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

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.

\begin{verbatim}
lemma A \implies A
proof
  assume A
  show A by fact+
qed
\end{verbatim}

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.

\begin{verbatim}
lemma A \implies A
proof
  by rule
qed
\end{verbatim}

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

\begin{verbatim}
lemma A \implies A ..
\end{verbatim}

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

\begin{verbatim}
lemma A \implies B \implies A
proof (intro impl)
  assume A
  show A by fact
qed
\end{verbatim}

Again, the body may be collapsed.

\begin{verbatim}
lemma A \implies B \implies A
\end{verbatim}

\(^1\)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 \(A \land B \rightarrow B \land A\).

The first version is purely backward.

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

Above, the 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.

\begin{verbatim}
lemma A \land B \rightarrow B \land A
proof
  assume A \land B
  show B \land A
  proof
    from \langle A \land B \rangle show B ..
    from \langle A \land B \rangle show A ..
  qed
qed
\end{verbatim}
In the next version, we move the forward step one level upwards. Forward-chaining from the most recent facts is indicated by the `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 `conjE` rule, including the repeated goal proposition that is abbreviated as `?thesis` below.

**lemma** $A \land B \rightarrow B \land A$

**proof**

assume $A \land B$
then show $B \land A$

**proof**

assume $B \land A$
then show `?thesis` .. — rule `conjI` of $B \land A$

qed

**qed**

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

**lemma** $A \land B \rightarrow B \land A$

**proof**

assume $A \land B$
from $(A \land B)$ have $A$ ..
from $(A \land B)$ have $B$ ..
from $(B) (A)$ show $B \land A$ ..

**qed**

**qed**

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

**lemma** $A \land B \rightarrow B \land A$

**proof**

{-
assume $A \land B$
from $(A \land B)$ have $A$ ..
from $(A \land B)$ have $B$ ..
from $(B) (A)$ have $B \land A$ ..
}
then show `?thesis` .. — rule `impl`

qed

**qed**

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

The general lesson learned here is that good proof style would achieve just the **right** balance of top-down backward decomposition, and bottom-up 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.

\textbf{lemma} \ A \land B \rightarrow B \land A \\
\textbf{proof} \\
\text{assume} \ A \land B \\
\text{then show} \ B \land A \\
\textbf{proof} \\
\text{assume} \ B \ A \\
\text{then show} \ ?thesis .. \\
\textbf{qed} \\
\textbf{qed}

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

\textbf{lemma} \ P \lor P \rightarrow P \\
\textbf{proof} \\
\text{assume} \ P \lor P \\
\text{then show} \ P \\
\textbf{proof} \\
\text{-- rule \textit{disjE}: } A \lor B \rightarrow C \\
\text{assume} \ P \text{ show} \ P \ by \ mathit{fact} \\
\textbf{next} \\
\text{assume} \ P \text{ show} \ P \ by \ mathit{fact} \\
\textbf{qed} \\
\textbf{qed}

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 `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 *not* require `next` to separate the subsequent step case.

In our example the situation is even simpler, since the two cases actually coincide. Consequently the proof may be rephrased as follows.

```isar
case

lemma \( P \lor P \longrightarrow P \)
proof
assume \( P \lor P \)
then show \( P \)
proof
assume \( P \)
show \( P \) by fact
show \( P \) by fact
qed
qed
```

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

```isar

lemma \( P \lor P \longrightarrow P \)
proof
assume \( P \lor P \)
then show \( P \)
proof
(rule exE) — rule exE:
\[
\exists x. A(x) \quad B
\]
fix \( a \)
```

### 1.3.2 A quantifier proof

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

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

```isar

lemma \( (\exists x. P(fx)) \longrightarrow (\exists y. Py) \)
proof
assume \( \exists x. P(fx) \)
then show \( \exists y. Py \)
proof
(rule exE)
\[
\exists x. A(x) \quad B
\]
fix \( a \)
```
assume \( P(f \ a) \) (is \( P \ ?\text{witness} \))
then show \( \text{thesis} \) by \( \text{rule exI [of } P \ ?\text{witness}] \)
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.

lemma \((\exists x. \ P(f \ x)) \rightarrow (\exists y. \ P y)\)
proof
\[ \begin{align*}
\text{assume } & \exists x. \ P (f x) \\
\text{then show } & \exists y. \ P y \\
\text{proof } & \\
\text{fix } a \\
\text{assume } & P (f a) \\
\text{then show } & \text{thesis ..} \\
\text{qed} \\
\text{qed}
\end{align*} \]

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.

lemma \((\exists x. \ P(f \ x)) \rightarrow (\exists y. \ P y)\)
proof
\[ \begin{align*}
\text{assume } & \exists x. \ P (f x) \\
\text{then obtain } & a \text{ where } P (f a) .. \\
\text{then show } & \exists y. \ P y .. \\
\text{qed}
\end{align*} \]

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 \texttt{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.

theorem \texttt{conjE}: \( A \wedge B \Rightarrow (A \Rightarrow B \Rightarrow C) \Rightarrow C \)
proof −
\[ \begin{align*}
\text{assume } & A \wedge B \\
\text{assume } & r: A \Rightarrow B \Rightarrow C
\end{align*} \]
show \( C \)

proof (rule r)
  show \( A \) by (rule conjunct1) fact
  show \( B \) by (rule conjunct2) fact
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 \( ?S \notin \text{ range } f \)
  proof
    assume \( ?S \in \text{ range } f \)
    then obtain \( y \) where \( ?S = f y \) ..
    then show \( \text{False} \)
      proof (rule equalityCE)
        assume \( y \in f y \)
        assume \( y \in ?S \)
        then have \( y \notin f y \) ..
        with \( \langle y : f y \rangle \) show \( \text{thesis by contradiction} \)
      next
        assume \( y \notin ?S \)
        assume \( y \notin f y \)
        then have \( y \in ?S \) ..
        with \( \langle y \notin ?S \rangle \) show \( \text{thesis by contradiction} \)
      qed
    qed
  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.

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

theory Drinker
imports Main
begin

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

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

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

```isar
theorem Drinker’s-Principle: \( \exists x. \text{drunk } x \rightarrow (\forall x. \text{drunk } x) \)
proof cases
  fix \( a \) assume \( \forall x. \text{drunk } x \)
  then have \( \text{drunk } a \rightarrow (\forall x. \text{drunk } x) \) ..
  then show \(?thesis ..\)
next
  assume \( \neg (\forall x. \text{drunk } x) \)
```

11
then have \( \exists x. \neg \text{drunk } x \) by (rule de-Morgan)
then obtain \( a \) where \( a: \neg \text{drunk } a \)
have \( \text{drunk } a \to (\forall x. \text{drunk } x) \)
proof
  assume \( \text{drunk } a \)
  with \( a \) show \( \forall x. \text{drunk } x \) by contradiction
qed
then show \( \text{thesis} \)
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 ('adr, '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.

