{Binop } \text{fn } e1 \text{ } e2) \text{ env}$

by *simp*

finally show *?case* .

qed

have $\text{execute } (\text{compile } e) \text{ env} = \text{hd } (\text{exec } (\text{compile } e) [] \text{ env})$

by (*simp add: execute-def*)

also from *exec-compile* **have** $\text{exec } (\text{compile } e) [] \text{ env} = [\text{eval } e \text{ env}]$.

also have $\text{hd } \dots = \text{eval } e \text{ env}$

by *simp*

finally show *?thesis* .

qed

end

9 Fib and Gcd commute

```
theory Fibonacci
imports ../Number-Theory/Primes
begin6 declare One-nat-def [simp]
```

9.1 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:
  fixes n :: nat
  shows  $P\ 0 \implies P\ 1 \implies (\bigwedge n. P\ (n + 1) \implies P\ n \implies P\ (n + 2)) \implies P\ n$ 
  by (induct rule: fib.induct) simp-all
```

9.2 Fib and gcd commute

A few laws taken from [3].

```
lemma fib-add:
  fib (n + k + 1) = fib (k + 1) * fib (n + 1) + fib k * fib n
  (is ?P n)
  — see [3, 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)
```

⁶Isar version by Gertrud Bauer. Original tactic script by Larry Paulson. A few proofs of laws taken from [3].

by (*simp add: add-mult-distrib2*)
 finally show $?P (n + 2)$.
 qed

lemma *gcd-fib-Suc-eq-1*: $\text{gcd} (\text{fib } n) (\text{fib } (n + 1)) = 1$ (**is** $?P n$)
proof (*induct n rule: fib-induct*)
 show $?P 0$ **by** *simp*
 show $?P 1$ **by** *simp*
 fix n
 have $\text{fib } (n + 2 + 1) = \text{fib } (n + 1) + \text{fib } (n + 2)$
 by *simp*
 also have $\dots = \text{fib } (n + 2) + \text{fib } (n + 1)$
 by *simp*
 also have $\text{gcd} (\text{fib } (n + 2)) \dots = \text{gcd} (\text{fib } (n + 2)) (\text{fib } (n + 1))$
 by (*rule gcd-add2-nat*)
 also have $\dots = \text{gcd} (\text{fib } (n + 1)) (\text{fib } (n + 1 + 1))$
 by (*simp add: gcd-commute-nat*)
 also assume $\dots = 1$
 finally show $?P (n + 2)$.
 qed

lemma *gcd-mult-add*: $(0::\text{nat}) < n \implies \text{gcd} (n * k + m) n = \text{gcd } m n$
proof –
 assume $0 < n$
 then have $\text{gcd} (n * k + m) n = \text{gcd } n (m \bmod n)$
 by (*simp add: gcd-non-0-nat add.commute*)
 also from $\langle 0 < n \rangle$ have $\dots = \text{gcd } m n$
 by (*simp add: gcd-non-0-nat*)
 finally show $?thesis$.
 qed

lemma *gcd-fib-add*: $\text{gcd} (\text{fib } m) (\text{fib } (n + m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$
proof (*cases m*)
 case 0
 then show $?thesis$ **by** *simp*
 next
 case (*Suc k*)
 then have $\text{gcd} (\text{fib } m) (\text{fib } (n + m)) = \text{gcd} (\text{fib } (n + k + 1)) (\text{fib } (k + 1))$
 by (*simp add: gcd-commute-nat*)
 also have $\text{fib } (n + k + 1)$
 = $\text{fib } (k + 1) * \text{fib } (n + 1) + \text{fib } k * \text{fib } n$
 by (*rule fib-add*)
 also have $\text{gcd} \dots (\text{fib } (k + 1)) = \text{gcd} (\text{fib } k * \text{fib } n) (\text{fib } (k + 1))$
 by (*simp add: gcd-mult-add*)
 also have $\dots = \text{gcd} (\text{fib } n) (\text{fib } (k + 1))$
 by (*simp only: gcd-fib-Suc-eq-1 gcd-mult-cancel-nat*)
 also have $\dots = \text{gcd} (\text{fib } m) (\text{fib } n)$
 using *Suc* **by** (*simp add: gcd-commute-nat*)
 finally show $?thesis$.

qed

lemma gcd-fib-diff:

assumes $m \leq n$

shows $\text{gcd} (\text{fib } m) (\text{fib } (n - m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$

proof -

have $\text{gcd} (\text{fib } m) (\text{fib } (n - m)) = \text{gcd} (\text{fib } m) (\text{fib } (n - m + m))$

by (simp add: gcd-fib-add)

also from $\langle m \leq n \rangle$ have $n - m + m = n$

by simp

finally show ?thesis .

qed

lemma gcd-fib-mod:

assumes $0 < m$

shows $\text{gcd} (\text{fib } m) (\text{fib } (n \bmod m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$

proof (induct n rule: nat-less-induct)

case (1 n) note hyp = this

show ?case

proof -

have $n \bmod m = (\text{if } n < m \text{ then } n \text{ else } (n - m) \bmod m)$

by (rule mod-if)

also have $\text{gcd} (\text{fib } m) (\text{fib } \dots) = \text{gcd} (\text{fib } m) (\text{fib } n)$

proof (cases $n < m$)

case True

then show ?thesis by simp

next

case False

then have $m \leq n$ by simp

from $\langle 0 < m \rangle$ and False have $n - m < n$

by simp

with hyp have $\text{gcd} (\text{fib } m) (\text{fib } ((n - m) \bmod m))$

$= \text{gcd} (\text{fib } m) (\text{fib } (n - m))$ by simp

also have $\dots = \text{gcd} (\text{fib } m) (\text{fib } n)$

using $\langle m \leq n \rangle$ by (rule gcd-fib-diff)

finally have $\text{gcd} (\text{fib } m) (\text{fib } ((n - m) \bmod m)) =$

$\text{gcd} (\text{fib } m) (\text{fib } n)$.

with False show ?thesis by simp

qed

finally show ?thesis .

qed

qed

theorem fib-gcd: $\text{fib} (\text{gcd } m \ n) = \text{gcd} (\text{fib } m) (\text{fib } n)$ (is ?P m n)

proof (induct m n rule: gcd-nat-induct)

fix m

show $\text{fib} (\text{gcd } m \ 0) = \text{gcd} (\text{fib } m) (\text{fib } 0)$

by simp

fix n :: nat

