Miscellaneous Isabelle/Isar examples for Higher-Order Logic

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

With contributions by Gertrud Bauer and Tobias Nipkow

June 8, 2008

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

theory *BasicLogic* **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 --> A
proof
  assume A
  show A by fact
qed

lemma K: A --> B --> A
proof
  assume A
  show B --> A
  proof
    show A by fact
  qed
qed

lemma S: (A --> B --> C) --> (A --> B) --> A --> C
proof
  assume A --> B --> C
  show (A --> B) --> A --> C
  proof
    assume A --> B
    show A --> C
    proof
      assume A
      show C
      proof (rule mp)
        show B --> C by (rule mp) fact+
        show B by (rule mp) fact+
      qed
    qed
  qed
qed
```

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

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

```
lemma A --> A
proof
  assume A
  show A by fact+
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.

lemma $A \longrightarrow A$
proof
qed

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

lemma $A \longrightarrow A$
  by *rule*

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

lemma $A \longrightarrow A$ **..**

Thus we have arrived at an adequate representation of the proof of a tautology that holds by a single standard rule.[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 *intro* proof method repeatedly decomposes a goal's conclusion.[3]

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

Again, the body may be collapsed.

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

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

Such well-tuned iterated decomposition of certain structures is the prime application of *intro* and *elim*. In contrast, terminal steps that solve a goal

[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 *elim*, acting on a goal's premises.

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

The first version is purely backward.

```
lemma A & B --> B & A
proof
  assume A & B
  show B & 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 $A \wedge B$ assumption as the current facts, enabling the use of double-dot proofs. Note that **from** already does forward-chaining, involving the *conjE* rule here.

```
lemma A & B --> B & A
proof
  assume A & B
  show B & A
  proof
    from ‹A & B› show B ..
    from ‹A & 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 **then** command. Thus the proof of $B \wedge A$ from $A \wedge B$ actually becomes an elimination, rather than an introduction. The resulting proof structure directly corresponds to that of the *conjE* rule, including the repeated goal proposition that is abbreviated as *?thesis* below.

```
lemma A & B --> B & A
proof
  assume A & B
  then show B & A
```

```
  proof                    — rule conjE of A ∧ B
    assume B A
    then show ?thesis ..   — rule conjI of B ∧ 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 & B --> B & A
proof
  assume A & B
  from ‹A & B› have A ..
  from ‹A & B› have B ..
  from ‹B› ‹A› show B & A ..
qed
```

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

```
lemma A & B --> B & A
proof -
  {
    assume A & B
    from ‹A & B› have A ..
    from ‹A & B› have B ..
    from ‹B› ‹A› have B & A ..
  }
  then show ?thesis ..          — rule impI
qed
```

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

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

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

```
lemma A & B --> B & A
```

```
proof
  assume A & B
  then show B & A
  proof
    assume B A
    then show ?thesis ..
  qed
qed
```

1.3 A few examples from "Introduction to Isabelle"

We rephrase some of the basic reasoning examples of [5], using HOL rather than FOL.

1.3.1 A propositional proof

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

```
lemma P | P --> P
proof
  assume P | P
  then show P
  proof                    — rule disjE:
    assume P show P by fact
  next
    assume P show P by fact
  qed
qed
```

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

Case splits are *not* hardwired into the Isar language as a special feature. The **next** command used to separate the cases above is just a short form of managing block structure.

In general, applying proof methods may split up a goal into separate "cases", i.e. new subgoals with individual local assumptions. The corresponding proof text typically mimics this by establishing results in appropriate contexts, separated by blocks.

In order to avoid too much explicit parentheses, the Isar system implicitly opens an additional block for any new goal, the **next** statement then closes one block level, opening a new one. The resulting 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.

```
lemma P | P --> P
proof
  assume P | P
  then show P
  proof
    assume P
    show P by fact
    show P by fact
  qed
qed
```

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

```
lemma P | P --> P
proof
  assume P | P
  then show P ..
qed
```

1.3.2 A quantifier proof

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

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

```
lemma (EX x. P (f x)) --> (EX y. P y)
proof
  assume EX x. P (f x)
  then show EX y. P y
  proof (rule exE)
    fix a
    assume P (f a) (is P ?witness)
    then show ?thesis by (rule exI [of P ?witness])
  qed
qed
```

— rule exE:

$$\frac{\exists x.\ A(x) \qquad \begin{array}{c}[A(x)]_x \\ \vdots \\ B\end{array}}{B}$$

