Miscellaneous Isabelle/Isar examples for Higher-Order Logic

Markus Wenzel
http://www.in.tum.de/~wenzelm/

With contributions by Gertrud Bauer and Tobias Nipkow

May 25, 2015

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 Basic logical reasoning 3
  1.1 Pure backward reasoning ................. 3
  1.2 Variations of backward vs. forward reasoning ............. 5
  1.3 A few examples from “Introduction to Isabelle” .......... 7
    1.3.1 A propositional proof ............... 7
    1.3.2 A quantifier proof .................. 8
    1.3.3 Deriving rules in Isabelle .......... 9

2 Cantor’s Theorem 10

3 The Drinker’s Principle 11

4 Correctness of a simple expression compiler 12
  4.1 Binary operations ....................... 12
  4.2 Expressions ............................ 12
  4.3 Machine ................................. 13
  4.4 Compiler ................................. 13

5 Fib and Gcd commute 16
  5.1 Fibonacci numbers ...................... 16
  5.2 Fib and gcd commute ................... 16
6 Basic group theory 19
  6.1 Groups and calculational reasoning .................. 19
  6.2 Groups as monoids .................................. 21
  6.3 More theorems of group theory ....................... 22

7 Some algebraic identities derived from group axioms – theory context version 24

8 Some algebraic identities derived from group axioms – proof notepad version 26

9 Hoare Logic 28
  9.1 Abstract syntax and semantics ....................... 28
  9.2 Primitive Hoare rules ............................... 29
  9.3 Concrete syntax for assertions ....................... 31
  9.4 Rules for single-step proof ......................... 32
  9.5 Verification conditions ............................. 34

10 Using Hoare Logic 35
  10.1 State spaces ....................................... 35
  10.2 Basic examples .................................... 36
  10.3 Multiplication by addition ......................... 37
  10.4 Summing natural numbers ......................... 38
  10.5 Time .............................................. 40

11 Textbook-style reasoning: the Knaster-Tarski Theorem 41
  11.1 Prose version ...................................... 41
  11.2 Formal versions .................................... 41

12 The Mutilated Checker Board Problem 43
  12.1 Tilings ............................................ 43
  12.2 Basic properties of “below” ......................... 44
  12.3 Basic properties of “evnodd” ....................... 44
  12.4 Dominoes .......................................... 45
  12.5 Tilings of dominoes ................................ 46
  12.6 Main theorem ...................................... 48

13 Nested datatypes 49
  13.1 Terms and substitution ............................. 49
  13.2 Alternative induction ............................... 50

14 Peirce’s Law 50

15 An old chestnut 52
1 Basic logical reasoning

theory Basic-Logic
imports Main
begin

1.1 Pure backward reasoning

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

lemma $I$: $A \rightarrow A$
proof
  assume $A$
  show $A$ by fact
qed

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

lemma $S$: $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$
proof
  assume $A \rightarrow B \rightarrow C$
  show $(A \rightarrow B) \rightarrow A \rightarrow C$
  proof
    assume $A \rightarrow B$
    show $A \rightarrow C$
    proof
      assume $A$
      show $C$
      proof
        (rule mp)
        show $B \rightarrow C$ by (rule mp) fact+
        show $B$ by (rule mp) fact+
      qed
    qed
  qed
qed

Isar provides several ways to fine-tune the reasoning, avoiding excessive de-
tail. 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.

First of all, proof by assumption may be abbreviated as a single dot.

\[
\text{lemma } A \rightarrow A \\
\text{proof} \\
\quad \text{assume } A \\
\quad \text{show } A \text{ by } \text{fact} \\
\text{qed}
\]

In fact, concluding any (sub-)proof already involves solving any remaining goals by assumption\(^1\). Thus we may skip the rather vacuous body of the above proof as well.

\[
\text{lemma } A \rightarrow A \\
\text{proof} \\
\text{qed}
\]

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

\[
\text{lemma } A \rightarrow A \\
\quad \text{by } \text{rule}
\]

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

\[
\text{lemma } A \rightarrow A ..
\]

Thus we have arrived at an adequate representation of the proof of a tautology that holds by a single standard rule.\(^2\)

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

The \texttt{intro} proof method repeatedly decomposes a goal’s conclusion.\(^3\)

\[
\text{lemma } A \rightarrow B \rightarrow A \\
\text{proof } (\text{intro } \text{impl}) \\
\quad \text{assume } A \\
\quad \text{show } A \text{ by } \text{fact} \\
\text{qed}
\]

Again, the body may be collapsed.

\[
\text{lemma } A \rightarrow B \rightarrow A
\]

---

\(^1\)This is not a completely trivial operation, as proof by assumption may involve full higher-order unification.

\(^2\)Apparently, the rule here is implication introduction.

\(^3\)The dual method is \textit{elim}, acting on a goal’s premises.
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.

1.2 Variations of backward vs. forward reasoning

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

The first version is purely backward.

```isar
lemma A \land B \to B \land A
proof
  assume A \land B
  show B \land A
  proof
    show B by (rule conjunct2) fact
    show A by (rule conjunct1) fact
  qed
qed
```

Above, the conjunct-1/2 projection rules 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 $A \land B$ assumption as the current facts, enabling the use of double-dot proofs. Note that from already does forward-chaining, involving the conjE rule here.

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

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

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

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

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 “\texttt{—}” proof method.

\begin{verbatim}
lemma A \land B \rightarrow B \land A
proof —
  \{ 
    assume A \land B
    from \( A \land B \) have A ..
    from \( A \land B \) have B ..
    from \( B \langle A \rangle \) have B \land A ..
  \}
  then show \texttt{?thesis} .. — rule \texttt{imp}\texttt{I}
qed
\end{verbatim}

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 \textit{right} balance of top-down backward decomposition, and bottom-up for-
ward 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.

\begin{verbatim}
lemma A \land B \implies B \land A
proof
  assume A \land B
  then show B \land A
  proof
    assume B A
    then show thesis
  qed
qed
\end{verbatim}

1.3 A few examples from “Introduction to Isabelle”

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

1.3.1 A propositional proof

We consider the proposition \( P \lor P \implies P \). The proof below involves forward-chaining from \( P \lor P \), followed by an explicit case-analysis on the two identical cases.

\begin{verbatim}
lemma P \lor P \implies P
proof
  assume P \lor P
  then show P
  proof
    — rule disjE:
    A \lor B
      \[A]\ \[B]\n    \vdash C
    \vdash C
  qed
next
  assume P show P by fact
qed
\end{verbatim}

Case splits are not hardwired into the Isar language as a special feature. The \texttt{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 \texttt{next} statement then closes one block level, opening a new one. The resulting behavior 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 \textit{not} require \texttt{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.

\begin{verbatim}
lemma P ∨ P → P
proof
  assume P ∨ P
  then show P
  proof
    assume P
    show P by fact
    show P by fact
  qed
qed
\end{verbatim}

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.

\begin{verbatim}
lemma P ∨ P → P
proof
  assume P ∨ P
  then show P
proof (rule exE)
  fix a
\end{verbatim}

1.3.2 A quantifier proof

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

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

\begin{verbatim}
lemma (∃x. P (f \, x)) → (∃y. P y)
proof
  assume (∃x. P (f \, x))
  then show (∃y. P y)
  proof (rule exE)
    fix a
\end{verbatim}
assume \( P(f \ a) \) (is \( P \) \( \texttt{?witness} \))
then show \( \texttt{thesis} \) by (\textit{rule exI} [of \( P \) \( \texttt{?witness} \)])
qed

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

\[
\text{lemma} \ (\exists x. \ P(f \ x)) \longrightarrow (\exists y. \ P \ y)
\]

\[
\text{proof}
\]
assume \( \exists x. \ P(f \ x) \)
then show \( \exists y. \ P \ y \)
proof
fix \( a \)
assume \( P(f \ a) \)
then show \( \texttt{thesis} \) ..
qed

\[
\text{qed}
\]

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

\[
\text{lemma} \ (\exists x. \ P(f \ x)) \longrightarrow (\exists y. \ P \ y)
\]

\[
\text{proof}
\]
assume \( \exists x. \ P(f \ x) \)
then obtain \( a \) where \( P(f \ a) \) ..
then show \( \exists y. \ P \ y \) ..
qed

\[
\text{qed}
\]

Technically, \texttt{obtain} is similar to \texttt{fix} and \texttt{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 \texttt{obtain} statement may \textit{not} refer to the parameters introduced there.