```

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

end

```

10 Basic group theory

```

theory Group
imports Main
begin

```

10.1 Groups and calculational reasoning

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

```

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

```

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

```

theorem (in group) group-right-inverse:  $x * \text{inverse } x = 1$ 

```

proof –

```

  have  $x * \text{inverse } x = 1 * (x * \text{inverse } x)$ 
    by (simp only: group-left-one)
  also have  $\dots = 1 * x * \text{inverse } x$ 
    by (simp only: group-assoc)
  also have  $\dots = \text{inverse } (\text{inverse } x) * \text{inverse } x * x * \text{inverse } x$ 
    by (simp only: group-left-inverse)
  also have  $\dots = \text{inverse } (\text{inverse } x) * (\text{inverse } x * x) * \text{inverse } x$ 
    by (simp only: group-assoc)
  also have  $\dots = \text{inverse } (\text{inverse } x) * 1 * \text{inverse } x$ 
    by (simp only: group-left-inverse)
  also have  $\dots = \text{inverse } (\text{inverse } x) * (1 * \text{inverse } x)$ 
    by (simp only: group-assoc)
  also have  $\dots = \text{inverse } (\text{inverse } x) * \text{inverse } x$ 
    by (simp only: group-left-one)
  also have  $\dots = 1$ 

```

```

    by (simp only: group-left-inverse)
  finally show ?thesis .
qed

```

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

```

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

```

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

Note that “...” is just a special term variable that is bound automatically to the argument⁷ of the last fact achieved by any local assumption or proven statement. In contrast to *?thesis*, the “...” variable is bound *after* the proof is finished.

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

```

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

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

  have  $\dots = x * inverse\ x * x$ 
    by (simp only: group-assoc)

```

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

```

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

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

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

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

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

show ?thesis .
qed

```

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

10.2 Groups as monoids

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

```

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

```

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

```

instance group  $\subseteq$  monoid
  by intro-classes
    (rule group-assoc,
     rule group-left-one,
     rule group-right-one)

```

The **instance** command actually is a version of **theorem**, setting up a goal that reflects the intended class relation (or type constructor arity). Thus

any Isar proof language element may be involved to establish this statement. When concluding the proof, the result is transformed into the intended type signature extension behind the scenes.

10.3 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 (*simp only: group-right-inverse*)

also have $\dots = (e * x) * \text{inverse } x$

by (*simp only: eq*)

also have $\dots = e * (x * \text{inverse } x)$

by (*simp only: group-assoc*)

also have $\dots = e * 1$

by (*simp only: group-right-inverse*)

also have $\dots = e$

by (*simp only: group-right-one*)

finally show *?thesis* .

qed

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

theorem (in *group*) *group-inverse-equality*:

assumes *eq*: $x' * x = 1$

shows $\text{inverse } x = x'$

proof –

have $\text{inverse } x = 1 * \text{inverse } x$

by (*simp only: group-left-one*)

also have $\dots = (x' * x) * \text{inverse } x$

by (*simp only: eq*)

also have $\dots = x' * (x * \text{inverse } x)$

by (*simp only: group-assoc*)

also have $\dots = x' * 1$

by (*simp only: group-right-inverse*)

also have $\dots = x'$

by (*simp only: group-right-one*)

finally show *?thesis* .

qed

The inverse operation has some further characteristic properties.

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

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 (simp only: group-assoc)
  also have  $\dots = (\text{inverse } y * 1) * y$ 
    by (simp only: group-left-inverse)
  also have  $\dots = \text{inverse } y * y$ 
    by (simp only: group-right-one)
  also have  $\dots = 1$ 
    by (simp only: group-left-inverse)
  finally show ?thesis .
qed
qed

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

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

end

```

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

```

theory Group-Context
imports Main
begin

hypothetical group axiomatization

```

```

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

some consequences

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

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

lemma one-equality:
  assumes  $eq: e \odot x = x$ 
  shows  $one = e$ 
proof –
  have  $one = x \odot inverse\ x$ 

```

```

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

```

```

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

```

end

end

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

```

theory Group-Notepad
imports Main
begin

```

```

notepad
begin

```

hypothetical group axiomatization

```

fix prod :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl  $\odot$  70)
and one :: 'a
and inverse :: 'a  $\Rightarrow$  'a
assume assoc: (x  $\odot$  y)  $\odot$  z = x  $\odot$  (y  $\odot$  z)

```

and *left-one*: $one \odot x = x$
and *left-inverse*: $inverse\ x \odot x = one$
for $x\ y\ z$

some consequences

have *right-inverse*: $x \odot inverse\ x = one$ **for** x
proof –
have $x \odot inverse\ x = one \odot (x \odot inverse\ x)$
by (*simp only: left-one*)
also have $\dots = one \odot x \odot inverse\ x$
by (*simp only: assoc*)
also have $\dots = inverse\ (inverse\ x) \odot inverse\ x \odot x \odot inverse\ x$
by (*simp only: left-inverse*)
also have $\dots = inverse\ (inverse\ x) \odot (inverse\ x \odot x) \odot inverse\ x$
by (*simp only: assoc*)
also have $\dots = inverse\ (inverse\ x) \odot one \odot inverse\ x$
by (*simp only: left-inverse*)
also have $\dots = inverse\ (inverse\ x) \odot (one \odot inverse\ x)$
by (*simp only: assoc*)
also have $\dots = inverse\ (inverse\ x) \odot inverse\ x$
by (*simp only: left-one*)
also have $\dots = one$
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 $\dots = x \odot inverse\ x \odot x$
by (*simp only: assoc*)
also have $\dots = one \odot x$
by (*simp only: right-inverse*)
also have $\dots = x$
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 $\dots = (e \odot x) \odot inverse\ x$
by (*simp only: eq*)
also have $\dots = e \odot (x \odot inverse\ x)$
by (*simp only: assoc*)
also have $\dots = e \odot one$
by (*simp only: right-inverse*)

```

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

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

end

end