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

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

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

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

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 *conjE*: $A\ \&\ B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$
proof −
  assume $A\ \&\ B$
  assume r: $A \Longrightarrow B \Longrightarrow C$
  show C
  proof (*rule* r)
    show A **by** (*rule conjunct1*) *fact*
    show B **by** (*rule conjunct2*) *fact*
  qed
qed

end

2 Cantor's Theorem

theory *Cantor* **imports** *Main* **begin**[4]

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:

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

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

```
theorem EX S. S ~: range (f :: 'a => 'a set)
proof
  let ?S = {x. x ~: f x}
  show ?S ~: range f
  proof
    assume ?S : range f
    then obtain y where ?S = f y ..
    then show False
    proof (rule equalityCE)
      assume y : f y
      assume y : ?S then have y ~: f y ..
      with ‹y : f y› show ?thesis by contradiction
    next
      assume y ~: ?S
      assume y ~: f y then have y : ?S ..
      with ‹y ~: ?S› show ?thesis by contradiction
    qed
  qed
qed
```

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

```
theorem EX S. S ~: range (f :: 'a => 'a 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.

[4] This is an Isar version of the final example of the Isabelle/HOL manual [4].

end

3 Peirce's Law

theory *Peirce* **imports** *Main* **begin**

We consider Peirce's Law: $((A \to B) \to A) \to A$. 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.[5]

```
theorem ((A --> B) --> A) --> A
proof
  assume (A --> B) --> A
  show A
  proof (rule classical)
    assume ~ A
    have A --> B
    proof
      assume A
      with ‹~ A› show B by contradiction
    qed
    with ‹(A --> B) --> A› show A ..
  qed
qed
```

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

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

```
theorem ((A --> B) --> A) --> A
proof
  assume (A --> B) --> A
  show A
  proof (rule classical)
    presume A --> B
```

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

```
    with ‹(A --> B) --> A› show A ..
  next
    assume ~ A
    show A --> B
    proof
      assume A
      with ‹~ A› show B by contradiction
    qed
  qed
qed
```

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

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

end

4 The Drinker's Principle

theory *Drinker* **imports** *Main* **begin**

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

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

```
lemma deMorgan:
  assumes ¬ (∀ x. P x)
  shows ∃ x. ¬ P x
  using prems
proof (rule contrapos-np)
  assume a: ¬ (∃ x. ¬ P x)
  show ∀ x. P x
  proof
    fix x
    show P x
    proof (rule classical)
```

```
    assume ¬ P x
    then have ∃ x. ¬ P x ..
    with a show ?thesis by contradiction
   qed
  qed
qed

theorem Drinker's-Principle: ∃ x. drunk x ⟶ (∀ x. drunk x)
proof cases
  fix a assume ∀ x. drunk x
  then have drunk a ⟶ (∀ x. drunk x) ..
  then show ?thesis ..
next
  assume ¬ (∀ x. drunk x)
  then have ∃ x. ¬ drunk x by (rule deMorgan)
  then obtain a where a: ¬ drunk a ..
  have drunk a ⟶ (∀ x. drunk x)
  proof
    assume drunk a
    with a show ∀ x. drunk x by (contradiction)
  qed
  then show ?thesis ..
qed

end
```

5 Correctness of a simple expression compiler

theory *ExprCompiler* **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.

5.1 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.

```
types
  'val binop = 'val => 'val => 'val
```

5.2 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.

```
consts
  eval :: ('adr, 'val) expr => ('adr => 'val) => 'val

primrec
  eval (Variable x) env = env x
  eval (Constant c) env = c
  eval (Binop f e1 e2) env = f (eval e1 env) (eval e2 env)
```

5.3 Machine

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

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

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

```
consts
  exec :: (('adr, 'val) instr) list
    => 'val list => ('adr => 'val) => 'val list

primrec
  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)

constdefs
  execute :: (('adr, 'val) instr) list => ('adr => 'val) => 'val
  execute instrs env == hd (exec instrs [] env)
```

5.4 Compiler

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

```
consts
  compile :: ('adr, 'val) expr => (('adr, 'val) instr) list
```

```
primrec
  compile (Variable x) = [Load x]
  compile (Constant c) = [Const c]
  compile (Binop f e1 e2) = compile e2 @ compile e1 @ [Apply f]
```

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

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

theorem correctness: execute (compile e) env = eval e env
proof −
  have ⋀stack. exec (compile e) stack env = eval e env # stack
  proof (induct e)
    case Variable show ?case by simp
  next
    case Constant show ?case by simp
  next
    case Binop then show ?case by (simp add: exec-append)
  qed
  then show ?thesis by (simp add: execute-def)
qed
```