1.3.3 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.

\[
\text{theorem} \ \texttt{conjE} : \ A \land B \implies (A \implies B \implies C) \implies C
\]

\[
\text{proof}
\]
assume \( A \land B \)
assume \( r : A \implies B \implies C \)
show C
proof (rule r)
  show A by (rule conjunct1) fact
  show B by (rule conjunct2) fact
qed
qed
end

2 Cantor’s Theorem

theory Cantor
imports Main
begin

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

\[ \forall f : \alpha \to \alpha \to \text{bool}. \exists S : \alpha \to \text{bool}. \forall x : \alpha. f x \neq S \]

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

theorem \( \exists S. S \notin \text{range} (f :: 'a \Rightarrow 'a \text{ set}) \)
proof
  let ?S = \{x. x \notin f x\}
  show ?S \notin 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 \( y : f y \) show ?thesis by contradiction
      next
        assume y \notin ?S
        assume y \notin f y
        then have y \in ?S ..
        with \( y \notin ?S \) show ?thesis by contradiction
      qed
  qed
qed

\footnote{This is an Isar version of the final example of the Isabelle/HOL manual [5].}
How much creativity is required? As it happens, Isabelle can prove this theorem automatically using best-first search. Depth-first search would diverge, but best-first search successfully navigates through the large search space. The context of Isabelle’s classical prover contains rules for the relevant constructs of HOL’s set theory.

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

While this establishes the same theorem internally, we do not get any idea of how the proof actually works. There is currently no way to transform internal system-level representations of Isabelle proofs back into Isar text. Writing intelligible proof documents really is a creative process, after all.

end

3 The Drinker’s Principle

\textbf{theory} Drinker \\
\textbf{imports} Main \\
\textbf{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.

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

\textbf{theorem} Drinker’s-Principle: \exists x. drunk x \rightarrow (\forall x. drunk x) \\
\textbf{proof} \textbf{cases} \\
\textbf{fix} a \textbf{ assume} \forall x. drunk x \\
\textbf{then have} drunk a \rightarrow (\forall x. drunk x) .. \\
\textbf{then show} \textbf{thesis} .. \\
\textbf{next} \\
\textbf{assume} \neg (\forall x. drunk x)
then have \( \exists x. \neg \text{drunk } x \) by (rule de-Morgan)
then obtain a where a: \( \neg \text{drunk } a \).

have \( \text{drunk } a \rightarrow (\forall x. \text{drunk } x) \)
proof
  assume \( \text{drunk } a \)
  with a show \( \forall x. \text{drunk } x \) by contradiction
qed
then show ?thesis ..
qed

end

4 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.

4.1 Binary operations

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

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

4.2 Expressions

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

datatype (dead 'adr, dead 'val) expr =
  Variable 'adr
| Constant 'val
| Binop 'val binop ('adr, 'val) expr ('adr, 'val) expr

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

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

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

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

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

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

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

4.4 Compiler

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

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

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

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

theorem correctness: execute (compile e) env = eval e env
proof -
  have \( \forall \text{stack. } \text{exec} (\text{compile } e) \text{ stack env} = \text{eval } e \text{ env # stack} \)
  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':
\ \text{exec} (xs @ ys) \text{ stack env} = \text{exec } ys (\text{exec } xs \text{ stack env}) \text{ env}
proof (induct xs arbitrary: stack)
  case (Nil s)
  have \( \text{exec} ([] @ ys) \text{ s env} = \text{exec } ys \text{ s env} \)
    by simp
  also have \( \ldots = \text{exec } ys (\text{exec } [] \text{ s env}) \text{ env} \)
    by simp
  finally show ?case .
  next
    case (Cons x xs s)
    show ?case
    proof (induct x)
      case (Const val)
      have \( \text{exec} ((\text{Const } val \# xs) @ ys) \text{ s env} = \text{exec } (\text{Const } val \# xs @ ys) \text{ s env} \)
        by simp
      also have \( \ldots = \text{exec } (xs @ ys) (val \# s) \text{ env} \)
        by simp
      also from Cons have \( \ldots = \text{exec } ys (\text{exec } xs (val \# s) \text{ env}) \text{ env} \).
      also have \( \ldots = \text{exec } ys (\text{exec } (\text{Const } val \# xs) \text{ s env}) \text{ env} \)
        by simp

14
finally show \texttt{?case}.

next

\texttt{case (Load adr)}
from \texttt{Cons show ?case}
by \texttt{simp} — same as above

next

\texttt{case (Apply fn)}
\begin{itemize}
  \item \texttt{have exec ((Apply fn \# xs) \@ ys) s env = exec (Apply fn \# xs \@ ys) s env by simp}
  \item \texttt{also have \ldots = exec (xs \@ ys) (fn (hd s) (hd (tl s)) \# (tl (tl s))) env by simp}
  \item \texttt{also from Cons have \ldots = exec ys (exec xs (fn (hd s) (hd (tl s)) \# tl (tl s))) env .}
  \item \texttt{also have \ldots = exec ys (exec (Apply fn \# xs) s env) env by simp}
\end{itemize}

finally show \texttt{?case}.

qed

qed

\textbf{theorem} correctness'': \texttt{execute (compile e) env = eval e env}

\begin{proof}
\begin{itemize}
  \item \texttt{have exec-compile: \forall stack. exec (compile e) stack env = eval e env \# stack}
  \item \texttt{proof (induct e)}
  \begin{itemize}
    \item \texttt{case (Variable adr s)}
    \begin{itemize}
      \item \texttt{have exec (compile (Variable adr)) s env = exec [Load adr] s env by simp}
      \item \texttt{also have \ldots = env adr \# s by simp}
      \item \texttt{also have env adr = eval (Variable adr) env by simp}
    \end{itemize}
    \item \texttt{finally show ?case}.
  \end{itemize}
  \item \texttt{next}
  \begin{itemize}
    \item \texttt{case (Constant val s)}
    \item \texttt{show ?case by simp} — same as above
  \end{itemize}
  \item \texttt{next}
  \begin{itemize}
    \item \texttt{case (Binop fn e1 e2 s)}
    \begin{itemize}
      \item \texttt{have exec (compile (Binop fn e1 e2)) s env = exec (compile e2 \@ compile e1 \@ [Apply fn]) s env by simp}
      \item \texttt{also have \ldots = exec [Apply fn] (exec (compile e1) (exec (compile e2) s env) env) env by simp only: exec-append}
      \item \texttt{also have exec (compile e2) s env = eval e2 env \# s by fact}
      \item \texttt{also have exec (compile e1) \ldots env = eval e1 env \# \ldots by fact}
      \item \texttt{also have exec [Apply fn] \ldots env = fn (hd \ldots) (hd (tl \ldots)) \# (tl (tl \ldots))}
    \end{itemize}
  \end{itemize}
\end{itemize}
\end{proof}
by simp
also have \ldots = fn (eval e1 env) (eval e2 env) # s
by simp
also have \(\text{fn} (\text{eval e1 env}) (\text{eval e2 env}) = \text{eval (Binop fn e1 e2 env)}\)
by simp
finally show \(? \text{case} \).
qed

have \(\text{execute (compile e) env = hd (\text{exec (compile e) [] env})}\)
by (simp add: execute-def)
also from exec-compile have \(\text{exec (compile e) [] env = [\text{eval e env}] .}\)
also have \(\text{hd \ldots = eval e env}\)
by simp
finally show \(? \text{thesis} \).
qed

end

5 Fib and Gcd commute

theory Fibonacci
imports ../Number-Theory/Primes
begin\(^\text{5}\) declare One-nat-def [simp]

5.1 Fibonacci numbers

fun fib :: nat \Rightarrow nat where
\begin{align*}
\text{fib } 0 &= 0 \\
\text{fib } (\text{Suc } 0) &= 1 \\
\text{fib } (\text{Suc } (\text{Suc } x)) &= \text{fib } x + \text{fib } (\text{Suc } x)
\end{align*}

lemma [simp]: \(\text{fib } (\text{Suc } n) > 0\)
by (induct n rule: fib.induct) simp-all

Alternative induction rule.

theorem fib-induct:
fixes \(n :: \text{nat}\)
shows \(P \ 0 \Rightarrow \ P \ 1 \Rightarrow (\forall n. \ P \ (n + 1) \Rightarrow \ P \ n \Rightarrow \ P \ (n + 2)) \Rightarrow \ P \ n\)
by (induct rule: fib.induct) simp-all

5.2 Fib and \text{gcd} commute

A few laws taken from \[2\].