```

13 Hoare Logic

```

theory Hoare
imports Main
begin

```

13.1 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 [11, §6]. See also `~~/src/HOL/Hoare` and [5].

```

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

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

abbreviation Skip (SKIP)
  where SKIP  $\equiv$  Basic id

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

```

primrec *iter* :: *nat* $\Rightarrow$ '*a bexp* $\Rightarrow$ '*a sem* $\Rightarrow$ '*a sem*

where

iter 0 *b S s s'* $\longleftrightarrow s \notin b \wedge s = s'$
| *iter* (*Suc n*) *b S s s'* $\longleftrightarrow s \in b \wedge (\exists s''. S s s'' \wedge \textit{iter } n \textit{ b S s'' s'})$

primrec *Sem* :: '*a com* $\Rightarrow$ '*a sem*

where

Sem (*Basic f*) *s s'* $\longleftrightarrow s' = f s$
| *Sem* (*c1; c2*) *s s'* $\longleftrightarrow (\exists s''. \textit{Sem } c1 \textit{ s s''} \wedge \textit{Sem } c2 \textit{ s'' s'})$
| *Sem* (*Cond b c1 c2*) *s s'* $\longleftrightarrow$
 (*if s* $\in$ *b* *then Sem c1 s s'* *else Sem c2 s s'*)
| *Sem* (*While b x c*) *s s'* $\longleftrightarrow (\exists n. \textit{iter } n \textit{ b (Sem c) s s'})$

definition *Valid* :: '*a bexp* $\Rightarrow$ '*a com* $\Rightarrow$ '*a bexp* $\Rightarrow$ *bool*

(($\exists \vdash$ -/ ($\exists$ -)/ -) [100, 55, 100] 50)

where $\vdash P \textit{ c Q} \longleftrightarrow (\forall s s'. \textit{Sem } c \textit{ s s'} \longrightarrow s \in P \longrightarrow s' \in Q)$

lemma *ValidI* [*intro?*]:

($\bigwedge s s'. \textit{Sem } c \textit{ s s'} \Longrightarrow s \in P \Longrightarrow s' \in Q$) $\Longrightarrow \vdash P \textit{ c Q}$

by (*simp add: Valid-def*)

lemma *ValidD* [*dest?*]:

$\vdash P \textit{ c Q} \Longrightarrow \textit{Sem } c \textit{ s s'} \Longrightarrow s \in P \Longrightarrow s' \in Q$

by (*simp add: Valid-def*)

13.2 Primitive Hoare rules

From the semantics defined above, we derive the standard set of primitive Hoare rules; e.g. see [11, §6]. Usually, variant forms of these rules are applied in actual proof, see also §13.4 and §13.5.

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 §13.4.

theorem *basic*: $\vdash \{s. f s \in P\} (\textit{Basic } f) P$

proof

fix *s s'*

assume *s*: $s \in \{s. f s \in P\}$

assume *Sem* (*Basic f*) *s s'*

then have $s' = f s$ **by** *simp*

with *s* **show** $s' \in P$ **by** *simp*

qed

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

theorem *seq*: $\vdash P \textit{ c1 Q} \Longrightarrow \vdash Q \textit{ c2 R} \Longrightarrow \vdash P (\textit{c1; c2}) R$

proof

assume *cmd1*: $\vdash P \textit{ c1 Q}$ **and** *cmd2*: $\vdash Q \textit{ c2 R}$

```

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  $sem: Sem\ c\ s\ s'$ 
  assume  $s : P'$  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  $\vdash P\ (Cond\ b\ c1\ c2)\ Q$ 
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  $s : P \cap \neg b$  by simp
    qed
  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  $\vdash P \ (While \ b \ X \ c) \ (P \cap \neg b)$ 
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  $b: s \in b$  and  $sem: Sem \ c \ s \ s''$ 
  and  $iter: iter \ n \ b \ (Sem \ c) \ s'' \ s'$  by auto
  from Suc and  $b$  have  $s \in P \cap b$  by simp
  with body sem have  $s'' \in P$  ..
  with iter show ?case by (rule Suc)
qed
qed

```

13.3 Concrete syntax for assertions

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

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

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

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

```

syntax
  -quote :: ' $b \Rightarrow (a \Rightarrow b)$ '
  -antiquote :: ' $(a \Rightarrow b) \Rightarrow b \ (- [1000] \ 1000)$ '
  -Subst :: ' $a \ bexp \Rightarrow b \Rightarrow idt \Rightarrow a \ bexp \ (-['/''] [1000] \ 999)$ '