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

```
lemma exec-append':
  exec (xs @ ys) stack env = exec ys (exec xs stack env) env
proof (induct xs arbitrary: stack)
```

```
  case (Nil s)
  have exec ([] @ ys) s env = exec ys s env by simp
  also have ... = exec ys (exec [] s env) env by simp
  finally show ?case .
next
  case (Cons x xs s)
  show ?case
  proof (induct x)
    case (Const val)
    have exec ((Const val # xs) @ ys) s env = exec (Const val # xs @ ys) s env
      by simp
    also have ... = exec (xs @ ys) (val # s) env by simp
    also from Cons have ... = exec ys (exec xs (val # s) env) env .
    also have ... = exec ys (exec (Const val # xs) s env) env by simp
    finally show ?case .
  next
    case (Load adr)
    from Cons show ?case by simp — same as above
  next
    case (Apply fn)
    have exec ((Apply fn # xs) @ ys) s env =
        exec (Apply fn # xs @ ys) s env by simp
    also have ... =
        exec (xs @ ys) (fn (hd s) (hd (tl s)) # (tl (tl s))) env by simp
    also from Cons have ... =
        exec ys (exec xs (fn (hd s) (hd (tl s)) # tl (tl s)) env) env .
    also have ... = exec ys (exec (Apply fn # xs) s env) env by simp
    finally show ?case .
  qed
qed

theorem correctness': execute (compile e) env = eval e env
proof −
  have exec-compile: ⋀stack. exec (compile e) stack env = eval e env # stack
  proof (induct e)
    case (Variable adr s)
    have exec (compile (Variable adr)) s env = exec [Load adr] s env
      by simp
    also have ... = env adr # s by simp
    also have env adr = eval (Variable adr) env by simp
    finally show ?case .
  next
    case (Constant val s)
    show ?case by simp — same as above
  next
    case (Binop fn e1 e2 s)
    have exec (compile (Binop fn e1 e2)) s env =
        exec (compile e2 @ compile e1 @ [Apply fn]) s env by simp
    also have ... = exec [Apply fn]
```

```
      (exec (compile e1) (exec (compile e2) s env) env) env
    by (simp only: exec-append)
  also have exec (compile e2) s env = eval e2 env # s by fact
  also have exec (compile e1) ... env = eval e1 env # ... by fact
  also have exec [Apply fn] ... env =
      fn (hd ...) (hd (tl ...)) # (tl (tl ...)) by simp
  also have ... = fn (eval e1 env) (eval e2 env) # s by simp
  also have fn (eval e1 env) (eval e2 env) =
      eval (Binop fn e1 e2) env
    by simp
  finally show ?case .
qed

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

end
```

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 *times* is provided by the basic HOL theory.

```
consts
  one :: 'a
  inverse :: 'a => 'a

axclass
  group < times
  group-assoc:         (x * y) * z = x * (y * z)
  group-left-one:      one * x = x
  group-left-inverse:  inverse x * x = one
```

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

```
theorem group-right-inverse: x * inverse x = (one::'a::group)
proof -
  have x * inverse x = one * (x * inverse x)
```

```
  by (simp only: group-left-one)
 also have ... = one * x * inverse x
  by (simp only: group-assoc)
 also have ... = inverse (inverse x) * inverse x * x * inverse x
  by (simp only: group-left-inverse)
 also have ... = inverse (inverse x) * (inverse x * x) * inverse x
  by (simp only: group-assoc)
 also have ... = inverse (inverse x) * one * inverse x
  by (simp only: group-left-inverse)
 also have ... = inverse (inverse x) * (one * inverse x)
  by (simp only: group-assoc)
 also have ... = inverse (inverse x) * inverse x
  by (simp only: group-left-one)
 also have ... = one
  by (simp only: group-left-inverse)
 finally show ?thesis .
qed
```

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

```
theorem group-right-one: x * one = (x::'a::group)
proof -
 have x * one = x * (inverse x * x)
  by (simp only: group-left-inverse)
 also have ... = x * inverse x * x
  by (simp only: group-assoc)
 also have ... = one * x
  by (simp only: group-right-inverse)
 also have ... = 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.

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

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

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 $x * one = (x::'a::group)$
proof −
  have $x * one = x * (inverse\ x * x)$
    by (*simp only*: *group-left-inverse*)

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

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

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

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

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

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

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

  show *?thesis* **.**
qed

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

6.2 Groups as monoids

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

axclass *monoid* < *times*
  monoid-assoc: $(x * y) * z = x * (y * z)$
  monoid-left-one: $one * x = x$
  monoid-right-one: $x * one = x$

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 18), 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 *group-one-equality*: $e * x = x \Longrightarrow one = (e::'a::group)$
proof −
  assume *eq*: $e * x = x$
  have $one = x * inverse\ x$
    by (*simp only*: *group-right-inverse*)
  also have ... = $(e * x) * inverse\ x$
    by (*simp only*: *eq*)
  also have ... = $e * (x * inverse\ x)$
    by (*simp only*: *group-assoc*)
  also have ... = $e * one$
    by (*simp only*: *group-right-inverse*)
  also have ... = e
    by (*simp only*: *group-right-one*)
  finally show *?thesis* **.**
qed

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

theorem *group-inverse-equality*:
  $x' * x = one \Longrightarrow inverse\ x = (x'::'a::group)$
proof −
  assume *eq*: $x' * x = one$
  have $inverse\ x = one * inverse\ x$
    by (*simp only*: *group-left-one*)
  also have ... = $(x' * x) * inverse\ x$
    by (*simp only*: *eq*)
  also have ... = $x' * (x * inverse\ x)$