lemma fib-add:
\(^\text{5}\)Isar version by Gertrud Bauer. Original tactic script by Larry Paulson. A few proofs of laws taken from \[2\].
\[
\text{fib} \ (n + k + 1) = \text{fib} \ (k + 1) \ast \text{fib} \ (n + 1) + \text{fib} \ k \ast \text{fib} \ n
\]
(is ?P n)
— see [2, 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) \ast fib \ (n + 1) + fib \ k \ast fib \ n
    (is - = ?R1)
  also assume fib \ (n + 1 + k + 1)
    = fib \ (k + 1) \ast fib \ (n + 1 + 1) + fib \ k \ast fib \ (n + 1)
    (is - = ?R2)
  also have ?R1 + ?R2
    = fib \ (k + 1) \ast fib \ (n + 2 + 1) + fib \ k \ast fib \ (n + 2)
    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 fib \ (n + 2 + 1)
    = fib \ (n + 1) + fib \ (n + 2)
    by simp
  also have \ldots = fib \ (n + 2) + fib \ (n + 1)
    by simp
  also have \ldots = gcd \ (fib \ (n + 2)) \ldots = gcd \ (fib \ (n + 2)) \ (fib \ (n + 1))
    by (rule gcd-add2-nat)
  also have \ldots = gcd \ (fib \ (n + 1)) \ (fib \ (n + 1 + 1))
    by (simp add: gcd-commute-nat)
  also assume \ldots = 1
  finally show ?P \ (n + 2) .
qed

lemma gcd-mult-add: \( \langle \ 0: \text{nat} \rangle < n \Longrightarrow \text{gcd} \ (n \ast k + m) \ n = \text{gcd} \ m \ n \)
proof
  assume \( 0 < n \)
  then have \( \text{gcd} \ (n \ast k + m) \ n = \text{gcd} \ n \ (m \ mod \ n) \)
    by (simp add: gcd-non-0-nat add.commute)
  also from \( \langle 0 < n \rangle \) have \ldots = \text{gcd} \ m \ n
    by (simp add: gcd-non-0-nat)
  finally show \( \text{thesis} \).
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 \(gcd (fib m) (fib (n + m)) = gcd (fib (n + k + 1)) (fib (k + 1))\)
    by (simp add: gcd-commute-nat)
  also have \(fib (n + k + 1)\)
    \(= fib (k + 1) * fib (n + 1) + fib k * fib n\)
    by (rule fib-add)
  also have \(gcd \ldots (fib (k + 1)) = gcd (fib k * fib n) (fib (k + 1))\)
    by (simp add: gcd-mult-add)
  also have \(\ldots = gcd (fib n) (fib (k + 1))\)
    by (simp only: gcd-fib-Suc-eq-1 gcd-mult-cancel-nat)
  also have \(\ldots = gcd (fib m) (fib n)\)
    using Suc by (simp add: gcd-commute-nat)
  finally show \(?thesis\).
qed

lemma \(gcd\-fib\-diff\): 
  assumes \(m \leq n\)
  shows \(gcd (fib m) (fib (n - m)) = gcd (fib m) (fib n)\)
proof
  have \(gcd (fib m) (fib (n - m)) = gcd (fib m) (fib (n - m + m))\)
    by (simp add: gcd-fib-add)
  also from \(m \leq n\) have \(n - m + m = n\)
    by simp
  finally show \(?thesis\).
qed

lemma \(gcd\-fib\-mod\): 
  assumes \(0 < m\)
  shows \(gcd (fib m) (fib (n \ mod \ m)) = gcd (fib m) (fib n)\)
proof (induct n rule: nat-less-induct)
  case (1 n)
  note hyp = this
  show \(?case\)
  proof
    have \(n \ mod \ m = (if n < m \ then \ n - m \ mod \ m)\)
      by (rule mod-if)
    also have \(gcd \ldots (fib m) (fib n)\)
      by (cases n < m)
    case True
    then show \(?thesis\) by simp
  next
    case False
    then have \(m \leq n\) by simp
    from \(0 < m\) and False have \(n - m < n\)
      by simp
    with hyp have \(gcd (fib m) (fib ((n - m) \ mod \ m))\)
  qed
\[ \text{gcd} \left( \text{fib} \left( m \right) \right) \left( \text{fib} \left( n \right) \right) \]

also have \( \ldots = \text{gcd} \left( \text{fib} \left( m \right) \right) \left( \text{fib} \left( n \right) \right) \)

by simp

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

finally have \( \text{gcd} \left( \text{fib} \left( m \right) \right) \left( \text{fib} \left( \left( n - m \right) \mod m \right) \right) = \)

\( \text{gcd} \left( \text{fib} \left( m \right) \right) \left( \text{fib} \left( n \right) \right) \).

with False show \( \text{thesis} \) by simp

qed

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

qed

theorem fib-gcd: \( \text{fib} \left( \text{gcd} \left( m \right) \left( n \right) \right) = \text{gcd} \left( \text{fib} \left( m \right) \right) \left( \text{fib} \left( n \right) \right) \)

(is \( \text{?P m n} \))

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

fix \( m \)

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

by simp

fix \( n :: \text{nat} \)

assume \( \text{?P m n} \)

then have \( \text{gcd} \left( m \right) \left( n \right) = \text{gcd} \left( n \right) \left( m \mod n \right) \)

by (simp add: gcd-non-0-nat)

also assume \( \text{hyp: fib} \ldots = \text{gcd} \left( \text{fib} \left( n \right) \right) \left( \text{fib} \left( \left( m \mod n \right) \right) \right) \)

also from \( \text{?P m n} \) have \( \ldots = \text{gcd} \left( \text{fib} \left( n \right) \right) \left( \text{fib} \left( m \right) \right) \)

by (rule gcd-fib-mod)

also have \( \ldots = \text{gcd} \left( \text{fib} \left( m \right) \right) \left( \text{fib} \left( n \right) \right) \)

by (rule gcd-commute-nat)

finally show \( \text{fib} \left( \text{gcd} \left( m \right) \left( n \right) \right) = \text{gcd} \left( \text{fib} \left( m \right) \right) \left( \text{fib} \left( n \right) \right) \).

qed

end

6 Basic group theory

theory Group

imports Main

begin

6.1 Groups and calculational reasoning

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

class group = \( \text{times} + \text{one} + \text{inverse} + \)

assumes group-assoc: \( \left( x \ast y \right) \ast z = x \ast \left( y \ast z \right) \)

and group-left-one: \( 1 \ast x = x \)

and group-left-inverse: \( \text{inverse} x \ast x = 1 \)

The group axioms only state the properties of left one and inverse, the right versions may be derived as follows.
theorem (in group) group-right-inverse: \( x \ast \text{inverse } x = 1 \)
proof
  have \( x \ast \text{inverse } x = 1 \ast (x \ast \text{inverse } x) \)
    by (simp only: group-left-one)
  also have \( \ldots = 1 \ast x \ast \text{inverse } x \)
    by (simp only: group-assoc)
  also have \( \ldots = \text{inverse } (\text{inverse } x) \ast \text{inverse } x \ast x \ast \text{inverse } x \)
    by (simp only: group-left-inverse)
  also have \( \ldots = \text{inverse } (\text{inverse } x) \ast (\text{inverse } x \ast x) \ast \text{inverse } x \)
    by (simp only: group-assoc)
  also have \( \ldots = \text{inverse } (\text{inverse } x) \ast 1 \ast \text{inverse } x \)
    by (simp only: group-right-inverse)
  also have \( \ldots = \text{inverse } (\text{inverse } x) \ast 1 \)
    by (simp only: group-left-one)
  finally show \( \text{thesis} \).
qed

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

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

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

Note that “\ldots” is just a special term variable that is bound automatically to the argument\(^6\) of the last fact achieved by any local assumption or proven statement. In contrast to \(\text{thesis}\), the “\ldots” variable is bound after the proof is finished, though.

\(^6\)The argument of a curried infix expression happens to be its right-hand side.
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.

\textbf{theorem (in group)} \(x \ast 1 = x\)

\textbf{proof –}

\begin{itemize}
  \item \textbf{have} \(x \ast 1 = x \ast (\text{inverse} \ x \ast x)\)
    \textbf{by} (simp only: group-left-inverse)
  \item \textbf{note} calculation = this
    \begin{itemize}
      \item first calculational step: init calculation register
    \end{itemize}
  \item \textbf{have} \ldots = \text{inverse} \ x \ast x
    \textbf{by} (simp only: group-assoc)
  \item \textbf{note} calculation = trans [OF calculation this]
    \begin{itemize}
      \item general calculational step: compose with transitivity rule
    \end{itemize}
  \item \textbf{have} \ldots = 1 \ast x
    \textbf{by} (simp only: group-right-inverse)
  \item \textbf{note} calculation = trans [OF calculation this]
    \begin{itemize}
      \item general calculational step: compose with transitivity rule
    \end{itemize}
  \item \textbf{have} \ldots = x
    \textbf{by} (simp only: group-left-one)
  \item \textbf{note} calculation = trans [OF calculation this]
    \begin{itemize}
      \item final calculational step: compose with transitivity rule \ldots
    \end{itemize}
  \item \textbf{from} calculation
    \begin{itemize}
      \item \ldots and pick up the final result
    \end{itemize}
\end{itemize}

\textbf{show} \(?\text{thesis}\).
\textbf{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.

\subsection*{6.2 Groups as monoids}

Monoids over signature \((\times :: \alpha \to \alpha \to \alpha, \text{one} :: \alpha)\) are defined like this.