```

```

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

```

translations

```

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

13.4 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 introduced above.

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

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

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

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

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

Identity and basic assignments.⁸

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

⁸The *hoare* method introduced in §13.5 is able to provide proper instances for any number of basic assignments, without producing additional verification conditions.

Note that above formulation of assignment corresponds to our preferred way to model state spaces, using (extensible) record types in HOL [4]. For any record field x , Isabelle/HOL provides a functions x (selector) and $x\text{-update}$ (update). Above, there is only a place-holder appearing for the latter kind of function: due to concrete syntax $'x := 'a$ also contains $x\text{-update}$.⁹

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

lemmas $[trans, intro?] = seq$

lemma $seq\text{-assoc}$ $[simp]: \vdash P\ c1;(c2;c3)\ Q \longleftrightarrow \vdash P\ (c1;c2);c3\ Q$
by $(auto\ simp\ add: Valid\text{-}def)$

Conditional statements.

lemmas $[trans, intro?] = cond$

lemma $[trans, intro?]:$
 $\vdash \llbracket 'P \wedge 'b \rrbracket\ c1\ Q$
 $\implies \vdash \llbracket 'P \wedge \neg 'b \rrbracket\ c2\ Q$
 $\implies \vdash \llbracket 'P \rrbracket\ IF\ 'b\ THEN\ c1\ ELSE\ c2\ FI\ Q$
by $(rule\ cond)\ (simp\text{-}all\ add: Valid\text{-}def)$

While statements — with optional invariant.

lemma $[intro?]: \vdash (P \cap b)\ c\ P \implies \vdash P\ (While\ b\ P\ c)\ (P \cap \neg b)$
by $(rule\ while)$

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

lemma $[intro?]:$
 $\vdash \llbracket 'P \wedge 'b \rrbracket\ c\ \llbracket 'P \rrbracket$
 $\implies \vdash \llbracket 'P \rrbracket\ WHILE\ 'b\ INV\ \llbracket 'P \rrbracket\ DO\ c\ OD\ \llbracket 'P \wedge \neg 'b \rrbracket$
by $(simp\ add: while\ Collect\text{-}conj\text{-}eq\ Collect\text{-}neg\text{-}eq)$

lemma $[intro?]:$
 $\vdash \llbracket 'P \wedge 'b \rrbracket\ c\ \llbracket 'P \rrbracket$
 $\implies \vdash \llbracket 'P \rrbracket\ WHILE\ 'b\ DO\ c\ OD\ \llbracket 'P \wedge \neg 'b \rrbracket$
by $(simp\ add: while\ Collect\text{-}conj\text{-}eq\ Collect\text{-}neg\text{-}eq)$

13.5 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

⁹Note that due to the external nature of HOL record fields, we could not even state a general theorem relating selector and update functions (if this were required here); this would only work for any particular instance of record fields introduced so far.

far as we are concerned here, the result is a proof method *hoare*, which may be applied to a Hoare Logic assertion to extract purely logical verification conditions. It is important to note that the method requires **WHILE** loops to be fully annotated with invariants beforehand. Furthermore, only *concrete* pieces of code are handled — the underlying tactic fails ungracefully if supplied with meta-variables or parameters, for example.

lemma *SkipRule*: $p \subseteq q \implies \text{Valid } p \text{ (Basic id) } q$
by (*auto simp add: Valid-def*)

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

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

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

lemma *iter-aux*:
 $\forall s \ s'. \text{Sem } c \ s \ s' \longrightarrow s \in I \wedge s \in b \longrightarrow s' \in I \implies$
 $(\bigwedge s \ s'. s \in I \implies \text{iter } n \ b \ (\text{Sem } c) \ s \ s' \implies s' \in I \wedge s' \notin b)$
by (*induct n*) *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 (*clarsimp simp: Valid-def*)
apply (*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 $\sim\sim/\text{src}/\text{HOL}/\text{Hoare}/\text{hoare-tac.ML}$

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

verification condition generator for Hoare logic

end

14 Using Hoare Logic

```
theory Hoare-Ex
imports Hoare
begin
```

14.1 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.

14.2 Basic examples

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

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

```
lemma  $\vdash \llbracket (N\text{-update } (\lambda\cdot. (2 * 'N))) \rrbracket \in \llbracket 'N = 10 \rrbracket \rrbracket 'N := 2 * 'N \llbracket 'N = 10 \rrbracket$ 
by (rule assign)
```

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

```
lemma  $\vdash \llbracket \text{True} \rrbracket 'N := 10 \llbracket 'N = 10 \rrbracket$ 
by hoare
```

```
lemma  $\vdash \llbracket 2 * 'N = 10 \rrbracket 'N := 2 * 'N \llbracket 'N = 10 \rrbracket$ 
by hoare
```

```
lemma  $\vdash \llbracket 'N = 5 \rrbracket 'N := 2 * 'N \llbracket 'N = 10 \rrbracket$ 
by hoare simp
```

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

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

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

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

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

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

lemma $\vdash \{\!| \text{'}N = a \!\} \text{'}N := \text{'}N \{\!| \text{'}N = a \!\}$
by *hoare*

lemma $\vdash \{\!| \text{'}x = a \!\} \text{'}x := \text{'}x \{\!| \text{'}x = a \!\}$
 $\vdots$

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
 $\vdots$

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

lemma $\vdash \{\!| \text{'}M = \text{'}N \!\} \text{'}M := \text{'}M + 1 \{\!| \text{'}M \neq \text{'}N \!\}$

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

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

14.3 Multiplication by addition

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

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

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

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

lemma
 $\vdash \{\!| M = 0 \wedge S = 0 |\!\}$
 $\text{ WHILE } M \neq a$
 $\text{ INV } \{\!| S = M * b |\!\}$

$DO \text{ 'S} := \text{'S} + b; \text{'M} := \text{'M} + 1 \text{ } OD$
 $\{\text{'S} = a * b\}$
by *hoare auto*

14.4 Summing natural numbers

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

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

theorem

$\vdash \{\text{True}\}$
 $\text{'S} := 0; \text{'I} := 1;$
 $WHILE \text{'I} \neq n$
 DO
 $\text{'S} := \text{'S} + \text{'I};$
 $\text{'I} := \text{'I} + 1$
 OD
 $\{\text{'S} = (\sum_{j < n} j)\}$
(is $\vdash - (-; ?while) -)$

proof $-$

let $?sum = \lambda k :: nat. \sum_{j < k} j$
let $?inv = \lambda s \ i :: nat. s = ?sum \ i$

have $\vdash \{\text{True}\} \text{'S} := 0; \text{'I} := 1 \ \{\text{'S} = 0 \wedge \text{'I} = 1\}$

proof $-$

have $\text{True} \longrightarrow 0 = ?sum \ 1$

by *simp*

also have $\vdash \{\dots\} \text{'S} := 0; \text{'I} := 1 \ \{\text{'S} = 0 \wedge \text{'I} = 1\}$

by *hoare*

finally show $?thesis$.

qed

also have $\vdash \dots \ ?while \ \{\text{'S} = 0 \wedge \text{'I} = 1 \wedge \neg \text{'I} \neq n\}$

proof

let $?body = \text{'S} := \text{'S} + \text{'I}; \text{'I} := \text{'I} + 1$

have $?inv \ s \ i \wedge i \neq n \longrightarrow ?inv \ (s + i) \ (i + 1) \text{ for } s \ i$

by *simp*

also have $\vdash \{\text{'S} + \text{'I} = ?sum \ (\text{'I} + 1)\} \ ?body \ \{\text{'S} = ?sum \ (\text{'I} + 1) \wedge \text{'I} = \text{'I} + 1\}$

by *hoare*

finally show $\vdash \{\text{'S} = 0 \wedge \text{'I} = 1 \wedge \neg \text{'I} \neq n\} \ ?body \ \{\text{'S} = 0 \wedge \text{'I} = 1 \wedge \neg \text{'I} \neq n\}$.

qed

also have $s = ?sum \ i \wedge i \neq n \longrightarrow s = ?sum \ n \text{ for } s \ i$

by *simp*

finally show $?thesis$.

qed

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

theorem