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

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

```hs
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.

```hs
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.

```hs
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 \( \text{case by simp} \)
next
  case Apply
    from Cons show \( \text{case by simp} \)
qed
qed

theorem correctness: execute (compile e) env = eval e env
proof -
  have \( \forall \text{stack}, \text{exec (compile e) stack env = eval e env \# stack} \)
  proof (induct e)
    case Variable
      show \( \text{case by simp} \)
    next
      case Constant
        show \( \text{case by simp} \)
    next
      case Binop
        then show \( \text{case by (simp add: exec-append)} \)
        qed
        then show \( \text{thesis by (simp add: execute-def)} \)
        qed
      qed

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

lemma exec-append‘:
  exec (xs @ ys) stack env = exec ys (exec xs stack env) env
proof (induct xs arbitrary: stack)
  case (Nil s)
  have \( \text{exec ([] @ ys) s env = exec ys s env} \)
    by simp
  also have \( \ldots = \text{exec ys (exec [] s env)} env \)
    by simp
  finally show \( \text{case} \).
next
  case (Cons x xs s)
  show \( \text{case} \)
  proof (induct x)
    case (Const val)
    have \( \text{exec ((Const val \# xs) @ ys) s env = exec (Const val \# xs \# ys) s env} \)
      by simp
    also have \( \ldots = \text{exec (xs @ ys) (val \# s) env} \)
      by simp
    also from Cons have \( \ldots = \text{exec ys (exec xs (val \# s) env)} env \).
    also have \( \ldots = \text{exec ys (exec (Const val \# xs) s env)} env \)
      by simp

14
finally show \(?\)case .
next
case (Load adr)
from Cons show \(?\)case
by simp — same as above
next
case (Apply fn)
have exec ((Apply fn \# xs) @ ys) s env = exec (Apply fn \# xs @ ys) s env by simp
also have ... = exec (xs @ ys) (fn (hd s) (hd (tl s)) \# (tl (tl s))) env
  by simp
also from Cons have ... =
  exec ys (exec xs (fn (hd s) (hd (tl s)) \# tl (tl s)) env) env .
also have ... = exec ys (exec (Apply fn \# xs) s env) env
  by simp
finally show \(?\)case .
qed
qed

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

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

end

5  Fib and Gcd commute

theory Fibonacci
imports ../Number-Theory/Primes
begin\footnote{Isar version by Gertrud Bauer. Original tactic script by Larry Paulson. A few proofs of laws taken from [2].}
declare One-nat-def [simp]

5.1  Fibonacci numbers

fun fib :: nat \Rightarrow nat where
  fib 0 = 0
| fib (Suc 0) = 1
| fib (Suc (Suc x)) = fib x + fib (Suc x)

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

Alternative induction rule.

theorem fib-induct:
  fixes n :: nat
  shows P 0 \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 gcd commute

A few laws taken from [2].

lemma fib-add:
\[ \text{fib} \ (n + k + 1) = \text{fib} \ (k + 1) \ast \text{fib} \ (n + 1) + \text{fib} \ k \ast \text{fib} \ n \]

(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)

\[ = \text{fib} \ (n + k + 1) + \text{fib} \ (n + 1 + k + 1) \text{ by simp} \]

also assume \ fib \ (n + k + 1)

\[ = \text{fib} \ (k + 1) \ast \text{fib} \ (n + 1) + \text{fib} \ k \ast \text{fib} \ n \]

(is \ - \ = \ ?R1)

also assume \ fib \ (n + 1 + k + 1)

\[ = \text{fib} \ (k + 1) \ast \text{fib} \ (n + 1 + 1) + \text{fib} \ k \ast \text{fib} \ (n + 1) \]

(is \ - \ = \ ?R2)

also have \ ?R1 \ + \ ?R2

\[ = \text{fib} \ (k + 1) \ast \text{fib} \ (n + 2 + 1) + \text{fib} \ k \ast \text{fib} \ (n + 2) \]

by (simp add: add-mult-distrib2)

finally show \ ?P \ (n + 2).

qed

lemma \ gcd-fib-Suc-eq-1: \ \gcd \ (\text{fib} \ n) \ (\text{fib} \ (n + 1)) = 1 \text{ (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) = \text{fib} \ (n + 1) + \text{fib} \ (n + 2)

by simp

also have \ \ldots = \text{fib} \ (n + 2) + \text{fib} \ (n + 1)

by simp

also have \ \gcd \ (\text{fib} \ (n + 2)) \ \ldots = \gcd \ (\text{fib} \ (n + 2)) \ (\text{fib} \ (n + 1))

by (rule gcd-add2-nat)

also have \ \ldots = \gcd \ (\text{fib} \ (n + 1)) \ (\text{fib} \ (n + 1 + 1))

by (simp add: gcd-commute-nat)

also assume \ \ldots = 1

finally show \ ?P \ (n + 2).

qed

lemma \ gcd-mult-add: \ (0::\text{nat}) \ < \ n \Longrightarrow \ \gcd \ (n \ast \ k + m) \ n = \gcd \ m \ n

proof

assume \ 0 \ < \ n

then have \ \gcd \ (n \ast \ k + m) \ n = \gcd \ n \ (m \ mod \ n)

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

also from \ 0 \ < \ n \ have \ \ldots = \gcd \ m \ n

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

finally show \ ?thesis.

qed

lemma \ gcd-fib-add: \ \gcd \ (\text{fib} \ m) \ (\text{fib} \ (n + m)) = \gcd \ (\text{fib} \ m) \ (\text{fib} \ n)
proof (cases m)
  case 0
  then show \textit{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 \(fib \, (n + k + 1)\)
    \(= fib \, (k + 1) \times fib \, (n + 1) + fib \, k \times fib \, n\)
    by (rule fib-add)
also have \(\ldots = gcd (fib \, k \, \times fib \, n) \, (fib \, (k + 1))\)
    by (simp only: gcd-mult-cancel-nat)
also have \(\ldots = gcd (fib \, m) \, (fib \, n)\)
    using Suc by (simp add: gcd-commute-nat)
finally show \textit{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 \textit{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 \textit{?case}
    proof
      have \(n \mod m = (if \, n < m \, then \, n \, else \, (n - m) \mod m)\)
        by (rule mod-if)
      also have \(gcd \, (fib \, m) \, (fib \, \ldots) = gcd (fib \, m) \, (fib \, n)\)
        proof (cases \(n < m\))
          case True