\textbf{class} \textit{monoid} = \textit{times} + \textit{one} +
assumes monoid-assoc: \((x \ast y) \ast z = x \ast (y \ast z)\)
and monoid-left-one: \(1 \ast x = x\)
and monoid-right-one: \(x \ast 1 = x\)

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 page 20), we may still instantiate group \(\subseteq\) monoid properly as follows.

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

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

6.3 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: \(e \ast x = x\)
  shows \(1 = e\)
proof –
  have \(1 = x \ast \text{inverse } x\)
    by (simp only: group-right-inverse)
  also have \(\ldots = (e \ast x) \ast \text{inverse } x\)
    by (simp only: eq)
  also have \(\ldots = e \ast (x \ast \text{inverse } x)\)
    by (simp only: group-assoc)
  also have \(\ldots = e \ast 1\)
    by (simp only: group-right-inverse)
  also have \(\ldots = e\)
    by (simp only: group-right-one)
  finally show ?thesis .
qed

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

theorem (in group) group-inverse-equality:
  assumes eq: \(x' \ast x = 1\)
  shows \(\text{inverse } x = x'\)
proof –
The inverse operation has some further characteristic properties.

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

**Proof (rule group-inverse-equality)**
- show \((\text{inverse } y * \text{inverse } x) * (x * y) = 1\)
  - proof –
    - have \((\text{inverse } y * \text{inverse } x) * (x * y) = (\text{inverse } y * (\text{inverse } x * x)) * y\)
      - by \((\text{simp only: group-assoc})\)
    - also have \(...) = (\text{inverse } y * 1) * y\)
      - by \((\text{simp only: group-left-inverse})\)
    - also have \(...) = \text{inverse } y * y\)
      - by \((\text{simp only: group-right-inverse})\)
    - also have \(...) = 1\)
      - by \((\text{simp only: group-right-inverse})\)
  - finally show \(?\text{thesis}\).

**Qed**

**Qed**

**Qed**

**Theorem (in group) inverse-inverse:** inverse \((\text{inverse } x)\) = \(x\)

**Proof (rule group-inverse-equality)**
- show \(x * \text{inverse } x = 1\)
  - by \((\text{simp only: group-right-inverse})\)

**Qed**

**Theorem (in group) inverse-inject:**
- assumes \(eq: \text{inverse } x = \text{inverse } y\)
- shows \(x = y\)

**Proof –
  - have \(x = x * 1\)
    - by \((\text{simp only: group-right-one})\)
  - also have \(...) = x * (\text{inverse } y * y)\)
    - by \((\text{simp only: group-left-inverse})\)
  - also have \(...) = x * (\text{inverse } x * y)\)
    - by \((\text{simp only: eq})\)
  - also have \(...) = (x * \text{inverse } x) * y\)
    - by \((\text{simp only: group-assoc})\)
also have \ldots = 1 * y
   by (simp only: group-right-inverse)
also have \ldots = y
   by (simp only: group-left-one)
finally show thesis.
qed

end

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

theory Group-Context
imports Main
begin

hypothesical group axiomatization

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

some consequences

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

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

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

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

end
8 Some algebraic identities derived from group axioms – proof notepad version

theory Group-Notepad
imports Main
begin

notepad begin

hypothetical group axiomatization

fix prod :: 'a ⇒ 'a ⇒ 'a (infixl ** 70)
and one :: 'a
and inverse :: 'a ⇒ 'a
assume assoc: ∀x y z. (x ** y) ** z = x ** (y ** z)
and left-one: ∀x. one ** x = x
and left-inverse: ∀x. inverse x ** x = one

some consequences

have right-inverse: ∀x. x ** inverse x = one
proof –
  fix x
  have x ** inverse x = one ** (x ** inverse x)
    by (simp only: left-one)
  also have ... = one ** x ** inverse x
    by (simp only: assoc)
  also have ... = inverse (inverse x) ** inverse x ** x ** inverse x
    by (simp only: left-inverse)
  also have ... = inverse (inverse x) ** (inverse x ** x) ** inverse x
    by (simp only: assoc)
  also have ... = inverse (inverse x) ** one ** inverse x
    by (simp only: left-inverse)
  also have ... = inverse (inverse x) ** (one ** inverse x)
    by (simp only: assoc)
  also have ... = inverse (inverse x) ** inverse x
    by (simp only: left-one)
  also have ... = one
    by (simp only: left-inverse)
finally show x ** inverse x = one .
qed

have right-one: ∀x. x ** one = x
proof –
  fix x
  have x ** one = x ** (inverse x ** x)
    by (simp only: left-inverse)
also have \( \ldots = x \ast x \ast x \)
by \((\text{simp only: assoc})\)
also have \( \ldots = \mathit{one} \ast x \)
by \((\text{simp only: right-inverse})\)
also have \( \ldots = x \)
by \((\text{simp only: left-one})\)
finally show \( x \ast \mathit{one} = x \).

\[ \text{qed} \]

have \( \text{one-equality}: \coprod x. e \ast x = x \implies \mathit{one} = e \)
proof –
\begin{itemize}
\item fix \( e, x \)
\item assume eq: \( e \ast x = x \)
\item have \( \mathit{one} = x \ast x \ast x \)
by \((\text{simp only: right-inverse})\)
\item also have \( \ldots = (e \ast x) \ast \mathit{inverse} x \)
by \((\text{simp only: eq})\)
\item also have \( \ldots = e \ast (x \ast \mathit{inverse} x) \)
by \((\text{simp only: assoc})\)
\item also have \( \ldots = e \ast \mathit{one} \)
by \((\text{simp only: right-inverse})\)
\item also have \( \ldots = e \)
by \((\text{simp only: right-one})\)
\item finally show \( \mathit{one} = e \).
\end{itemize}
\[ \text{qed} \]

have \( \text{inverse-equality}: \coprod x \mathit{x'}. x \ast x' = \mathit{one} \implies \mathit{inverse} x = x' \)
proof –
\begin{itemize}
\item fix \( x, x' \)
\item assume eq: \( x' \ast x = \mathit{one} \)
\item have \( \mathit{inverse} x = x \ast x \ast \mathit{inverse} x \)
by \((\text{simp only: left-one})\)
\item also have \( \ldots = (x' \ast x) \ast \mathit{inverse} x \)
by \((\text{simp only: eq})\)
\item also have \( \ldots = x' \ast (x \ast \mathit{inverse} x) \)
by \((\text{simp only: assoc})\)
\item also have \( \ldots = x' \ast \mathit{one} \)
by \((\text{simp only: right-inverse})\)
\item also have \( \ldots = x' \)
by \((\text{simp only: right-one})\)
\item finally show \( \mathit{inverse} x = x' \).
\end{itemize}
\[ \text{qed} \]

end
9 Hoare Logic

theory Hoare
imports Main
begin

9.1 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 [10, §6]. See also ~/src/HOL/Hoare and [4].

type-synonym 'a bexp = 'a set

type-synonym 'a assn = 'a set

datatype 'a com =
  Basic 'a ⇒ 'a
| Seq 'a com 'a com
| Cond 'a bexp 'a com 'a com
| While 'a bexp 'a assn 'a com

abbreviation Skip (SKIP)
where SKIP ≡ Basic id

type-synonym 'a sem = 'a ⇒ 'a ⇒ bool

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

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

definition Valid :: 'a bexp ⇒ 'a com ⇒ 'a bexp ⇒ bool
where ⊢ P c Q ←→ (∀ s s′. Sem c s s′ → s ∈ P → s′ ∈ Q)

lemma ValidI [intro?]:
  (∀ s s′. Sem c s s′ → s ∈ P → s′ ∈ Q) → ⊢ P c Q
by (simp add: Valid-def)

lemma ValidD [dest?]:
  ⊢ P c Q → Sem c s s′ → s ∈ P → s′ ∈ Q
by (simp add: Valid-def)
9.2 Primitive Hoare rules

From the semantics defined above, we derive the standard set of primitive Hoare rules; e.g. see [10, §6]. Usually, variant forms of these rules are applied in actual proof, see also §9.4 and §9.5.

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 §9.4.