```

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

```

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

theorem

```

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

```

14.5 Time

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

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

type-synonym 'a *time* = (time :: *nat*, ... :: 'a)

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

where

$timeit\ (Basic\ f) = (Basic\ f;\ Basic(\lambda s. s(time := Suc\ (time\ s))))$
 $| timeit\ (c1;\ c2) = (timeit\ c1;\ timeit\ c2)$
 $| timeit\ (Cond\ b\ c1\ c2) = Cond\ b\ (timeit\ c1)\ (timeit\ c2)$
 $| timeit\ (While\ b\ iv\ c) = While\ b\ iv\ (timeit\ c)$

record $tvars = tstate +$

$I :: nat$

$J :: nat$

lemma $lem: (0::nat) < n \implies n + n \leq Suc\ (n * n)$

by $(induct\ n)\ simp-all$

lemma

$\vdash \{i = 'I \wedge 'time = 0\}$
 $(timeit$
 $\ (WHILE\ 'I \neq 0$
 $\ \ \ \ \ \ INV\ \{2 * 'time + 'I * 'I + 5 * 'I = i * i + 5 * i\}$
 $\ \ \ \ \ \ DO$
 $\ \ \ \ \ \ \ 'J := 'I;$
 $\ \ \ \ \ \ \ WHILE\ 'J \neq 0$
 $\ \ \ \ \ \ \ \ \ \ \ \ INV\ \{0 < 'I \wedge 2 * 'time + 'I * 'I + 3 * 'I + 2 * 'J - 2 = i * i +$
 $\ 5 * i\}$
 $\ \ \ \ \ \ \ \ \ \ \ \ DO\ 'J := 'J - 1\ OD;$
 $\ \ \ \ \ \ \ \ \ \ \ \ 'I := 'I - 1$
 $\ \ \ \ \ \ \ \ \ \ \ \ OD))$
 $\ \ \ \ \ \ \ \ \ \ \ \ \{2 * 'time = i * i + 5 * i\}$
apply $simp$
apply $hoare$
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ simp$
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ clarsimp$
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ clarsimp$
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ arith$
prefer 2
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ clarsimp$
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ (clarsimp\ simp:\ nat-distrib)$
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ (frule\ lem)$
 $\ \ \ \ \ \ \ \ \ \ \ \ \mathbf{apply}\ arith$
done

end

15 The Mutilated Checker Board Problem

theory *Mutilated-Checkerboard*

imports *Main*

begin

The Mutilated Checker Board Problem, formalized inductively. See [8] for

the original tactic script version.

15.1 Tilings

```

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

```

The union of two disjoint tilings is a tiling.

```

lemma tiling-Un:
  assumes  $t \in \text{tiling } A$ 
  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 (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  $\langle a \subseteq - t \rangle$  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 ?thesis .
    qed
  qed
qed

```

15.2 Basic properties of “below”