```
    by (simp only: group-assoc)
  also have ... = x' * one
    by (simp only: group-right-inverse)
  also have ... = x'
    by (simp only: group-right-one)
  finally show ?thesis .
qed
```

The inverse operation has some further characteristic properties.

```
theorem group-inverse-times:
  inverse (x * y) = inverse y * inverse (x::'a::group)
proof (rule group-inverse-equality)
  show (inverse y * inverse x) * (x * y) = one
  proof -
    have (inverse y * inverse x) * (x * y) =
        (inverse y * (inverse x * x)) * y
      by (simp only: group-assoc)
    also have ... = (inverse y * one) * y
      by (simp only: group-left-inverse)
    also have ... = inverse y * y
      by (simp only: group-right-one)
    also have ... = one
      by (simp only: group-left-inverse)
    finally show ?thesis .
  qed
qed

theorem inverse-inverse: inverse (inverse x) = (x::'a::group)
proof (rule group-inverse-equality)
  show x * inverse x = one
    by (simp only: group-right-inverse)
qed

theorem inverse-inject: inverse x = inverse y ==> x = (y::'a::group)
proof -
  assume eq: inverse x = inverse y
  have x = x * one
    by (simp only: group-right-one)
  also have ... = x * (inverse y * y)
    by (simp only: group-left-inverse)
  also have ... = x * (inverse x * y)
    by (simp only: eq)
  also have ... = (x * inverse x) * y
    by (simp only: group-assoc)
  also have ... = one * y
    by (simp only: group-right-inverse)
  also have ... = y
    by (simp only: group-left-one)
  finally show ?thesis .
```

qed

end

7 Summing natural numbers

theory *Summation*
imports *Main*
begin[7]

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.

7.1 Summation laws

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

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

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

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

[7] This example is somewhat reminiscent of the http://isabelle.in.tum.de/library/HOL/ex/NatSum.html, which is discussed in [6] in the context of permutative rewrite rules and ordered rewriting.

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:
  (∑ i::nat=0..<n. 2 * i + 1) = n^Suc (Suc 0)
  (is ?P n is ?S n = -)
proof (induct n)
  show ?P 0 by simp
next
  fix n have ?S (n + 1) = ?S n + 2 * n + 1 by simp
  also assume ?S n = n^Suc (Suc 0)
  also have ... + 2 * n + 1 = (n + 1)^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 * (∑ i::nat=0..n. i^Suc (Suc 0)) = n * (n + 1) * (2 * n + 1)
  (is ?P n is ?S n = -)
proof (induct n)
  show ?P 0 by simp
next
  fix n have ?S (n + 1) = ?S n + 6 * (n + 1)^Suc (Suc 0)
    by (simp add: distrib)
  also assume ?S n = n * (n + 1) * (2 * n + 1)
  also have ... + 6 * (n + 1)^Suc (Suc 0) =
    (n + 1) * (n + 2) * (2 * (n + 1) + 1) by (simp add: distrib)
  finally show ?P (Suc n) by simp
qed
```

```
theorem sum-of-cubes:
  4 * (∑ i::nat=0..n. i^3) = (n * (n + 1))^Suc (Suc 0)
```

```
  (is ?P n is ?S n = -)
proof (induct n)
  show ?P 0 by (simp add: power-eq-if)
next
  fix n have ?S (n + 1) = ?S n + 4 * (n + 1) ^3
    by (simp add: power-eq-if distrib)
  also assume ?S n = (n * (n + 1)) ^Suc (Suc 0)
  also have ... + 4 * (n + 1) ^3 = ((n + 1) * ((n + 1) + 1)) ^Suc (Suc 0)
    by (simp add: power-eq-if distrib)
  finally show ?P (Suc n) by simp
qed
```

Comparing these examples with the tactic script version http://isabelle.in.tum.de/library/HOL/ex/NatSum.html, we note an important difference of how induction vs. simplification is applied. While [6, §10] advises for these examples that "induction should not be applied until the goal is in the simplest form" this would be a very bad idea in our setting.