**theorem basic**: \( \vdash \{ s \cdot f s \in P \} (\text{Basic } f) P \)

**proof**
- fix \( s s' \)
- assume \( s : s \in \{ s \cdot f s \in P \} \)
- assume \( \text{Sem } (\text{Basic } f) s s' \)
- then have \( s' = f s \) by simp

with \( s \) show \( s' \in P \) by simp

qed

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

**theorem seq**: \( \vdash P c_1 Q \Rightarrow \vdash Q c_2 R \Rightarrow \vdash P (c_1; c_2) R \)

**proof**
- assume \( \text{cmd1}: \vdash P c_1 Q \) and \( \text{cmd2}: \vdash Q c_2 R \)
- fix \( s s' \)
- assume \( s : s \in P \)
- assume \( \text{Sem } (c_1; c_2) s s' \)
- then obtain \( s'' \) where \( \text{sem1}: \text{Sem } c_1 s s'' \) and \( \text{sem2}: \text{Sem } c_2 s'' s' \)
  - by auto
- from \( \text{cmd1 sem1 } s \) have \( s'' \in Q \)
- with \( \text{cmd2 sem2 } s \) show \( s' \in R \)

qed

**theorem conseq**: \( P' \subseteq P \Rightarrow \vdash P c Q \Rightarrow Q \subseteq Q' \Rightarrow \vdash P' c Q' \)

**proof**
- assume \( \text{P'P}: P' \subseteq P \) and \( \text{QQ'}: Q \subseteq Q' \)
- assume \( \text{cmd}: \vdash P c Q \)
- fix \( s s' : 'a \)
- assume \( \text{sem}: \text{Sem } c s s' \)
- assume \( s : P' \) with \( \text{P'P } s \) have \( s \in P \)
- with \( \text{cmd sem } s \) have \( s' \in Q \)
- with \( \text{QQ'} \) 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 \( \text{case-b}: \vdash (P \cap b) c_1 Q \)
  and \( \text{case-nb}: \vdash (P \cap \neg b) c_2 Q \)
- shows \( \vdash P (\text{Cond } b c_1 c_2) Q \)
proof
  fix s s'
  assume s: s ∈ P
  assume sem: Sem (Cond b c1 c2) s s'
  show s' ∈ Q
proof cases
  assume b: s ∈ b
  from case-b show ?thesis
  proof
    from sem b show Sem c1 s s' by simp
    from s b show s ∈ P ∩ b by simp
  qed
next
  assume nb: s /∈ b
  from case-nb show ?thesis
  proof
    from sem nb show Sem c2 s s' by simp
    from s nb show s : P ∩ ¬b by simp
  qed
qed

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

theorem while:
  assumes body: ⊢ (P ∩ b) c P
  shows ⊢ P (While b X c) (P ∩ ¬b)
proof
  fix s s' assume s: s ∈ 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' ∈ P ∩ ¬b
proof (induct n arbitrary: s)
  case 0
  then show ?case by auto
next
  case (Suc n)
  then obtain s'' where b: s ∈ b and sem: Sem c s s''
    and iter: iter n b (Sem c) s'' s' by auto
  from Suc and b have s ∈ P ∩ b by simp
  with body sem have s'' ∈ P ..
  with iter show ?case by (rule Suc)
qed

qed
9.3 Concrete syntax for assertions

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

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

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

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

**syntax**

- `quote`: `'b ⇒ ('a ⇒ 'b)`
- `antiquote`: '(a ⇒ 'b) ⇒ 'b (·· [1000] 1000)
- `Subst`: 'a bexp ⇒ 'b ⇒ idt ⇒ 'a bexp (··[··] [1000] 999)
- `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, 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.
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

9.4 Rules for single-step proof

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

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

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

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

lemma [trans]:
\[\vdash \{P\} c Q \Rightarrow (\forall s. P' s \rightarrow P s) \Rightarrow \vdash \{P'\} c Q\]
  by (simp add: Valid-def)
lemma [trans]:
\[ (\forall s. P' s \rightarrow P s) \implies \vdash \{ \langle P \rangle \} c Q \implies \vdash \{ \langle P' \rangle \} c Q \]
by (simp add: Valid-def)

**lemma** [trans]:
\[ \vdash P c \{ \langle Q \rangle \} \implies (\forall s. Q s \rightarrow Q' s) \implies \vdash P c \{ \langle Q' \rangle \} \]
by (simp add: Valid-def)

**lemma** [trans]:
\[ (\forall s. Q s \rightarrow Q' s) \implies \vdash P c \{ \langle Q \rangle \} \implies \vdash P c \{ \langle Q' \rangle \} \]
by (simp add: Valid-def)

Identity and basic assignments.\(^7\)

**lemma** skip [intro?]: \[ \vdash P \text{ SKIP } P \]
**proof** –
\begin{itemize}
\item have \[ \vdash \{ \text{s}, \text{id s} \in P \} \text{ SKIP } P \vdash (\text{rule basic}) \]
\item then show \[ ?\text{thesis} \text{ by simp} \]
\end{itemize}
**qed**

**lemma** assign: \[ \vdash P [\langle a / 'x::'a \rangle] 'x := 'a P \]
by (rule basic)

Note that above formulation of assignment corresponds to our preferred way to model state spaces, using (extensible) record types in HOL \([3]\). For any record field \(x\), Isabelle/HOL provides a functions \(x\) (selector) and \(x\)-update (update). Above, there is only a place-holder appearing for the latter kind of function: due to concrete syntax \(\hat{x} := \hat{a}\) also contains \(x\text{.update}\).\(^8\)

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

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

**lemma** seq-assoc [simp]: \[ \vdash P \langle c1; (c2; c3) \rangle Q \longleftrightarrow \vdash P \langle c1; c2 \rangle c3 Q \]
by (auto simp add: Valid-def)

Conditional statements.

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

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

While statements — with optional invariant.

\(^7\)The *hoare* method introduced in \(\S 9.5\) is able to provide proper instances for any number of basic assignments, without producing additional verification conditions.

\(^8\)Note that due to the external nature of HOL record fields, we could not even state a general theorem relating selector and update functions (if this were required here); this would only work for any particular instance of record fields introduced so far.
lemma \([\text{intro?}]: \vdash (P \cap b) \ c \ P \implies \vdash P \ (\text{While} \ b \ P \ c) \ (P \cap -b)\)
by (rule while)

lemma \([\text{intro?}]: \vdash (P \cap b) \ c \ P \implies \vdash P \ (\text{While} \ b \ undefined \ c) \ (P \cap -b)\)
by (rule while)

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

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

9.5 Verification conditions

We now load the original ML file for proof scripts and tactic definition for the Hoare Verification Condition Generator (see ~/src/HOL/Hoare/). As far as we are concerned here, the result is a proof method \text{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 \text{WHILE} loops to be fully annotated with invariants beforehand. Furthermore, only \textit{concrete} pieces of code are handled — the underlying tactic fails ungracefully if supplied with meta-variables or parameters, for example.

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

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

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

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

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

\textbf{lemma \text{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: \( \neg \{ x. \neg b \, x \} \)
by blast

lemmas AbortRule = SkipRule — dummy version

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

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

end

10 Using Hoare Logic

theory Hoare-Ex
imports Hoare
begin

10.1 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.
10.2 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 ⊢ {⌜(N-update (λ-. (2 * ´N))) ∈ ⌜´N = 10⌟⌟} ´N := 2 * ´N {⌜´N = 10⌟⌟}
by (rule assign)
```

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

```
lemma ⊢ {⌜True⌟} ´N := 10 {⌜´N = 10⌟⌟}
by hoare
```

```
lemma ⊢ {⌜2 * ´N = 10⌟⌟} ´N := 2 * ´N {⌜´N = 10⌟⌟}
by hoare
```

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

```
lemma ⊢ {⌜´N + 1 = a + 1⌟⌟} ´N := ´N + 1 {⌜´N = a + 1⌟⌟}
by hoare
```

```
lemma ⊢ {⌜´N = a⌟⌟} ´N := ´N + 1 {⌜´N = a + 1⌟⌟}
by hoare simp
```

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

```
lemma ⊢ {⌜True⌟} ´M := a; ´N := b {⌜´M = a ∧ ´N = b⌟⌟}
by hoare
```

```
lemma
⊢ {⌜´M = a ∧ ´N = b⌟⌟}
´I := ´M; ´M := ´N; ´N := ´I
⌜´M = b ∧ ´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 ⊢ {⌜´N = a⌟⌟} ´N := ´N {⌜´N = a⌟⌟}
by hoare
```