```

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

```

```

lemma below-less-iff [iff]:  $i \in \text{below } k \longleftrightarrow i < k$ 
  by (simp add: below-def)

```

lemma *below-0*: $\text{below } 0 = \{\}$
by (*simp add: below-def*)

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

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

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

15.3 Basic properties of “evnodd”

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

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

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

lemma *evnoddD*: $x \in \text{evnodd } A \ b \implies x \in A$
by (*rule subsetD*) (*rule evnodd-subset*)

lemma *evnodd-finite*: $\text{finite } A \implies \text{finite } (\text{evnodd } A \ b)$
by (*rule finite-subset*) (*rule evnodd-subset*)

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

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

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

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

15.4 Dominoes

inductive-set *domino* :: $(\text{nat} \times \text{nat}) \text{ set set}$
where

$horiz: \{(i, j), (i, j + 1)\} \in domino$
 $| vertl: \{(i, j), (i + 1, j)\} \in domino$

lemma *dominoes-tile-row*:

$\{i\} \times below\ (2 * n) \in tiling\ domino$
 (is $?B\ n \in ?T$)

proof (*induct n*)

case 0

show $?case$ by (*simp add: below-0 tiling.empty*)

next

case (*Suc n*)

let $?a = \{i\} \times \{2 * n + 1\} \cup \{i\} \times \{2 * n\}$

have $?B\ (Suc\ n) = ?a \cup ?B\ n$

by (*auto simp add: Sigma-Suc Un-assoc*)

also have $\dots \in ?T$

proof (*rule tiling.Un*)

have $\{(i, 2 * n), (i, 2 * n + 1)\} \in domino$

by (*rule domino.horiz*)

also have $\{(i, 2 * n), (i, 2 * n + 1)\} = ?a$ by *blast*

finally show $\dots \in domino$.

show $?B\ n \in ?T$ by (*rule Suc*)

show $?a \subseteq -\ ?B\ n$ by *blast*

qed

finally show $?case$.

qed

lemma *dominoes-tile-matrix*:

$below\ m \times below\ (2 * n) \in tiling\ domino$

(is $?B\ m \in ?T$)

proof (*induct m*)

case 0

show $?case$ by (*simp add: below-0 tiling.empty*)

next

case (*Suc m*)

let $?t = \{m\} \times below\ (2 * n)$

have $?B\ (Suc\ m) = ?t \cup ?B\ m$ by (*simp add: Sigma-Suc*)

also have $\dots \in ?T$

proof (*rule tiling-Un*)

show $?t \in ?T$ by (*rule dominoes-tile-row*)

show $?B\ m \in ?T$ by (*rule Suc*)

show $?t \cap ?B\ m = \{\}$ by *blast*

qed

finally show $?case$.

qed

lemma *domino-singleton*:

assumes $d \in domino$

and $b < 2$

shows $\exists i\ j. evnodd\ d\ b = \{(i, j)\}$ (is $?P\ d$)

```

  using assms
proof induct
  from  $\langle b < 2 \rangle$  have b-cases:  $b = 0 \vee b = 1$  by arith
  fix i j
  note [simp] = evnodd-empty evnodd-insert mod-Suc
  from b-cases show  $?P \{(i, j), (i, j + 1)\}$  by rule auto
  from b-cases show  $?P \{(i, j), (i + 1, j)\}$  by rule auto
qed

```

```

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

```

15.5 Tilings of dominoes

```

lemma tiling-domino-finite:
  assumes  $t: t \in \text{tiling domino}$  (is  $t \in ?T$ )
  shows finite t (is  $?F t$ )
  using t
proof induct
  show  $?F \{\}$  by (rule finite.emptyI)
  fix  $a t$  assume  $?F t$ 
  assume  $a \in \text{domino}$ 
  then have  $?F a$  by (rule domino-finite)
  from this and  $\langle ?F t \rangle$  show  $?F (a \cup t)$  by (rule finite-UnI)
qed

```

```

lemma tiling-domino-01:
  assumes  $t: t \in \text{tiling domino}$  (is  $t \in ?T$ )
  shows  $\text{card } (\text{evnodd } t \ 0) = \text{card } (\text{evnodd } t \ 1)$ 
  using t
proof induct
  case empty
  show  $?case$  by (simp add: evnodd-def)
next
  case ( $Un \ a \ t$ )
  let  $?e = \text{evnodd}$ 
  note hyp =  $\langle \text{card } (?e \ t \ 0) = \text{card } (?e \ t \ 1) \rangle$ 
  and at =  $\langle a \subseteq - \ t \rangle$ 
  have card-suc:  $\text{card } (?e \ (a \cup t) \ b) = \text{Suc } (\text{card } (?e \ t \ b))$  if  $b < 2$  for  $b :: \text{nat}$ 
proof -
  have  $?e \ (a \cup t) \ b = ?e \ a \ b \cup ?e \ t \ b$  by (rule evnodd-Un)
  also obtain  $i \ j$  where  $e: ?e \ a \ b = \{(i, j)\}$ 

```

```

proof –
  from  $\langle a \in \text{domino} \rangle$  and  $\langle b < 2 \rangle$ 
  have  $\exists i j. ?e\ a\ b = \{(i, j)\}$  by (rule domino-singleton)
  then show ?thesis by (blast intro: that)
qed
also have  $\dots \cup ?e\ t\ b = \text{insert}\ (i, j)\ (?e\ t\ b)$  by simp
also have  $\text{card}\ \dots = \text{Suc}\ (\text{card}\ (?e\ t\ b))$ 
proof (rule card-insert-disjoint)
  from  $\langle t \in \text{tiling domino} \rangle$  have finite  $t$ 
  by (rule tiling-domino-finite)
  then show finite  $(?e\ t\ b)$ 
  by (rule evnodd-finite)
  from  $e$  have  $(i, j) \in ?e\ a\ b$  by simp
  with at show  $(i, j) \notin ?e\ t\ b$  by (blast dest: evnoddD)
qed
finally show ?thesis .
qed
then have  $\text{card}\ (?e\ (a \cup t)\ 0) = \text{Suc}\ (\text{card}\ (?e\ t\ 0))$  by simp
also from hyp have  $\text{card}\ (?e\ t\ 0) = \text{card}\ (?e\ t\ 1)$  .
also from card-suc have  $\text{Suc}\ \dots = \text{card}\ (?e\ (a \cup t)\ 1)$ 
by simp
finally show ?case .
qed

```

15.6 Main theorem

definition mutilated-board :: $\text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})\ \text{set}$

where

```

mutilated-board  $m\ n =$ 
  below  $(2 * (m + 1)) \times \text{below}\ (2 * (n + 1))$ 
  –  $\{(0, 0)\} - \{(2 * m + 1, 2 * n + 1)\}$ 

```

theorem mutil-not-tiling: $\text{mutilated-board}\ m\ n \notin \text{tiling domino}$

proof (unfold mutilated-board-def)

let ? $T = \text{tiling domino}$

let ? $t = \text{below}\ (2 * (m + 1)) \times \text{below}\ (2 * (n + 1))$

let ? $t' = ?t - \{(0, 0)\}$

let ? $t'' = ?t' - \{(2 * m + 1, 2 * n + 1)\}$

show ? $t'' \notin ?T$

proof

have $t: ?t \in ?T$ **by** (rule dominoes-tile-matrix)

assume $t'': ?t'' \in ?T$

let ? $e = \text{evnodd}$

have fin: finite $(?e\ ?t\ 0)$

by (rule evnodd-finite, rule tiling-domino-finite, rule t)

note [simp] = evnodd-iff evnodd-empty evnodd-insert evnodd-Diff