          then show \textit{?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
    qed
  qed
\[
gcd (\text{fib } m) (\text{fib } (n - m)) \text{ by simp}
\]
also have \[\ldots = \gcd (\text{fib } m) (\text{fib } n)\]
using \(m \leq n\) by (rule gcd-fib-diff)
finally have \[\gcd (\text{fib } m) (\text{fib } ((n - m) \mod m)) = \gcd (\text{fib } m) (\text{fib } n)\]
with False show ?thesis by simp
qed
finally show ?thesis.
qed

\begin{proof}
\begin{induction} m \ n \rule: \gcd\text{-nat-induct}
\end{induction}
fix \(m\)
show \(\text{fib } (\gcd m \ 0) = \gcd (\text{fib } m) (\text{fib } 0)\)
by simp
fix \(n::\text{nat}\)
assume \(n < n\)
then have \(\gcd m \ n = \gcd n \ (m \mod n)\)
by (simp add: gcd-non-0-nat)
also assume hyp: \(\text{fib } \ldots = \gcd (\text{fib } n) (\text{fib } (m \mod n))\)
also from \(n\) have \(\ldots = \gcd (\text{fib } n) (\text{fib } m)\)
by (rule gcd-fib-mod)
also have \(\ldots = \gcd (\text{fib } m) (\text{fib } n)\)
by (rule gcd-commute-nat)
finally show \(\text{fib } (\gcd m \ n) = \gcd (\text{fib } m) (\text{fib } n)\).
qed
\end{proof}

end

6 Basic group theory

theory Group
imports Main
begin

6.1 Groups and calculational reasoning

Groups over signature \((\times :: \alpha \to \alpha \to \alpha), one :: \alpha, inverse :: \alpha \to \alpha)\) 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} + one + inverse +
\begin{assumesteps}
\text{group-assoc}: \((x * y) * z = x * (y * z)\)
\text{and group-left-one}: \(1 * x = x\)
\text{and group-left-inverse}: \(\text{inverse } x * x = 1\)
\end{assumesteps}

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 \( \ldots = I \ast x \ast \text{inverse} \ x \)
    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 \( \ldots = 1 \)
    by (simp only: group-left-one)
finally show \(?thesis\).
qed

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

theorem (in group) group-right-one: \( x \ast 1 = x \)
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 \( \ldots = I \ast x \)
    by (simp only: group-right-inverse)
  also have \( \ldots = x \)
    by (simp only: group-left-one)
finally show \(?thesis\).
qed

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

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

theorem (in group) \( x \cdot 1 = x \)
proof
  have \( x \cdot 1 = x \cdot (\text{inverse } x \cdot x) \)
    by (simp only: group-left-inverse)
  
  note calculation = this
  — first calculational step: init calculation register

  have \( \ldots = x \cdot \text{inverse } x \cdot x \)
    by (simp only: group-assoc)
  
  note calculation = trans [OF calculation this]
  — general calculational step: compose with transitivity rule

  have \( \ldots = 1 \cdot x \)
    by (simp only: group-right-inverse)
  
  note calculation = trans [OF calculation this]
  — general calculational step: compose with transitivity rule

  have \( \ldots = x \)
    by (simp only: group-left-one)
  
  note calculation = trans [OF calculation this]
  — final calculational step: compose with transitivity rule 
  from calculation
  — \( \ldots \) and pick up the final result

  show \(?thesis\).
qed

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

6.2 Groups as monoids

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

class monoid = times + 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 \(\text{thesis}\).

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 –

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

The inverse operation has some further characteristic properties.

\textbf{Theorem (in group)} \textbf{group-inverse-times}: inverse \( (x \ast y) = \text{inverse } y \ast \text{inverse } x \)
\textbf{Proof (rule group-inverse-equality)}
\begin{itemize}
  \item show \( \text{inverse } y \ast \text{inverse } x \ast (x \ast y) = 1 \)
  \item proof –
    \begin{itemize}
      \item have \( \text{inverse } y \ast \text{inverse } x \ast (x \ast y) = \)
        \( \text{inverse } y \ast (\text{inverse } x \ast x) \ast y \)
        by (simp only: group-assoc)
      \item also have \( \ldots = (\text{inverse } y \ast 1) \ast y \)
        by (simp only: group-left-inverse)
      \item also have \( \ldots = \text{inverse } y \ast y \)
        by (simp only: group-right-one)
      \item also have \( \ldots = 1 \)
        by (simp only: group-left-inverse)
      \item finally show \( \text{thesis} \).
    \end{itemize}
\end{itemize}
qed

\textbf{Theorem (in group)} \textbf{inverse-inverse}: inverse \( (\text{inverse } x) = x \)
\textbf{Proof (rule group-inverse-equality)}
\begin{itemize}
  \item show \( x \ast \text{inverse } x = 1 \)
  \item by (simp only: group-right-inverse)
\end{itemize}
qed

\textbf{Theorem (in group)} \textbf{inverse-inject}:
\begin{itemize}
  \item assumes \( \text{eq: inverse } x = \text{inverse } y \)
  \item shows \( x = y \)
\end{itemize}
\textbf{Proof –}
\begin{itemize}
  \item have \( x = x \ast 1 \)
    by (simp only: group-right-one)
  \item also have \( \ldots = x \ast (\text{inverse } y \ast y) \)
    by (simp only: group-left-inverse)
  \item also have \( \ldots = x \ast (\text{inverse } x \ast y) \)
    by (simp only: eq)
  \item also have \( \ldots = (x \ast \text{inverse } x) \ast y \)
    by (simp only: group-assoc)
\end{itemize}
also have \ldots = 1 \ast 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

hypothetical group axiomatization

context
fixes prod :: 'a \Rightarrow 'a (infixl ** 70)
  and one :: 'a
  and inverse :: 'a \Rightarrow 'a
assumes assoc: (x ** y) ** z = x ** (y ** z)
  and left-one: one ** x = x
  and left-inverse: inverse x ** 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: \( x \ast\ast one = x \)