```
lemma ⊢ {⌜´x = a⌟⌟} ´x := ´x {⌜´x = a⌟⌟}
```

;
lemma
Valid \( \{ s. \, x \, s = a \} \) (Basic \( (\lambda s. \, x\text{-update} \, (x \, s) \, s) \) \( \{ s. \, x \, s = n \} \))

— same statement without concrete syntax

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

lemma \( \vdash \{ \{ \, M = \, \mathbf{N} \} \, \rightarrow \, M := \, M + 1 \} \, \{ \, M \neq \, \mathbf{N} \} \)

proof –
  have \( \{ \, M = \, \mathbf{N} \} \subseteq \{ \, M + 1 \neq \, \mathbf{N} \} \)
    by auto
  also have \( \vdash \ldots \, M := \, M + 1 \} \, \{ \, M \neq \, \mathbf{N} \} \)
    by hoare
finally show \( \text{thesis} \).
qed

10.3 Multiplication by addition

We now do some basic examples of actual \textsc{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
\( \vdash \{ \{ \, M = \, \mathbf{N} \} \, \rightarrow \, M := \, M + 1 \} \, \{ \, M \neq \, \mathbf{N} \} \)

proof –
  let \( \vdash \ldots \text{while} - = \text{thesis} \)
  let \( \{ \, ?\text{inv} \} = \{ \, S = \, M \ast b \} \)
  have \( \{ \, M = 0 \wedge \, S = 0 \} \subseteq \{ \, ?\text{inv} \} \) by auto
  also have \( \vdash \ldots \text{while} \} \, \{ \, ?\text{inv} \wedge \neg (\, M \neq \, a) \} \)

37
proof
let \( ?c = \`S := \`S + b; \`M := \`M + 1 \)
have \( \{ \langle \`\text{inv} \land \`M \neq a \rangle \subseteq \{ \`S + b = (\`M + 1) \ast b \} \} \)
by auto
also have \( \vdash \ldots \langle \`\text{inv} \rangle \) by hoare
finally show \( \vdash \langle \`\text{inv} \land \`M \neq a \rangle \) \( ?c \) \( \langle \`\text{inv} \rangle \).
qed
also have \( \ldots \subseteq \{ \`S = a \ast b \} \) by auto
finally show \( \text{thesis} \).
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
\( \vdash \{ \`M = 0 \land \`S = 0 \} \)
\( \text{WHILE} \`M \neq a \)
\( \text{INV} \{ \`S = \`M \ast b \} \)
\( \text{DO} \`S := \`S + b; \`M := \`M + 1 \text{ OD} \)
\( \{ \`S = a \ast b \} \)
by hoare auto

10.4 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
\( \vdash \{ \text{True} \} \)
\( \`S := 0; \`I := 1; \text{WHILE} \`I \neq n \)
\( \text{DO} \`S := \`S + \`I; \`I := \`I + 1 \text{ OD} \)
\( \{ \`S = (\sum_{j<n.} j) \} \)
(is \( \vdash - (; \`\text{while} ; -) \))

proof –
let \( \text{?sum} = \lambda k::\text{nat}. \sum_{j<k.} j \)
let \( \text{?inv} = \lambda s i::\text{nat}. s = \text{?sum} i \)

have \( \vdash \{ \text{True} \} \`S := 0; \`I := 1 \} \langle \`\text{inv} \ \`S \ \`I \rangle \)

proof –
have \( \text{True} \rightarrow 0 = ?\text{sum} \ 1 \)
by \( \text{simp} \)
also have \( \vdash \{ \ldots \} \ S := 0; \ I := 1 \ \{ ?\text{inv} \ S \ I \} \)
by \( \text{hoare} \)
finally show \( ?\text{thesis} \).
qed
also have \( \vdash \ldots \ \{ ?\text{sum} \ S \ I \} \)
proof
let \( ?\text{body} = \{ S := S + I; \ I := I + 1 \} \)
\( \vdash \{ S + I = ?\text{sum} (I + 1) \} \ ?\text{body} \ ?\text{inv} \ S \ I \)
by \( \text{hoare} \)
finally show \( \{ ?\text{inv} \ S \ I \ \{ I \neq n \} \} \ ?\text{body} \ ?\text{inv} \ S \ I \).
qed
also have \( \bigwedge s \ i. \ s = ?\text{sum} \ i \ \{ I \neq n \} \rightarrow s = ?\text{sum} \ n \)
by \( \text{simp} \)
finally show \( ?\text{thesis} \).
qed

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

\textbf{theorem}

\( \vdash \{ \text{True} \} \)
\( \{ S := 0; \ I := 1; \} \)
\( \text{WHILE} \ \{ I \neq n \} \)
\( \text{INV} \ \{ S = (\sum j < I. \ j) \} \)
\( \text{DO} \)
\( \{ S := S + I; \ I := I + 1 \} \)
\( \text{OD} \)
\( \{ S = (\sum j < n. \ j) \} \)
proof
let \( ?\text{sum} = \lambda k::\text{nat}. \ \sum j < k. \ j \)
let \( ?\text{inv} = \lambda s i::\text{nat}. \ s = ?\text{sum} \ i \)
show \( ?\text{thesis} \)
proof \( \text{hoare} \)
show \( ?\text{inv} \ 0 \ 1 \) by \( \text{simp} \)
next
fix \( s \ i \)
assume \( ?\text{inv} \ s \ i \ \{ I \neq n \} \)
then show \( ?\text{inv} (s + i) (i + 1) \) by \( \text{simp} \)
next
fix \( s \ i \)
assume \( ?\text{inv} \ s \ i \ \{ I \neq n \} \)
then show \( s = ?\text{sum} \ n \) by \( \text{simp} \)
qed
qed

39
Certainly, this proof may be done fully automatic as well, provided that the invariant is given beforehand.

\[ \begin{array}{c}
\text{theorem} \\
\vdash \{ \text{True} \}
\end{array} \]

\[ \begin{array}{c}
\`S := 0; \`I := 1; \\
\text{WHILE} \ `I \neq n \\
\text{INV} \ \{ \`S = (\sum j < I. j) \}
\end{array} \]

\[ \begin{array}{c}
\text{DO} \\
\`S := `S + `I; \\
`I := `I + 1
\end{array} \]

\[ \{ \`S = (\sum j < n. j) \} \]

by hoare auto

\section{10.5 Time}

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

\begin{itemize}
\item \textbf{record} \textit{tstate} = \textit{time} :: \textit{nat}
\item \textbf{type-synonym} `a time = (\textit{time} :: \textit{nat}, \ldots :: `a)
\item \textbf{primrec} \textit{timeit} :: `a time \Rightarrow `a time \Rightarrow `a time com
\item \textbf{where}
\begin{itemize}
\item \textit{timeit} (\textit{Basic} \textit{f}) = (\textit{Basic} \textit{f}; \textit{Basic}(\lambda s. s(\textit{time} := \textit{Suc} (\textit{time} s))))
\item \textit{timeit} (c1; c2) = (\textit{timeit} c1; \textit{timeit} c2)
\item \textit{timeit} (\textit{Cond} \textit{b} c1 c2) = \textit{Cond} \textit{b} (\textit{timeit} c1) (\textit{timeit} c2)
\item \textit{timeit} (\textit{While} \textit{b} iv c) = \textit{While} \textit{b} iv (\textit{timeit} c)
\end{itemize}
\item \textbf{record} \textit{tvars} = \textit{tstate} +
\item \textit{I} :: \textit{nat}
\item \textit{J} :: \textit{nat}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \textit{lem} : (0 :: \textit{nat}) < \textit{n} \implies \textit{n} + \textit{n} \leq \textit{Suc} (\textit{n} * \textit{n})
\item by (induct \textit{n}) simp-all
\end{itemize}

\begin{itemize}
\item \textbf{lemma}
\item \textbf{where}
\item \textbf{do}
\item \textbf{end}
\item \textbf{by}
\end{itemize}

40
\[ 2 \times \text{time} = i \times i + 5 \times i \]

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

end

11  Textbook-style reasoning: the Knaster-Tarski Theorem

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

11.1 Prose version

According to the textbook [1, pages 93–94], the Knaster-Tarski fixpoint theorem is as follows.\(^9\)

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

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

11.2 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.

\(^9\)We have dualized the argument, and tuned the notation a little bit.
theorem Knaster-Tarski:
fixes f :: 'a::complete-lattice ⇒ 'a
assumes mono f
shows ∃ a. f a = a
proof
  let ?H = {a. f u ≤ u}
  let ?a = ∩ ?H
  show f ?a = ?a
  proof
    
    fix x
    assume x ∈ ?H
    then have ?a ≤ x by (rule Inf-lower)
    with (mono f) have f ?a ≤ f x ..
    also from ⟨x ∈ ?H⟩ have ... ≤ x ..
    finally have f ?a ≤ x .
  
  then have f ?a ≤ ?a by (rule Inf-greatest)
  
  also presume ... ≤ f ?a
  finally (order-antisym) show ?thesis .
  
  from ⟨mono f⟩ and ⟨f ?a ≤ ?a⟩ have f (f ?a) ≤ f ?a ..
  then have f ?a ∈ ?H ..
  then show ?a ≤ f ?a by (rule Inf-lower)
qed
qed

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

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

theorem Knaster-Tarski':
fixes f :: 'a::complete-lattice ⇒ 'a
assumes mono f
shows ∃ a. f a = a
proof
  let ?H = {a. f u ≤ u}
  let ?a = ∩ ?H
  show f ?a = ?a
  proof (rule order-antisym)
    show f ?a ≤ ?a
    proof (rule Inf-greatest)
      fix x
  qed
qed

assume \( x \in \mathcal{H} \)
then have \( ?a \leq x \) by (rule Inf-lower)
with (mono f) have \( f ?a \leq f x \).
also from \( \langle x \in \mathcal{H} \rangle \) have \( \ldots \leq x \).