```

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

```

16 An old chestnut

```

theory Puzzle
imports Main
begin10

```

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

```

theorem
  assumes f-ax:  $\bigwedge n. f(f\ n) < f(\text{Suc } n)$ 
  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  $n$ )

```

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

```

    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  $\dots < f n$  by (simp only: Suc)
    finally have  $f m < f n$  .
    with less have  $m \leq f m$  .
    also note  $\langle \dots < f n \rangle$ 
    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
qed

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

show  $f n \leq n$ 
proof -
  have  $\neg n < f n$ 
  proof
    assume  $n < f n$ 
    then have  $Suc n \leq f n$  by simp
    then have  $f (Suc n) \leq f (f n)$  by (rule mono)
    also have  $\dots < f (Suc n)$  by (rule f-ax)
    finally have  $\dots < \dots$  . then show False ..
  qed
  then show ?thesis by simp
qed
qed

```

end

17 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.

17.1 Summation laws

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

```
theorem sum-of-naturals:
  2 * ( $\sum i::nat=0..n. i$ ) =  $n * (n + 1)$ 
  (is ?P n is ?S n = -)
proof (induct n)
  show ?P 0 by simp
next
  fix n have ?S (n + 1) = ?S n + 2 * (n + 1)
    by simp
  also assume ?S n =  $n * (n + 1)$ 
  also have  $\dots + 2 * (n + 1) = (n + 1) * (n + 2)$ 
    by simp
  finally show ?P (Suc n)
    by simp
qed
```

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

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

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

1. Facts involved in **also** / **finally** calculational chains may be just anything. There is nothing special about **have**, so the natural deduction element **assume** works just as well.
2. There are two *separate* primitives for building natural deduction contexts: **fix** x and **assume** A . Thus it is possible to start reasoning with some new “arbitrary, but fixed” elements before bringing in the actual assumption. In contrast, natural deduction is occasionally formalized with basic context elements of the form $x:A$ instead.

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

theorem *sum-of-odds*:

$$(\sum i::nat=0..<n. 2 * i + 1) = n^{\wedge}Suc (Suc 0)$$

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

proof (*induct n*)

show ?P 0 **by** *simp*

next

fix n

have ?S ($n + 1$) = ?S $n + 2 * n + 1$

by *simp*

also assume ?S $n = n^{\wedge}Suc (Suc 0)$

also have $\dots + 2 * n + 1 = (n + 1)^{\wedge}Suc (Suc 0)$

by *simp*

finally show ?P ($Suc n$)

by *simp*

qed

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

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

theorem *sum-of-squares*:

$$6 * (\sum i::nat=0..n. i^{\wedge}Suc (Suc 0)) = n * (n + 1) * (2 * n + 1)$$

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

proof (*induct n*)

show ?P 0 **by** *simp*

next

fix n

have ?S ($n + 1$) = ?S $n + 6 * (n + 1)^{\wedge}Suc (Suc 0)$

by (*simp add: distrib*)

also assume ?S $n = n * (n + 1) * (2 * n + 1)$

also have $\dots + 6 * (n + 1)^{\wedge}Suc (Suc 0) =$

$$(n + 1) * (n + 2) * (2 * (n + 1) + 1)$$

by (*simp add: distrib*)

finally show ?P ($Suc n$)

by *simp*

qed

theorem *sum-of-cubes*:

$4 * (\sum i::nat=0..n. i^3) = (n * (n + 1)) ^ Suc (Suc 0)$
 (is ?P n is ?S n = -)

proof (*induct n*)

show ?P 0 **by** (*simp add: power-eq-if*)

next

fix n

have ?S (n + 1) = ?S n + 4 * (n + 1) ^ 3

by (*simp add: power-eq-if distrib*)

also assume ?S n = (n * (n + 1)) ^ Suc (Suc 0)

also have ... + 4 * (n + 1) ^ 3 = ((n + 1) * ((n + 1) + 1)) ^ Suc (Suc 0)

by (*simp add: power-eq-if distrib*)

finally show ?P (Suc n)

by *simp*

qed

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

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

end

18 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

18.1 Abstract syntax

typedecl *i*

typedecl *o*

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

18.2 Propositional logic

axiomatization *false* :: *o* ($\perp$)

```

where falseE [elim]:  $\perp \Longrightarrow A$ 

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

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

theorem conjE [elim]:
  assumes  $A \wedge B$ 
  obtains  $A$  and  $B$ 
proof
  from  $\langle A \wedge B \rangle$  show  $A$ 
    by (rule conjD1)
  from  $\langle A \wedge B \rangle$  show  $B$ 
    by (rule conjD2)
qed