proof
  have \( x \ast\ast one = x \ast\ast (inverse x \ast\ast x) \)
    by (simp only: left-inverse)
  also have \( \ldots = x \ast\ast inverse x \ast\ast x \)
    by (simp only: assoc)
  also have \( \ldots = one \ast\ast x \)
    by (simp only: right-inverse)
  also have \( \ldots = x \)
    by (simp only: left-one)
finally show \( x \ast\ast one = x \).
qed

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

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

26
also have \ldots = x ** inverse x ** x
  by (simp only: assoc)
also have \ldots = one ** x
  by (simp only: right-inverse)
also have \ldots = x
  by (simp only: left-one)
finally show x ** one = x .
qed

have one-equality: \(\forall x. e ** x = x \implies one = e\)
proof -
  fix e x
  assume eq: e ** x = x
  have one = x ** inverse x
    by (simp only: right-inverse)
also have \ldots = (e ** x) ** inverse x
  by (simp only: eq)
also have \ldots = e ** (x ** inverse x)
  by (simp only: assoc)
also have \ldots = e ** one
  by (simp only: right-inverse)
also have \ldots = e
  by (simp only: right-one)
finally show one = e .
qed

have inverse-equality: \(\forall x' . x' ** x = one \implies inverse x = x'\)
proof -
  fix x x'
  assume eq: x' ** x = one
  have inverse x = one ** inverse x
    by (simp only: left-one)
also have \ldots = (x' ** x) ** inverse x
  by (simp only: eq)
also have \ldots = x' ** (x ** inverse x)
  by (simp only: assoc)
also have \ldots = x'
  by (simp only: right-inverse)
also have \ldots = x'
  by (simp only: right-one)
finally show inverse x = x' .
qed

end

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 (([::-] [60, 61] 60)
  | Cond 'a bexp 'a com 'a com
  | While 'a bexp 'a assn 'a com

abbreviation Skip (SKIP)
where SKIP ≡ Basic id

type-synonym 'a sem = 'a ⇒ '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'' ∧ 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''. 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
  (([:−]/ [:−]) [100, 55, 100] 50)
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.

\[ \begin{align*}
\text{theorem} & \quad \text{basic: } \vdash \{ s, f s \in P \} (\text{Basic } f) \ P \\
\text{proof} & \quad \text{fix } s s' \\
& \quad \text{assume } s: s \in \{ s, f s \in P \} \\
& \quad \text{assume } \text{Sem } (\text{Basic } f) s s' \\
& \quad \text{then have } s' = f s \text{ by simp} \\
& \quad \text{with } s \text{ show } s' \in P \text{ by simp} \\
\text{qed}
\end{align*} \]

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

\[ \begin{align*}
\text{theorem} & \quad \text{seq: } \vdash P \ c1 \ Q = \vdash Q \ c2 \ R \implies \vdash P \ (c1; c2) \ R \\
\text{proof} & \quad \text{assume } \text{cmd1: } \vdash P \ c1 \ Q \text{ and cmd2: } \vdash Q \ c2 \ R \\
& \quad \text{fix } s s' \\
& \quad \text{assume } s: s \in P \\
& \quad \text{assume } \text{Sem } (c1; c2) s s' \\
& \quad \text{then obtain } s'' \text{ where } \text{sem1: } \text{Sem } c1 s s'' \text{ and sem2: } \text{Sem } c2 s'' s' \\
& \quad \text{by auto} \\
& \quad \text{from cmd1 sem1 } s \text{ have } s'' \in Q .. \\
& \quad \text{with cmd2 sem2 show } s' \in R .. \\
\text{qed}
\end{align*} \]

\[ \begin{align*}
\text{theorem} & \quad \text{conseq: } P' \subseteq P \implies \vdash P \ c \ Q \implies Q \subseteq Q' \implies \vdash P' \ c \ Q' \\
\text{proof} & \quad \text{assume } P'P: P' \subseteq P \text{ and } QQ': Q \subseteq Q' \\
& \quad \text{assume } \text{cmd: } \vdash P \ c \ Q \\
& \quad \text{fix } s s': 'a \\
& \quad \text{assume } \text{sem: } \text{Sem } c s s' \\
& \quad \text{assume } s : P' \text{ with } P'P \text{ have } s \in P .. \\
& \quad \text{with } \text{cmd sem have } s' \in Q .. \\
& \quad \text{with } QQ' \text{ show } s' \in Q' .. \\
\text{qed}
\end{align*} \]

The rule for conditional commands is directly reflected by the corresponding semantics; in the proof we just have to look closely which cases apply.

\[ \begin{align*}
\text{theorem} & \quad \text{cond:} \\
& \quad \text{assumes } \text{case-b: } \vdash (P \cap b) \ c1 \ Q \\
& \quad \text{and case-nb: } \vdash (P \cap \neg b) \ c2 \ Q \\
& \quad \text{shows } \vdash P \ (\text{Cond } b \ c1 \ c2) \ Q
\end{align*} \]
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
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: Γ ⊢ (P ∩ b) c P
  shows Γ ⊢ 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 \emph{quotation} is a syntactic entity delimited by an implicit abstraction, say over the state space. An \emph{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 \texttt{Syntax_Trans.quote_tr}, and \texttt{Syntax_Trans.quote_tr’}).

\begin{verbatim}
syntax
  -quote :: 'b ⇒ ('a ⇒ 'b)
  -antiquote :: ('a ⇒ 'b) ⇒ 'b (\texttt{\textasciitilde} [1000] 1000)
  -Subst :: 'a bexp ⇒ 'b ⇒ idt ⇒ 'a bexp \texttt{(\texttt{\textasciitilde} /\textasciitilde \texttt{/} \texttt{\textasciitilde}) [1000] 999}
  -Assert :: 'a ⇒ 'a set (\texttt{\{}\texttt{|} \texttt{-} \texttt{|} \texttt{\}} [0] 1000)
  -Assign :: idt ⇒ 'b ⇒ 'a com\texttt{(\texttt{\textasciitilde}/\textasciitilde \texttt{/} \texttt{\textasciitilde}) [70, 65] 61}
  -Cond :: 'a bexp ⇒ 'a com ⇒ 'a com ⇒ 'a com\texttt{(0IF \texttt{-/\texttt{-} \texttt{-} \texttt{FI}} [0, 0, 0] 61)
  -While-inv :: 'a bexp ⇒ 'a assn ⇒ 'a com ⇒ 'a com\texttt{(0WHILE \texttt{-/\texttt{-} \texttt{-} \texttt{DO -/\texttt{-} \texttt{OD}} [0, 0, 0] 61)
  -While :: 'a bexp ⇒ 'a com ⇒ 'a com\texttt{(0WHILE \texttt{-/\texttt{-} \texttt{-} \texttt{DO -/\texttt{-} \texttt{OD}} [0, 0] 61)

translations
\{b\} ⇒ CONST Collect (-quote b)