finally show \( f ?a \leq x \).
qed

deprecated

show \( ?a \leq f ?a \)
proof (rule Inf-lower)
from (mono f) and (\( f ?a \leq ?a \)) have \( f (f ?a) \leq f ?a \).
then show \( f ?a \in \mathcal{H} \).
qed

qed

end

12 The Mutilated Checker Board Problem

theory Mutilated-Checkerboard
imports Main
begin

The Mutilated Checker Board Problem, formalized inductively. See [7] for
the original tactic script version.

12.1 Tilings

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

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)

43
show \((a \cup t) \cup u \in \mathcal{T}\)
proof -
have \(a \cup (t \cup u) \in \mathcal{T}\)
  using \((a \in A)\)
proof (rule tiling.Un)
  from \((a \cup t) \cap u = \emptyset\) have \(t \cap u = \emptyset\) by blast
  then show \(t \cup u \in \mathcal{T}\) by (rule Un)
  from \((a \subseteq t)\) and \((a \cup t) \cap u = \emptyset\)
  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

12.2 Basic properties of “below”
definition below :: \(\text{nat} \Rightarrow \text{nat set}\)
  where below \(n = \{i. \ i < n\}\)
lemma below-less-iff [iff]: \(i \in \text{below} \ k \iff i < k\)
  by (simp add: below-def)
lemma below-0: \(\text{below} \ 0 = \emptyset\)
  by (simp add: below-def)
lemma Sigma-Suc1: \(m = n + 1 \Rightarrow \text{below} \ m \times B = ((n) \times B) \cup (\text{below} \ n \times B)\)
  by (simp add: below-def less-Suc-eq) blast
lemma Sigma-Suc2:
  \(m = n + 2 \Rightarrow \text{above} \ m = (A \times \{n\}) \cup (A \times \{n + 1\}) \cup (A \times \text{below} \ n)\)
  by (auto simp add: below-def)
lemmas Sigma-Suc = Sigma-Suc1 Sigma-Suc2

12.3 Basic properties of “evnodd”
definition evnodd :: \((\text{nat} \times \text{nat}) \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat})\)
  where evnodd \(A \ b = \{ \cdot i, j. \ (i + j) \ mod \ 2 = b\}\)
lemma evnodd-iff: \((i, j) \in \text{evnodd} \ A \ b \iff (i, j) \in A \ \land \ (i + j) \ mod \ 2 = b\)
  by (simp add: evnodd-def)
lemma evnodd-subset: \(\text{evnodd} \ A \ b \subseteq A\)
  unfolding evnodd-def by (rule Int-lower1)
lemma \textit{evnoddD}: \( x \in \text{evnodd} \ A \ b \implies x \in A \)
by (rule subsetD) (rule evnodd-subset)

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

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

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

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

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

\textbf{12.4 Dominoes}

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

\textbf{lemma} \textit{dominoes-tile-row}:
\( \{i\} \times \text{below} \ (2 \times n) \in \text{tiling domino} \)
(is \( B \ n \in \ ?T \))
\textbf{proof} (induct n)
\textbf{case} \( 0 \)
\textbf{show} \( ?\text{case} \) by (simp add: below-0 tiling.empty)
\textbf{next}
\textbf{case} \( \text{Suc} \ n \)
\textbf{let} \( ?a = \{i\} \times \{2 \times n + 1\} \cup \{i\} \times \{2 \times n\} \)
\textbf{have} \( ?B \ (\text{Suc} \ n) = ?a \cup ?B \ n \)
\textbf{by} (auto simp add: Sigma-Suc Un-assoc)
\textbf{also have} \( \ldots \in \ ?T \)
\textbf{proof} (rule tiling.Un)
\textbf{have} \( \{i, 2 \times n\}, \{i, 2 \times n + 1\} \in \text{domino} \)
\textbf{by} (rule domino.horiz)
\textbf{also have} \( \{i, 2 \times n\}, \{i, 2 \times n + 1\} = ?a \text{ by blast} \)
\textbf{finally show} \( \ldots \in \text{domino} \).
\textbf{show} \( ?B \ n \in \ ?T \) by (rule Suc)
\textbf{show} \( ?a \subseteq - ?B \ n \) by blast
\textbf{qed}
\textbf{finally show} \( ?\text{case} \).
\textbf{qed}
lemma dominoes-tile-matrix:
below m × below (2 * n) ∈ tiling domino
(is ?B m ∈ ?T)
proof (induct m)
case 0
show ?case by (simp add: below-0 tiling.empty)
next
case (Suc m)
let ?t = {?m} × below (2 * n)
have ?B (Suc m) = ?t ∪ ?B m by (simp add: Sigma-Suc)
also have ... ∈ ?T
proof (rule tiling-Un)
  show ?t ∈ ?T by (rule dominoes-tile-row)
  show ?B m ∈ ?T by (rule Suc)
  show ?t ∩ ?B m = {} by blast
qed
finally show ?case .
qed

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

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

12.5  Tilings of dominoes

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

lemma \textit{tiling-domino-01}:
  assumes $t. t \in \text{tiling domino}$ (is $t \in \bar{T}$)
  shows $\text{card (evnodd } t 0) = \text{card (evnodd } t 1)$
  using $t$
proof \textit{induct}
  case \text{empty}
  show $\text{?case}$ by (simp add: evnodd-def)
next
case $\text{Un } a \ t$
  let $\bar{e} = \text{evnodd}$
  note hyp = $\langle \text{card (\bar{e} } t 0) = \text{card (\bar{e} } t 1) \rangle$
  and at = $(a \subseteq - t)$
  have \textit{card-suc}:
    $\forall b. b < 2 \implies \text{card (\bar{e} } (a \cup t) b) = \text{Suc (\text{card (\bar{e} } t b)})$
    proof
      fix $b :: \text{nat}$
      assume $b < 2$
      have $\bar{e} (a \cup t) b = \bar{e} a b \cup \bar{e} t b$ by (rule evnodd-Un)
      also obtain $i j$ where $e. \bar{e} a b = \{ (i, j) \}$
        proof
          from $(a \in \text{domino})$ and $(b < 2)$
          have $\exists i j. \bar{e} a b = \{ (i, j) \}$ by (rule domino-singleton)
          then show $\text{thesis}$ by (blast intro: that)
        qed
      also have $\ldots \cup \bar{e} t b = \text{insert (i, j) (\bar{e} t b)}$ by simp
      also have $\text{card } \ldots = \text{Suc (\text{card (\bar{e} } t b))}$
        proof (rule card-insert-disjoint)
          from $(t \in \text{tiling domino})$ have $\text{finite } t$
            by (rule tiling-domino-finite)
          then show $\text{finite (\bar{e} t b)}$
            by (rule evnodd-finite)
          from $e$ have $(i, j) \in \bar{e} a b$ by simp
            with at show $(i, j) \notin \bar{e} t b$ by (blast dest: evnoddD)
        qed
      finally show $\text{thesis } b$
    qed
  from $\bar{e}$ have $(i, j) \in \bar{e} a b$ by simp
    with at show $(i, j) \notin \bar{e} t b$ by (blast dest: evnoddD)
  qed
finally show $\text{?case }$.
qed
12.6 Main theorem

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

theorem mutil-not-tiling; mutilated-board m n ⋄ tiling domino
proof (unfold mutilated-board-def)
let ?T = tiling domino
let ?t = below (2 * (m + 1)) × below (2 * (n + 1))
let ?t' = ?t − {(0, 0)}
let ?t'' = ?t' − {(2 * m + 1, 2 * n + 1)}

show ?t'' ⋄ ?T
proof
  have t: ?t ∈ ?T by (rule dominoes-tile-matrix)
  assume t'': ?t'' ∈ ?T
  let ?e = evnodd
  have fin: finite (?e ?t 0)
    by (rule evnodd-finite, rule tiling-domino-finite, rule t)

  note [simp] = evnodd-iff evnodd-empty evnodd-insert evnodd-Diff
  have card (?e ?t'' 0) < card (?e ?t' 0)
  proof
    have (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 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)
48
by (rule tiling-domino-01 [symmetric])
finally have \ldots < \ldots then show False ..
qed
qed
end

13 Nested datatypes

theory Nested-Datatype
imports Main
begin

13.1 Terms and substitution
datatype ('a, 'b) term =
  Var 'a
| App 'b ('a, 'b) term list

primrec subst-term :: ('a ⇒ ('a, 'b) term ⇒ ('a, 'b) term)
  and subst-term-list :: ('a ⇒ ('a, 'b) term ⇒ ('a, 'b) term list ⇒ ('a, 'b) term list)
  list
where
  subst-term f (Var a) = f a
| subst-term f (App b ts) = App b (subst-term-list f ts)
| subst-term-list f [] = []
| subst-term-list f (t # ts) = subst-term f t # subst-term-list f ts

lemmas subst-simps = subst-term.simps subst-term-list.simps

A simple lemma about composition of substitutions.

lemma
  subst-term (subst-term f1 ∘ f2) t =
  subst-term f1 (subst-term f2 t)
  and
  subst-term-list (subst-term f1 ∘ f2) ts =