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

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

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

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

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

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

```

qed

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

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

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

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

18.3 Equality

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

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

theorem *sym* [*sym*]: $x = y \Longrightarrow y = x$
proof –
assume $x = y$
from *this* and *refl* show $y = x$

by (*rule subst*)
qed

18.4 Quantifiers

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

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

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

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

19 Foundations of HOL

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

The following theory development demonstrates Higher-Order Logic itself, represented directly within the Pure framework of Isabelle. The “HOL” logic given here is essentially that of Gordon [2], although we prefer to present basic concepts in a slightly more conventional manner oriented towards plain Natural Deduction.

19.1 Pure Logic

class *type*
default-sort *type*

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

19.1.1 Basic logical connectives

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

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

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

19.1.2 Extensional equality

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

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

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

theorem *sym* [*sym*]:
 assumes *x* = *y*
 shows *y* = *x*
 using *assms* **by** (*rule subst*) (*rule refl*)

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

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

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

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

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

19.1.3 Derived connectives

definition *false* :: o ($\perp$) **where** $\perp \equiv \forall A. A$

theorem *falseE* [*elim*]:
 assumes $\perp$
 shows A
proof –
 from $\langle \perp \rangle$ **have** $\forall A. A$ **unfolding** *false-def* .
 then **show** A ..
qed

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

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

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

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

theorem *notI* [*intro*]:
 assumes $A \implies \perp$
 shows $\neg A$
using *assms* **unfolding** *not-def* ..

theorem *notE* [*elim*]:
 assumes $\neg A$ **and** A
 shows B
proof –
 from $\langle \neg A \rangle$ **have** $A \longrightarrow \perp$ **unfolding** *not-def* .
 from *this* **and** $\langle A \rangle$ **have** $\perp$..
 then **show** B ..
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 ⇒ o ⇒ o (infixr ∧ 35)
  where conj ≡ λA B. ∀ C. (A → B → C) → C

theorem conjI [intro]:
  assumes A and B
  shows A ∧ B
  unfolding conj-def
proof
  fix C
  show (A → B → C) → C
  proof
    assume A → B → C
    also note ⟨A⟩
    also note ⟨B⟩
    finally show C .
  qed
qed

theorem conjE [elim]:
  assumes A ∧ B
  obtains A and B
proof
  from ⟨A ∧ B⟩ have *: (A → B → C) → C for C
    unfolding conj-def ..
  show A
  proof –
    note * [of A]
    also have A → B → A
    proof
      assume A
      then show B → A ..
    qed
    finally show ?thesis .
  qed
  show B
  proof –
    note * [of B]
    also have A → B → B
    proof
      show B → B ..
    qed
    finally show ?thesis .
  qed
qed

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

```

theorem *disjI1* [*intro*]:

assumes A

shows $A \vee B$

unfolding *disj-def*

proof

fix C

show $(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$

proof

assume $A \longrightarrow C$

from *this* and $\langle A \rangle$ have C ..

then show $(B \longrightarrow C) \longrightarrow C$..

qed

qed

theorem *disjI2* [*intro*]:

assumes B

shows $A \vee B$

unfolding *disj-def*

proof

fix C

show $(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$

proof

show $(B \longrightarrow C) \longrightarrow C$

proof

assume $B \longrightarrow C$

from *this* and $\langle B \rangle$ show C ..

qed

qed

qed

theorem *disjE* [*elim*]:

assumes $A \vee B$

obtains $(a) A \mid (b) B$

proof –

from $\langle A \vee B \rangle$ have $(A \longrightarrow thesis) \longrightarrow (B \longrightarrow thesis) \longrightarrow thesis$

unfolding *disj-def* ..

also have $A \longrightarrow thesis$

proof

assume A

then show *thesis* by (rule *a*)

qed

also have $B \longrightarrow thesis$

proof

assume B

then show *thesis* by (rule *b*)

qed

finally show *thesis* .

qed

```

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

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

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

```

19.2 Cantor's Theorem

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

```

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

```

```

qed
with * show A ..
qed

```

```

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

```

19.3 Classical logic

The subsequent rules of classical reasoning are all equivalent.

```

locale classical =
  assumes classical:  $(\neg A \Longrightarrow A) \Longrightarrow A$ 

```

```

theorem (in classical) Peirce's-Law:  $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$ 
proof
  assume  $a: (A \longrightarrow B) \longrightarrow A$ 
  show A
  proof (rule classical)
    assume  $\neg A$ 
    have  $A \longrightarrow B$ 
    proof
      assume A
      with  $\neg A$  show B by (rule contradiction)
    qed
    with a show A ..
  qed
qed

```

```

theorem (in classical) double-negation:
  assumes  $\neg \neg A$ 
  shows A
proof (rule classical)
  assume  $\neg A$ 
  with  $\neg \neg A$  show ?thesis by (rule contradiction)
qed

```

```

theorem (in classical) tertium-non-datur:  $A \vee \neg A$ 
proof (rule double-negation)
  show  $\neg \neg (A \vee \neg A)$ 
  proof

```

```

    assume  $\neg (A \vee \neg A)$ 
    have  $\neg A$ 
    proof
      assume  $A$  then have  $A \vee \neg A$  ..
      with  $\langle \neg (A \vee \neg A) \rangle$  show  $\perp$  by (rule contradiction)
    qed
    then have  $A \vee \neg A$  ..
    with  $\langle \neg (A \vee \neg A) \rangle$  show  $\perp$  by (rule contradiction)
  qed
qed

theorem (in classical) classical-cases:
  obtains  $A \mid \neg A$ 
  using tertium-non-datur
proof
  assume  $A$ 
  then show thesis ..
next
  assume  $\neg A$ 
  then show thesis ..
qed

lemma
  assumes classical-cases:  $\bigwedge A C. (A \implies C) \implies (\neg A \implies C) \implies C$ 
  shows PROP classical
proof
  fix  $A$ 
  assume *:  $\neg A \implies A$ 
  show  $A$ 
  proof (rule classical-cases)
    assume  $A$ 
    then show  $A$  .
  next
    assume  $\neg A$ 
    then show  $A$  by (rule *)
  qed
qed

end

```

References

- [1] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [2] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

- [3] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.
- [4] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in Higher-Order Logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *LNCS*, 1998.
- [5] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
- [6] L. C. Paulson. *Introduction to Isabelle*.
- [7] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.
- [8] L. C. Paulson. A simple formalization and proof for the mutilated chess board. Technical Report 394, Comp. Lab., Univ. Camb., 1996. <http://www.cl.cam.ac.uk/users/lcp/papers/Reports/mutil.pdf>.
- [9] M. Wenzel. *The Isabelle/Isar Reference Manual*.
- [10] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, LNCS 1690, 1999.
- [11] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.