Simplification normalizes all arithmetic expressions involved, producing huge intermediate goals. With applying induction afterwards, the Isar proof text would have to reflect the emerging configuration by appropriate sub-proofs. This would result in badly structured, low-level technical reasoning, without any good idea of the actual point.

As a general rule of good proof style, automatic methods such as *simp* or *auto* should normally be never used as initial proof methods, but only as terminal ones, solving certain goals completely.

end

8 Textbook-style reasoning: the Knaster-Tarski Theorem

theory *KnasterTarski* **imports** *Main* **begin**

8.1 Prose version

According to the textbook [1, pages 93–94], the Knaster-Tarski fixpoint theorem is as follows.[8]

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

Proof. Let $H = \{x \in L \mid f(x) \leq x\}$ and $a = \bigwedge H$. 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

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

$f(a) \leq a$. 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 $a \leq f(a)$.

8.2 Formal versions

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

```
theorem KnasterTarski: mono f ==> EX a::'a set. f a = a
proof
  let ?H = {u. f u <= u}
  let ?a = Inter ?H

  assume mono: mono f
  show f ?a = ?a
  proof -
    {
      fix x
      assume H: x : ?H
      hence ?a <= x by (rule Inter-lower)
      with mono have f ?a <= f x ..
      also from H have ... <= x ..
      finally have f ?a <= x .
    }
    hence ge: f ?a <= ?a by (rule Inter-greatest)
    {
      also presume ... <= f ?a
      finally (order-antisym) show ?thesis .
    }
    from mono ge have f (f ?a) <= f ?a ..
    hence f ?a : ?H by simp
    thus ?a <= f ?a by (rule Inter-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 KnasterTarski': mono f ==> EX a::'a set. f a = a
```

```
proof
  let ?H = {u. f u <= u}
  let ?a = Inter ?H

  assume mono: mono f
  show f ?a = ?a
  proof (rule order-antisym)
    show ge: f ?a <= ?a
    proof (rule Inter-greatest)
      fix x
      assume H: x : ?H
      hence ?a <= x by (rule Inter-lower)
      with mono have f ?a <= f x ..
      also from H have ... <= x ..
      finally show f ?a <= x .
    qed
    show ?a <= f ?a
    proof (rule Inter-lower)
      from mono ge have f (f ?a) <= f ?a ..
      thus f ?a : ?H by simp
    qed
  qed
qed

end
```

9 The Mutilated Checker Board Problem

theory *MutilatedCheckerboard* **imports** *Main* **begin**

The Mutilated Checker Board Problem, formalized inductively. See [7] and http://isabelle.in.tum.de/library/HOL/Induct/Mutil.html for the original tactic script version.

9.1 Tilings

```
inductive-set
  tiling :: 'a set set => 'a set set
  for A :: 'a set set
  where
    empty: {} : tiling A
  | Un: a : A ==> t : tiling A ==> a <= - t ==> a Un t : tiling A
```

The union of two disjoint tilings is a tiling.

```
lemma tiling-Un:
  assumes t : tiling A and u : tiling A and t Int u = {}
  shows t Un u : tiling A
```

```
proof −
  let ?T = tiling A
  from ‹t : ?T› and ‹t Int u = {}›
  show t Un u : ?T
  proof (induct t)
    case empty
    with ‹u : ?T› show {} Un u : ?T by simp
  next
    case (Un a t)
    show (a Un t) Un u : ?T
    proof −
      have a Un (t Un u) : ?T
        using ‹a : A›
      proof (rule tiling.Un)
        from ‹(a Un t) Int u = {}› have t Int u = {} by blast
        then show t Un u: ?T by (rule Un)
        from ‹a <= − t› and ‹(a Un t) Int u = {}›
        show a <= − (t Un u) by blast
      qed
      also have a Un (t Un u) = (a Un t) Un u
        by (simp only: Un-assoc)
      finally show ?thesis .
    qed
  qed
qed
```

9.2 Basic properties of "below"

```
constdefs
  below :: nat => nat set
  below n == {i. i < n}

lemma below-less-iff [iff]: (i: below k) = (i < k)
  by (simp add: below-def)

lemma below-0: below 0 = {}
  by (simp add: below-def)

lemma Sigma-Suc1:
    m = n + 1 ==> below m <*> B = ({n} <*> B) Un (below n <*> B)
  by (simp add: below-def less-Suc-eq) blast

lemma Sigma-Suc2:
    m = n + 2 ==> A <*> below m =
      (A <*> {n}) Un (A <*> {n + 1}) Un (A <*> below n)
  by (auto simp add: below-def)

lemmas Sigma-Suc = Sigma-Suc1 Sigma-Suc2
```