B\texttt{[\texttt{/a/\texttt{/} x} \texttt{\rightarrow \{}\texttt{(\texttt{-update-name x} (\texttt{\textasciitilde} /\texttt{\textasciitilde} \texttt{\textasciitilde}) \texttt{\textasciitilde} \texttt{\textasciitilde} (\texttt{\textasciitilde} a))})\texttt{]} \texttt{\textasciitilde} x :\texttt{=} a ⇒ CONST Basic (-quote (\texttt{(\texttt{-update-name x} (\texttt{\textasciitilde} /\texttt{\textasciitilde} \texttt{\textasciitilde}) \texttt{\textasciitilde} \texttt{\textasciitilde} (\texttt{\textasciitilde} a))}))\texttt{]} \texttt{\textasciitilde} b \texttt{THEN} c1\texttt{ELSE} c2 \texttt{FI} ⇒ CONST Cond \texttt{\{b\} c1 c2}
WHILE b INV i DO c OD ⇒ CONST While \texttt{\{b\} i c}
WHILE b DO c OD ⇒ WHILE b INV CONST undefined DO c OD

parse-translation
let
  \texttt{fan quote-tr \{t = Syntax-Trans.quote-tr \@\{syntax-const -antiquote\} t}
  | quote-tr ts = raise TERM (quote-tr, ts);
\texttt{in \{[@\{syntax-const -quote\}, K quote-tr\}] end}
\end{verbatim}

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.
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 introduced above.

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

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

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

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

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

lemma [trans]:
\(\vdash \{P\} \land Q \implies \{P\} \land Q \implies \vdash \{P\} \land Q\)
by (simp add: Valid-def)

lemma [trans]:

\[(\forall s. P' \rightarrow P s) \rightarrow \vdash \{\bar{\{ P \}}\} c Q \rightarrow \vdash \{\bar{\{ P' \}}\} c Q\]

by \((simp \ add: \ Valid-def)\)

**Lemma** [trans]:
\[\vdash P c \{\bar{\{ Q \}}\} \rightarrow (\forall s. Q s \rightarrow Q' s) \rightarrow \vdash P c \{\bar{\{ Q' \}}\}\]

by \((simp \ add: \ Valid-def)\)

**Lemma** [trans]:
\[\vdash (\forall s. Q s \rightarrow Q' s) \rightarrow \vdash P c \{\bar{\{ Q \}}\} \rightarrow \vdash P c \{\bar{\{ Q' \}}\}\]

by \((simp \ add: \ Valid-def)\)

Identity and basic assignments.\(^7\)

**Lemma** skip [intro?): \[\vdash P SKIP P\]

**Proof** –

have \[\vdash \{ s. \text{id} s \in P \} \text{SKIP} P\] by \((rule \ basic)\)

then show \(?\text{thesis}\) by \((simp)\)

**Qed**

**Lemma** assign: \[\vdash P [a/\bar{x}::\bar{a}] \bar{x} := \bar{a} P\]

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 function \(x\) selector and \(x\)-update (update). Above, there is only a place-holder appearing for the latter kind of function: due to concrete syntax \(\bar{x} := \bar{a}\) also contains \(x\)-update.\(^8\)

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

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

**Lemma** seq-assoc [simp]: \[\vdash P c1;(c2;c3) Q \leftrightarrow \vdash P (c1;c2);c3 Q\]

by \((auto \ simp \ add: \ Valid-def)\)

Conditional statements.

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

**Lemma** [trans, intro?]:
\[\vdash \{\bar{\{ P \}} \land \bar{\{ b \}}\} c1 Q \rightarrow \vdash \{\bar{\{ P \}} \land \bar{\{ b \}}\} c2 Q \rightarrow \vdash \{\bar{\{ P \}}\} \text{IF} \bar{\{ b \}} \text{THEN} c1 \text{ELSE} c2 \text{FI} Q\]

by \((rule \ cond) \ (simp-all \ add: \ Valid-def)\)

While statements — with optional invariant.

---

\(^7\)The hoare method introduced in \(\S9.5\) 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 [intro?]: \( \vdash (P \cap b) \ c \ P \implies \vdash P \ (\text{While} \ b \ P \ c) \ (P \cap \neg b) \)
by (rule while)

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

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

lemma [intro?]:
\[ \vdash \{ P \land \neg b \} \ c \ { P \lor \neg b } \]
\[ \implies \vdash \{ P \lor \neg b \} \ \text{WHILE} \ b \ DO \ c \ OD \ \{ P \land \neg \ b \} \]
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 `hoare`, which may be applied to a Hoare Logic assertion to extract purely logical verification conditions. It is important to note that the method requires WHILE loops to be fully annotated with invariants beforehand. Furthermore, only concrete pieces of code are handled — the underlying tactic fails ungracefully if supplied with meta-variables or parameters, for example.

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

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

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

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

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

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

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

lemmas AbortRule = SkipRule — dummy version

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

method-setup hoare = ⟨⟨
  Scan.succeed (fn ctxt =>
    (SIMPLE-METHOD'
      (Hoare.hoare-tac ctxt
        (simp-tac (put-simpset HOL-basic-sset ctxt addsimps [@{thm Record.K-record-comp}])))
    ))))⟩

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.

\[
\begin{align*}
\text{lemma} \vdash \{ (N\text{-update } (\lambda\cdot \mathbf{2} \ast N)) \in \{ N = 10 \} \} \quad \{ N := 2 \ast N \ \{ N = 10 \} \} \\
\text{by (rule assign)}
\end{align*}
\]

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

\[
\begin{align*}
\text{lemma} \vdash \{ \text{True}\} \quad N := 10 \ \{ N = 10 \} \\
\text{by hoare}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \vdash \{ 2 \ast N = 10 \} \quad N := 2 \ast N \ \{ N = 10 \} \\
\text{by hoare simp}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \vdash \{ \mathbf{N} = 5\} \quad N := 2 \ast N \ \{ N = 10 \} \\