  subst-term-list f1 (subst-term-list f2 ts)
by (induct t and ts rule: subst-term.induct subst-term-list.induct) simp-all

lemma subst-term (subst-term f1 ∘ f2) t = subst-term f1 (subst-term f2 t)
proof –
  let ?P t = ?thesis
  let ?Q = λts. subst-term-list (subst-term f1 ∘ f2) ts =
    subst-term-list f1 (subst-term-list f2 ts)
  show ?thesis
  proof (induct t rule: subst-term.induct)
    fix a show ?P (Var a) by simp
  next

49
13.2 Alternative induction

lemma subst-term (subst-term f1 \circ f2) t = subst-term f1 (subst-term f2 t)
proof (induct t rule: term.induct)
case (Var a)
  show ?case by (simp add: o-def)
next
case (App b ts)
  then show ?case by (induct ts) simp-all
qed

end

14 Peirce’s Law

theory Peirce
imports Main
begin

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

The first proof is again a well-balanced combination of plain backward and forward reasoning. The actual classical step is where the negated goal may be introduced as additional assumption. This eventually leads to a contradiction. \footnote{The rule involved there is negation elimination; it holds in intuitionistic logic as well.}

theorem ((A \rightarrow B) \rightarrow A) \rightarrow A
proof
  assume (A \rightarrow B) \rightarrow A
  show A
  proof (rule classical)
    assume \neg A
    have A \rightarrow B
    proof

assume \( A \)
with \( \neg A \) show \( B \) by contradiction
qed
with \((A \rightarrow B) \rightarrow A\) show \( A \)
qed
qed

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

Technically speaking, whenever some goal is solved by \texttt{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 \texttt{assume}) are solved immediately.

\begin{verbatim}
theorem \(((A \rightarrow B) \rightarrow A) \rightarrow A\)
proof
assume \( (A \rightarrow B) \rightarrow A \)
show \( A \)
proof (rule classical)
  presume \( A \rightarrow B \)
  with \((A \rightarrow B) \rightarrow A\) show \( A \)
next
assume \( \neg A \)
show \( A \rightarrow B \)
proof
assume \( A \)
with \( \neg A \) show \( B \) by contradiction
qed
qed
qed
\end{verbatim}

Note that the goals stemming from weak assumptions may be even left until \texttt{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 \textit{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 backtrack-
15 An old chestnut

theory Puzzle
imports Main
begin

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

theorem
assumes f-ax: \( \forall n. f(f(n)) < f(Suc(n)) \)
sows f n = n
proof (rule order-antisym)
{ 
  fix \( n \) show \( n \leq f(n) \)
proof (induct \( f(n) \) arbitrary: \( n \) rule: less-induct)
  case less
  show \( n \leq f(n) \)
proof (cases \( n \))
    case (Suc m)
      from f-ax have \( f(f(m)) < f(n) \) by (simp only: Suc)
      with less have \( f(m) \leq f(f(m)) \).
      also from f-ax have \( \ldots < f(n) \) by (simp only: Suc)
      finally have \( f(m) < f(n) \).
      with less have \( m \leq f(m) \).
      also note \( \ldots < f(n) \)
      finally have \( m < f(n) \).
      then have \( n \leq f(n) \) by (simp only: Suc)
      then show \( \text{thesis} \).
  next
    case 0
    then show \( \text{thesis} \) by simp
  qed
  qed
} note ge = this

{ 
  fix \( m \) \( n \) :: nat
  assume \( m \leq n \)
  then have \( f(m) \leq f(n) \)
proof (induct \( n \))
  case 0
  then have \( m = 0 \) by simp

\footnote{A question from “Bundeswettbewerb Mathematik”. Original pen-and-paper proof due to Herbert Ehler; Isabelle tactic script by Tobias Nipkow.}
then show \( ?\) case by simp
next
case (Suc \( n \))
from Suc.prems show \( f \cdot m \leq f \cdot (Suc \cdot n) \)
proof (rule le-SucE)
assume \( m \leq n \)
with Suc.hyps have \( f \cdot m \leq f \cdot n \).
also from ge f-ax have \( \ldots \leq f \cdot (Suc \cdot n) \)
by (rule le-less-trans)
finally show \( ?\) thesis by simp
next
assume \( m = Suc \cdot n \)
then show \( ?\) thesis by simp
qed
qed

} note mono = this

show \( f \cdot n \leq n \)
proof —

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

end

16 Summing natural numbers

theory Summation
imports Main
begin

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

16.1 Summation laws

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

theorem sum-of-naturals:
\[ 2 \times (\sum_{i::\text{nat}=0\ldots n} i) = n \times (n + 1) \]

(is \( \varphi \) \( n \) is \( \varepsilon \) \( n = - \))  
proof (induct \( n \))  
show \( \varphi \) \( 0 \) by simp  
next  
fix \( n \) have \( \varepsilon \) \( n + 1 \) = \( \varepsilon \) \( n \) + 2 \times (n + 1)  
  by simp  
also assume \( \varepsilon \) \( n = n \times (n + 1) \)  
also have \( \ldots + 2 \times (n + 1) = (n + 1) \times (n + 2) \)  
  by simp  
finally show \( \varphi \) \( (\text{Suc} \ n) \)  
  by simp  
qed

The above proof is a typical instance of mathematical induction. The main statement is viewed as some \( \varphi \) \( n \) that is split by the induction method into base case \( \varphi \) \( 0 \), and step case \( \varphi \) \( n \Rightarrow \varphi \) \( \text{Suc} \ n \) for arbitrary \( n \).

The step case is established by a short calculation in forward manner. Starting from the left-hand side \( \varepsilon \) \( (n + 1) \) 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 \( \varepsilon \) \( n = n \times (n + 1) \) is introduced in order to replace a certain subterm. So the “transitivity” rule involved here is actual substitution. Also note how the occurrence of “\( \ldots \)” in the subsequent step documents the position where the right-hand side of the hypothesis got filled in.

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

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

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

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

theorem sum-of-odds:  
\( (\sum_{i::\text{nat}=0\ldots n} 2 \times i + 1) = n \times \text{Suc} \ (\text{Suc} \ 0) \)  
(is \( \varphi \) \( n \) is \( \varepsilon \) \( n = - \))  
proof (induct \( n \))  
show \( \varphi \) \( 0 \) by simp
next
fix $n$
have $?S (n + 1) = ?S n + 2 \cdot n + 1$
  by simp
also assume $?S n = n \cdot Suc (Suc 0)$
also have \ldots $+ 2 \cdot n + 1 = (n + 1) \cdot Suc (Suc 0)$
  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:
\[ 6 \cdot \sum_{i::nat=0..n} i \cdot Suc (Suc 0) = n \cdot (n + 1) \cdot (2 \cdot n + 1) \]
(is $\sim P n \sim S n = -$)
proof (induct $n$)
show $?P \ 0 \ by \ simp$
next
fix $n$
have $?S (n + 1) = ?S n + 6 \cdot (n + 1) \cdot Suc (Suc 0)$
  by (simp add: distrib)
also assume $?S n = n \cdot (n + 1) \cdot (2 \cdot n + 1)$
also have \ldots $+ 6 \cdot (n + 1) \cdot Suc (Suc 0) =$
\quad $(n + 1) \cdot (n + 2) \cdot (2 \cdot (n + 1) + 1)$
  by (simp add: distrib)
finally show $?P (Suc n)$
  by simp
qed

theorem sum-of-cubes:
\[ 4 \cdot \sum_{i::nat=0..n} i \cdot 3 = (n \cdot (n + 1)) \cdot Suc (Suc 0) \]
(is $\sim P n \sim S n = -$)
proof (induct $n$)
show $?P \ 0 \ by \ (simp \ add: \ power-eq-if)$
next
fix $n$
have $?S (n + 1) = ?S n + 4 \cdot (n + 1) \cdot 3$
  by (simp add: power-eq-if distrib)
also assume $?S n = (n \cdot (n + 1)) \cdot Suc (Suc 0)$
also have \ldots $+ 4 \cdot (n + 1) \cdot 3 = ((n + 1) \cdot ((n + 1) + 1)) \cdot Suc (Suc 0)$
  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 struc-
tured 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 \texttt{simp} or \texttt{auto} should normally be never used as initial proof methods with a nested sub-proof to address the automatically produced situation, but only as terminal ones to solve sub-problems.

\end

References