9.3 Basic properties of "evnodd"

constdefs
evnodd :: $(nat * nat)$ *set* => *nat* => $(nat * nat)$ *set*
evnodd A b == *A Int* $\{(i, j).\ (i + j)\ mod\ 2 = b\}$

lemma *evnodd-iff*:
(i, j): *evnodd A b* = $((i, j)$: *A* & $(i + j)\ mod\ 2 = b)$
by (*simp add*: *evnodd-def*)

lemma *evnodd-subset*: *evnodd A b* <= *A*
by (*unfold evnodd-def*, *rule Int-lower1*)

lemma *evnoddD*: *x* : *evnodd A b* ==> *x* : *A*
by (*rule subsetD*, *rule evnodd-subset*)

lemma *evnodd-finite*: *finite A* ==> *finite* (*evnodd A b*)
by (*rule finite-subset*, *rule evnodd-subset*)

lemma *evnodd-Un*: *evnodd* (*A Un B*) *b* = *evnodd A b Un evnodd B b*
by (*unfold evnodd-def*) *blast*

lemma *evnodd-Diff*: *evnodd* (*A* − *B*) *b* = *evnodd A b* − *evnodd B b*
by (*unfold evnodd-def*) *blast*

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

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

9.4 Dominoes

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

lemma *dominoes-tile-row*:
$\{i\}$ <*> *below* $(2 * n)$: *tiling domino*
(**is** *?B n* : *?T*)
proof (*induct n*)
case *0*
show *?case* **by** (*simp add*: *below-0 tiling.empty*)
next
case (*Suc n*)
let *?a* = $\{i\}$ <*> $\{2 * n + 1\}$ *Un* $\{i\}$ <*> $\{2 * n\}$

```
  have ?B (Suc n) = ?a Un ?B n
    by (auto simp add: Sigma-Suc Un-assoc)
  moreover have ... : ?T
  proof (rule tiling.Un)
    have {(i, 2 * n), (i, 2 * n + 1)} : domino
      by (rule domino.horiz)
    also have {(i, 2 * n), (i, 2 * n + 1)} = ?a by blast
    finally show ... : domino .
    show ?B n : ?T by (rule Suc)
    show ?a <= - ?B n by blast
  qed
  ultimately show ?case by simp
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 Un ?B m by (simp add: Sigma-Suc)
  moreover have ... : ?T
  proof (rule tiling-Un)
    show ?t : ?T by (rule dominoes-tile-row)
    show ?B m : ?T by (rule Suc)
    show ?t Int ?B m = {} by blast
  qed
  ultimately show ?case by simp
qed

lemma domino-singleton:
  assumes d: d : domino and b < 2
  shows EX i j. evnodd d b = {(i, j)}  (is ?P d)
  using d
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: d: domino
  shows finite d
  using d
```

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

9.5 Tilings of dominoes

lemma *tiling-domino-finite*:
  assumes *t*: *t* : *tiling domino* (**is** *t* : *?T*)
  shows *finite t* (**is** *?F t*)
  using *t*
proof *induct*
  show *?F* {} **by** (*rule finite.emptyI*)
  fix *a t* **assume** *?F t*
  assume *a* : *domino* **then have** *?F a* **by** (*rule domino-finite*)
  from *this* **and** ‹*?F t*› **show** *?F* (*a Un t*) **by** (*rule finite-UnI*)
qed