\text{by hoare simp}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \vdash \{ \mathbf{N} + 1 = a + 1\} \quad N := \mathbf{N} + 1 \ \{ N = a + 1 \} \\
\text{by hoare simp}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \vdash \{ a = a \land b = b\} \quad \mathbf{M} := a; \quad \mathbf{N} := b \ \{ \mathbf{M} = a \land \mathbf{N} = b \} \\
\text{by hoare}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \vdash \{ \text{True}\} \quad \mathbf{M} := a; \quad \mathbf{N} := b \ \{ \mathbf{M} = a \land \mathbf{N} = b \} \\
\text{by hoare}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \quad \{ \mathbf{M} = a \land \mathbf{N} = b\}
\quad I := \mathbf{M}; \quad \mathbf{M} := \mathbf{N}; \quad \mathbf{N} := \mathbf{I} \\
\quad \{ \mathbf{M} = b \land \mathbf{N} = a\} \\
\text{by hoare simp}
\end{align*}
\]

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.

\[
\begin{align*}
\text{lemma} \vdash \{ \mathbf{N} = a\} \quad \mathbf{N} := \mathbf{N} \ \{ \mathbf{N} = a\} \\
\text{by hoare}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} \vdash \{ \mathbf{x} = a\} \quad \mathbf{x} := \mathbf{x} \ \{ \mathbf{x} = a\}
\end{align*}
\]
lemma

Valid \{ s. x s = a \} (Basic (\lambda s. x-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 \{ \delta \}
\{ \delta \} := \{ \delta \} + 1 \{ \delta \} 

proof

have \{ \delta \} \subseteq \{ \delta \} + 1 \neq \{ \delta \} 
by auto
also have \vdash ... \delta := \delta + 1 \{ \delta \} 
by hoare
finally show ?thesis .

qed

10.3 Multiplication by addition

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

lemma

\vdash \{ \delta \} \{ \delta \} := \delta + 1 \{ \delta \} 

proof

let \vdash - ?while - = ?thesis 
let \{ ?inv \} = \{ \delta \} = \{ \delta \} + b \}

have \{ \delta \} \subseteq \{ \delta \} \subseteq \{ \delta \} \subseteq \{ \delta \} by auto
also have \vdash ... ?while \{ ?inv \wedge \neg (\delta \neq a) \}

qed
proof

let \( ?c = 'S := 'S + b; 'M := 'M + 1 \)

have \( \{ '?inv \land 'M \neq a \} \subseteq \{ 'S + b = ( 'M + 1) \ast b \} \)

by auto

also have \( \vdash \ldots '?c \{ '?inv \} \) by hoare

finally show \( \vdash \{ '?inv \land 'M \neq a \} '?c \{ '?inv \} . \)

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

\( \vdash \{ '?'M = 0 \land 'S = 0 \} \)

\( \text{WHILE} \ 'M \neq a \)

\( \text{INV} \ 'S = 'M \ast b \)

\( \text{DO} \ 'S := 'S + 'I; 'M := 'M + 1 \text{ OD} \)

\( \{ 'S = a \ast b \} \)

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

\( \vdash \{ 'True \} \)

\( 'S := 0; 'I := 1; \text{WHILE} 'I \neq n \)

\( \text{DO} \ 'S := 'S + 'I; 'I := 'I + 1 \text{ OD} \)

\( \{ 'S = (\sum_{j<n} j) \} \)

(is \( \vdash - (\text{while}) -) \)

proof –

let \( ?sum = \lambda k :: \text{nat}. \sum_{j<k} j \)

let \( ?inv = \lambda s i :: \text{nat}. s = ?sum i \)

have \( \vdash \{ 'True \} \ 'S := 0; 'I := 1 \ 'I \in \{ '?inv \ 'S \} \)

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

**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 hoare
show $?\text{inv} \ 0 \ I$ by simp
next
fix $s \ i$
assume $?\text{inv} \ s \ i \land i \neq n$
then show $?\text{inv} \ (s + i) \ (i + 1)$ by simp
next
fix $s \ i$
assume $?\text{inv} \ s \ i \land \neg i \neq n$
then show $s = ?\text{sum} \ n$ by simp
qed
qed
Certainly, this proof may be done fully automatic as well, provided that the invariant is given beforehand.

**Theorem**

\[ \{ \text{True} \} \]

`S := 0; `I := 1;

WHILE `I ≠ n

INV \{ `S = (\sum j<`I. j) \}

DO

`S := `S + `I;
`I := `I + 1

OD

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

by hoare auto

### 10.5 Time

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

**Record**

`tstate = time :: nat`

**Type-synonym**

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

**Primrec**

`timeit :: 'a time com ⇒ 'a time com`

**Where**

\[
\begin{align*}
\text{timeit (Basic f)} &= (\text{Basic f}; \text{Basic}(\lambda s. s(time := \text{Suc}(\text{time s})))) \\
\text{timeit (c1; c2)} &= (\text{timeit c1}; \text{timeit c2}) \\
\text{timeit (Cond b c1 c2)} &= \text{Cond b (timeit c1) (timeit c2)} \\
\text{timeit (While b iv c)} &= \text{While b iv (timeit c)}
\end{align*}
\]

**Record**

`tvars = tstate +

I :: nat

J :: nat`

**Lemma**

`lem: (0::nat) < n ⇒ n + n ≤ Suc (n * n)`

by (induct n) simp-all

**Lemma**

\[ \{ i = 'I ∧ 'time = 0 \} \]

(timeit

WHILE `I ≠ 0

INV \{ 2 * 'time + 'I * 'I + 5 * 'I = i * i + 5 * i \}

DO

'J := 'I;

WHILE 'J ≠ 0

INV \{ 0 < 'I ∧ 2 * 'time + 'I * 'I + 3 * 'I + 2 * 'J + 2 = i * i + 5 * i \}

DO 'J := 'J - 1 OD;

'J := 'I - 1
Theorem. Let $L$ be a complete lattice and $f: L \to L$ an order-preserving map. Then $\bigcap \{x \in L \mid f(x) \leq x\}$ is a fixpoint of $f$.

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

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.

\textsuperscript{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 \( x \in \ ?H \)
then have \( ?a \leq x \) by (rule Inf-lower)
with (mono \( f \)) have \( f ?a \leq f x \).
also from \( \{ x \in \ ?H \} \) have \( \ldots \leq x \).
finally show \( f ?a \leq x \).
qed
show \( ?a \leq f ?a \)
proof (rule Inf-lower)
from (mono \( f \)) and \( f ?a \leq ?a \) have \( f (f ?a) \leq f ?a \).
then show \( f ?a \in \ ?H \).
qed
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-ldt tiling :: 'a set set \( \Rightarrow \) 'a set set
for A :: 'a set set
where
empty: \{ \} \in tiling A
| \( \text{Un} \): \( a \in A \Rightarrow t \in tiling A \Rightarrow a \subseteq - t \Rightarrow a \cup t \in tiling A \)

The union of two disjoint tilings is a tiling.

lemma tiling-Un:
assumes t \( \in \) tiling A
and u \( \in \) tiling A