lemma *tiling-domino-01*:
  assumes *t*: *t* : *tiling domino* (**is** *t* : *?T*)
  shows *card* (*evnodd t 0*) = *card* (*evnodd t 1*)
  using *t*
proof *induct*
  case *empty*
  show *?case* **by** (*simp add*: *evnodd-def*)
next
  case (*Un a t*)
  let *?e* = *evnodd*
  note *hyp* = ‹*card* (*?e t 0*) = *card* (*?e t 1*)›
    and *at* = ‹*a* <= − *t*›
  have *card-suc*:
    !!*b*. *b* < *2* ==> *card* (*?e* (*a Un t*) *b*) = *Suc* (*card* (*?e t b*))
  proof −
    fix *b* :: *nat* **assume** *b* < *2*
    have *?e* (*a Un t*) *b* = *?e a b Un ?e t b* **by** (*rule evnodd-Un*)
    also obtain *i j* **where** *e*: *?e a b* = {(*i*, *j*)}
    proof −
      from ‹*a* ∈ *domino*› **and** ‹*b* < *2*›
      have *EX i j*. *?e a b* = {(*i*, *j*)} **by** (*rule domino-singleton*)
      then show *?thesis* **by** (*blast intro*: *that*)
    qed
    moreover have ... *Un ?e t b* = *insert* (*i*, *j*) (*?e t b*) **by** *simp*
    moreover have *card* ... = *Suc* (*card* (*?e t b*))
    proof (*rule card-insert-disjoint*)
      from ‹*t* ∈ *tiling domino*› **have** *finite t*
        by (*rule tiling-domino-finite*)
      then show *finite* (*?e t b*)
        by (*rule evnodd-finite*)

```
    from e have (i, j) : ?e a b by simp
    with at show (i, j) ~: ?e t b by (blast dest: evnoddD)
  qed
  ultimately show ?thesis b by simp
 qed
 then have card (?e (a Un 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 Un t) 1)
  by simp
 finally show ?case .
qed
```

9.6 Main theorem

```
constdefs
 mutilated-board :: nat => nat => (nat * nat) set
 mutilated-board m n ==
  below (2 * (m + 1)) <*> below (2 * (n + 1))
   - {(0, 0)} - {(2 * m + 1, 2 * n + 1)}

theorem mutil-not-tiling: mutilated-board m n ~: tiling domino
proof (unfold mutilated-board-def)
 let ?T = tiling domino
 let ?t = below (2 * (m + 1)) <*> below (2 * (n + 1))
 let ?t' = ?t - {(0, 0)}
 let ?t'' = ?t' - {(2 * m + 1, 2 * n + 1)}

 show ?t'' ~: ?T
 proof
  have t: ?t : ?T by (rule dominoes-tile-matrix)
  assume t'': ?t'' : ?T

  let ?e = evnodd
  have fin: finite (?e ?t 0)
   by (rule evnodd-finite, rule tiling-domino-finite, rule t)

  note [simp] = evnodd-iff evnodd-empty evnodd-insert evnodd-Diff
  have card (?e ?t'' 0) < card (?e ?t' 0)
  proof -
   have card (?e ?t' 0 - {(2 * m + 1, 2 * n + 1)})
     < card (?e ?t' 0)
   proof (rule card-Diff1-less)
    from - fin show finite (?e ?t' 0)
     by (rule finite-subset) auto
    show (2 * m + 1, 2 * n + 1) : ?e ?t' 0 by simp
   qed
   then show ?thesis by simp
  qed
  also have ... < card (?e ?t 0)
```

```
    proof −
      have (0, 0) : ?e ?t 0 by simp
      with fin have card (?e ?t 0 − {(0, 0)}) < card (?e ?t 0)
        by (rule card-Diff1-less)
      then show ?thesis by simp
    qed
    also from t have ... = card (?e ?t 1)
      by (rule tiling-domino-01)
    also have ?e ?t 1 = ?e ?t'' 1 by simp
    also from t'' have card ... = card (?e ?t'' 0)
      by (rule tiling-domino-01 [symmetric])
    finally have ... < ... . then show False ..
  qed
qed

end
```

10 An old chestnut

theory *Puzzle* **imports** *Main* **begin**[9]

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

```
theorem
  assumes f-ax: ⋀n. f (f n) < f (Suc n)
  shows f n = n
proof (rule order-antisym)
  {
    fix n show n ≤ f n
    proof (induct k ≡ f n arbitrary: n rule: less-induct)
      case (less k n)
      then have hyp: ⋀m. f m < f n ⟹ m ≤ f m by (simp only:)
      show n ≤ f n
      proof (cases n)
        case (Suc m)
        from f-ax have f (f m) < f n by (simp only: Suc)
        with hyp have f m ≤ f (f m) .
        also from f-ax have ... < f n by (simp only: Suc)
        finally have f m < f n .
        with hyp have m ≤ f m .
        also note ‹... < f n›
        finally have m < f n .
        then have n ≤ f n by (simp only: Suc)
        then show ?thesis .
      next
```

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

```
      case 0
      then show ?thesis by simp
    qed
  qed
} note ge = this

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

show f n ≤ n
proof -
  have ¬ n < f n
  proof
    assume n < f n
    then have Suc n ≤ f n by simp
    then have f (Suc n) ≤ f (f n) by (rule mono)
    also have ... < f (Suc n) by (rule f-ax)
    finally have ... < ... . then show False ..
  qed
  then show ?thesis by simp
qed
qed

end
```