and t \( \cap \) u = \{ \}
shows t \( \cup \) u \( \in \) tiling A
proof –
let \( ?T = \text{tiling} A \)
from t \( \in ?T \) and \( t \cap u = \{ \} \)
show t \( \cup \) u \( \in ?T \)
proof (induct t)
case empty
with \( u \in ?T \) show \{ \} \( \cup \) u \( \in ?T \) by simp
next
case \( (\text{Un} \ a \ t) \)
show \((a \cup t) \cup u \in ?T\)

proof -
have \(a \cup (t \cup u) \in ?T\)
    using \((a \in A)\)
proof (rule tiling.Un)
  from \((a \cup t) \cap u = \{\}\) have \(t \cap u = \{\}\) by blast
then show \(t \cup u \in ?T\) by (rule Un)
from \((a \subseteq t)\) and \((a \cup t) \cap u = \{\}\)
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 :: nat ⇒ 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: \(\text{below} 0 = \{\}\)
by (simp add: below-def)

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

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

lemmas Sigma-Suc = Sigma-Suc1 Sigma-Suc2

12.3 Basic properties of “evnodd”

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

lemma evnodd-iff: \((i, j) \in \text{evnodd} A b \iff (i, j) \in A \land (i + j) \mod 2 = b\)
by (simp add: evnodd-def)

lemma evnodd-subset: \(\text{evnodd} A b \subseteq A\)
unfolding evnodd-def by (rule Int-lower1)
lemma evnoddD: \( x \in \text{evnodd} A b \implies x \in A \)
by (rule subsetD) (rule evnodd-subset)

lemma evnodd-finite: finite \( A \implies \) finite (evnodd \( A \) \( b \))
by (rule finite-subset) (rule evnodd-subset)

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

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

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

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

12.4 Dominoes

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

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

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

lemma domino-finite:  
assumes \(d \in \) 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 \(t : t \in \) tiling domino  
  (is \(?t \in ?T\))  
  shows finite \(t\) (is \(?F t\))  
  using \(t\)
proof induct
  show ?F {} by (rule finite.emptyI)
fix a t assume ?F t
  assume a ∈ domino
then have ?F a by (rule domino-finite)
from this and a ∈ domino have ?F a by (rule domino-finite)
proof induct
next
  case empty
  show ?case by (simp add: domino-finite)
next
  case (Un a t)
  let ?e = evnodd
  note hyp = ⟨card (?e t 0) = card (?e t 1)⟩
  and at = (a ⊆ − t)
  have card-suc:
    (∀ b. b < 2 ⇒ card (?e a b) = Suc (card (?e b)))
    proof -
      fix b :: nat
      assume b < 2
      have ?e a b = ?e a b ∪ ?e b by (rule evnodd-Un)
      also obtain i j where e: ?e a b = {(i, j)}
      proof -
        from ⟨a ∈ domino⟩ and (b < 2)
        have ∃ i j. ?e a b = {(i, j)} by (rule domino-singleton)
        then show ?thesis by (blast intro: that)
      qed
      also have ... ∪ ?e b = insert (i, j) (?e t b) by simp
      also have card ... = Suc (card (?e t b))
      proof (rule card-insert-disjoint)
        from ⟨t ∈ domino⟩ have finite t
        by (rule tiling-domino-finite)
        then show finite (?e t b)
        by (rule evnodd-finite)
        from e have (i, j) ∈ ?e a b by simp
        with at show (i, j) /∈ ?e b by (blast dest: evnoddD)
      qed
      finally show ?thesis a .
  qed
  then have card (?e a t 0) = Suc (card (?e t 0)) by simp
  also from hyp have card (?e t 0) = card (?e t 1) .
  also from card-suc have Suc ... = card (?e a t 1) by simp
  finally show ?case .
qed
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 card (?e ?t' 0 − {(2 * m + 1, 2 * n + 1)})

< card (?e ?t' 0)

proof (rule card-Diff1-less)

from - fin show finite (?e ?t' 0)

by (rule finite-subset) auto

show (2 * m + 1, 2 * n + 1) ∈ ?e ?t' 0 by simp

qed

then show ?thesis by simp

qed

also have ... < card (?e ?t 0)

proof –

have (0, 0) ∈ ?e ?t 0 by simp

with fin have card (?e ?t 0 − {(0, 0)}) < card (?e ?t 0)

by (rule card-Diff1-less)

then show ?thesis by simp

qed

also from t'' have ... = card (?e ?t'')

by (rule tiling-domino-01)

also have ?e ?t 0 = ?e ?t'' 0 by simp

also from t'' have card ... = card (?e ?t'' 0)
by (rule tiling-domino-01 [symmetric])
finally have ... < ... then show False ..
qed
qed
end

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
primrec subst-term :: ('a ⇒ ('a, 'b) term) ⇒ ('a, 'b) term ⇒ ('a, 'b) term
and subst-term-list :: ('a ⇒ ('a, 'b) term) ⇒ ('a, 'b) term list ⇒ ('a, 'b) term 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-subst-term-list.simps

A simple lemma about composition of substitutions.

lemma
  subst-term (subst-term f1 o f2) t =
  subst-term f1 (subst-term f2 t)
and
  subst-term-list (subst-term f1 o f2) ts =
  subst-term-list f1 (subst-term-list f2 ts)
by (induct t and ts) simp-all

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

49
next
  fix b ts assume ?Q ts
  then show ?P (App b ts)
    by (simp only: subst-simps)
next
  show ?Q [] by simp
next
  fix t ts
  assume ?P t ?Q ts then show ?Q (t ≠ ts)
    by (simp only: subst-simps)
qed
qed

13.2 Alternative induction

theorem term-induct' [case-names Var App]:
  assumes var: ∀a. P (Var a)
  and app: ∀b ts. (∀t ∈ set ts. P t) ⟹ P (App b ts)
  shows P t
proof (induct t)
  fix a show P (Var a) by (rule var)
next
  fix b t ts assume ∀t ∈ set ts. P t
  then show P (App b ts) by (rule app)
next
  show ∀t ∈ set[]. P t by simp
next
  fix t ts assume P t ∀t' ∈ set ts. P t'
  then show ∀t' ∈ set (t ≠ ts). P t' by simp
qed

lemma subst-term (subst-term f1 o 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 → B) → A) → 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.}

\textbf{theorem} \( ((A \rightarrow B) \rightarrow A) \rightarrow A \)

\textbf{proof}
\begin{itemize}
  \item \textbf{assume} \( (A \rightarrow B) \rightarrow A \)
  \item \textbf{show} \( A \)
\end{itemize}
\textbf{proof} (\textit{rule classical})
\begin{itemize}
  \item \textbf{assume} \( \neg A \)
  \item \textbf{have} \( A \rightarrow B \)
  \item \textbf{proof}
  \begin{itemize}
    \item \textbf{assume} \( A \)
    \item \textbf{with} \( (\neg A) \) \textbf{show} \( B \) \textbf{by contradiction}
    \item \textbf{qed}
  \end{itemize}
  \item \textbf{with} \( (A \rightarrow B) \rightarrow A; \) \textbf{show} \( A \).
\item \textbf{qed}
\item \textbf{qed}
\end{itemize}

In the subsequent version the reasoning is rearranged by means of “weak assumptions” (as introduced by \textbf{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 \textbf{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 \textbf{assume}) are solved immediately.

\textbf{theorem} \( ((A \rightarrow B) \rightarrow A) \rightarrow A \)

\textbf{proof}
\begin{itemize}
  \item \textbf{assume} \( (A \rightarrow B) \rightarrow A \)
  \item \textbf{show} \( A \)
\end{itemize}
\textbf{proof} (\textit{rule classical})
\begin{itemize}
  \item \textbf{presume} \( A \rightarrow B \)
  \item \textbf{with} \( (A \rightarrow B) \rightarrow A; \) \textbf{show} \( A .. \)
\end{itemize}
\textbf{next}
\begin{itemize}
  \item \textbf{assume} \( \neg A \)
  \item \textbf{show} \( A \rightarrow B \)
\end{itemize}
\textbf{proof}
\begin{itemize}
  \item \textbf{assume} \( A \)
  \item \textbf{with} \( (\neg A) \) \textbf{show} \( B \) \textbf{by contradiction}
  \item \textbf{qed}
\item \textbf{qed}
\item \textbf{qed}
Note that the goals stemming from weak assumptions may be even left until qed time, where they get eventually solved “by assumption” as well. In that case there is really no fundamental difference between the two kinds of assumptions, apart from the order of reducing the individual parts of the proof configuration.

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

15 An old chestnut

theory Puzzle
imports Main
begin

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

theorem
assumes f-ax: $\forall n. \ f \ (f \ n) < f \ (Suc \ n)$
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 $m < f \ n$.
\}

end

\footnote{A question from “Bundeswettbewerb Mathematik”. Original pen-and-paper proof due to Herbert Ehler; Isabelle tactic script by Tobias Nipkow.}
then have \( n \leq f \cdot n \) by \( \text{simp only: Suc} \)
then show \(?\text{thesis} \).

next

  case 0
  then show \(?\text{thesis by simp} \)
  qed

  qed

} note \( ge = \text{this} \)

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

  qed

} note \( \text{mono = this} \)

show \( f \cdot n \leq n \)
proof -

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

  then show \(?\text{thesis by simp} \)
  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.

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

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

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

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

1. Facts involved in also / finally calculational chains may be just anything. There is nothing special about have, so the natural deduction element assume works just as well.
2. There are two separate primitives for building natural deduction contexts: \textbf{fix} \(x\) and \textbf{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.

\textbf{theorem sum-of-odds:}
\[
(\sum i :: \text{nat} \leq n. \ 2 \times i + 1) = n \cdot \text{Suc} (\text{Suc} 0)
\]
(is \(\cong P n\) is \(\cong S n = -\))

\textbf{proof (induct n)}
- \textbf{show} \(\cong P 0\) by simp
- \textbf{next}
  - \textbf{fix} \(n\)
  - \textbf{have} \(\cong S (n + 1) = \cong S n + 2 \times n + 1\)
    by simp
  - also assume \(\cong S n = n \cdot \text{Suc} (\text{Suc} 0)\)
  - also have \(\ldots + 2 \times n + 1 = (n + 1) \cdot \text{Suc} (\text{Suc} 0)\)
    by simp
  - \textbf{finally show} \(\cong P (\text{Suc} n)\)
    by simp

\textbf{qed}

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

\textbf{lemmas distrib = add-mult-distrib add-mult-distrib2}

\textbf{theorem sum-of-squares:}
\[
6 \cdot (\sum i :: \text{nat} \leq n. \ i \cdot \text{Suc} (\text{Suc} 0)) = n \times (n + 1) \times (2 \times n + 1)
\]
(is \(\cong P n\) is \(\cong S n = -\))

\textbf{proof (induct n)}
- \textbf{show} \(\cong P 0\) by simp
- \textbf{next}
  - \textbf{fix} \(n\)
  - \textbf{have} \(\cong S (n + 1) = \cong S n + 6 \times (n + 1) \cdot \text{Suc} (\text{Suc} 0)\)
    by (simp add: distrib)
  - also assume \(\cong S n = n \times (n + 1) \times (2 \times n + 1)\)
  - also have \(\ldots + 6 \times (n + 1) \cdot \text{Suc} (\text{Suc} 0) = (n + 1) \times (n + 2) \times (2 \times (n + 1) + 1)\)
    by (simp add: distrib)
  - \textbf{finally show} \(\cong P (\text{Suc} n)\)
    by simp

\textbf{qed}

\textbf{theorem sum-of-cubes:}
\[
4 \cdot (\sum i :: \text{nat} \leq n. \ i^3) = (n \times (n + 1))^3 \cdot \text{Suc} (\text{Suc} 0)
\]
(is ?P n is ?S n = -)
proof (induct n)
  show ?P 0 by (simp add: power-eq-if)
next
  fix n
  have ?S (n + 1) = ?S n + 4 * (n + 1)^3
    by (simp add: power-eq-if distrib)
  also assume ?S n = (n * (n + 1))^Suc (Suc 0)
  also have ... + 4 * (n + 1)^3 = ((n + 1) * ((n + 1) + 1))^Suc (Suc 0)
    by (simp add: power-eq-if distrib)
  finally show ?P (Suc n)
    by simp
qed

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

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

end

References