11 Nested datatypes

theory *NestedDatatype* **imports** *Main* **begin**

11.1 Terms and substitution

datatype (*'a*, *'b*) *term* =
  Var 'a
| *App 'b* (*'a*, *'b*) *term list*

consts
  subst-term :: (*'a* => (*'a*, *'b*) *term*) => (*'a*, *'b*) *term* => (*'a*, *'b*) *term*
  subst-term-list ::
    (*'a* => (*'a*, *'b*) *term*) => (*'a*, *'b*) *term list* => (*'a*, *'b*) *term list*

primrec (*subst*)
  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*

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* = λ*ts*. *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*
  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

11.2 Alternative induction

theorem *term-induct'* [*case-names Var App*]:
  assumes *var*: !!*a*. *P* (*Var a*)
    and *app*: !!*b ts*. *list-all P ts* ==> *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** *list-all P ts*
  then show *P* (*App b ts*) **by** (*rule app*)
next
  show *list-all P* [] **by** *simp*
next
  fix *t ts* **assume** *P t list-all P ts*
  then show *list-all P* (*t* # *ts*) **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

12 Hoare Logic

theory *Hoare* **imports** *Main*
uses (~~/*src*/*HOL*/*Hoare*/*hoare-tac.ML*) **begin**

12.1 Abstract syntax and semantics

The following abstract syntax and semantics of Hoare Logic over `WHILE` programs closely follows the existing tradition in Isabelle/HOL of formalizing the presentation given in [10, §6]. See also http://isabelle.in.tum.de/library/Hoare/ and [3].

types
  'a bexp = *'a set*
  '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

types
  'a sem = 'a => 'a => bool

consts
  iter :: nat => 'a bexp => 'a sem => 'a sem
primrec
  iter 0 b S s s' = (s ~: b & s = s')
  iter (Suc n) b S s s' =
    (s : b & (EX s''. S s s'' & iter n b S s'' s'))

consts
  Sem :: 'a com => 'a sem
primrec
  Sem (Basic f) s s' = (s' = f s)
  Sem (c1; c2) s s' = (EX 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' = (EX n. iter n b (Sem c) s s')

constdefs
  Valid :: 'a bexp => 'a com => 'a bexp => bool
    ((3|- -/ (2-)/ -) [100, 55, 100] 50)
  |- P c Q == ALL s s'. Sem c s s' --> s : P --> s' : Q

syntax (xsymbols)
  Valid :: 'a bexp => 'a com => 'a bexp => bool
    ((3⊢ -/ (2-)/ -) [100, 55, 100] 50)

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)
```

12.2 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 §12.4 and §12.5.

The *basic* rule represents any kind of atomic access to the state space. This subsumes the common rules of *skip* and *assign*, as formulated in §12.4.

theorem *basic*: $\vdash \{s.\ f\ s : P\}\ (Basic\ f)\ P$
proof
  fix $s\ s'$ **assume** *s*: $s : \{s.\ f\ s : P\}$
  assume $Sem\ (Basic\ f)\ s\ s'$
  hence $s' = f\ s$ **by** *simp*
  with s **show** $s' : P$ **by** *simp*
qed

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

theorem *seq*: $\vdash P\ c1\ Q \Longrightarrow \vdash Q\ c2\ R \Longrightarrow \vdash P\ (c1;\ c2)\ R$
proof
  assume *cmd1*: $\vdash P\ c1\ Q$ **and** *cmd2*: $\vdash Q\ c2\ R$
  fix $s\ s'$ **assume** *s*: $s : P$
  assume $Sem\ (c1;\ c2)\ s\ s'$
  then obtain s'' **where** *sem1*: $Sem\ c1\ s\ s''$ **and** *sem2*: $Sem\ c2\ s''\ s'$
    by *auto*
  from *cmd1 sem1 s* **have** $s'' : Q$ **..**
  with *cmd2 sem2* **show** $s' : R$ **..**
qed

theorem *conseq*: $P' <= P \Longrightarrow \vdash P\ c\ Q \Longrightarrow Q <= Q' \Longrightarrow \vdash P'\ c\ Q'$
proof
  assume $P'P$: $P' <= P$ **and** QQ': $Q <= Q'$
  assume *cmd*: $\vdash P\ c\ Q$
  fix $s\ s' :: 'a$
  assume *sem*: $Sem\ c\ s\ s'$
  assume $s : P'$ **with** $P'P$ **have** $s : P$ **..**
  with *cmd sem* **have** $s' : Q$ **..**
  with QQ' **show** $s' : Q'$ **..**
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*:
  $\vdash (P\ Int\ b)\ c1\ Q \Longrightarrow \vdash (P\ Int\ -b)\ c2\ Q \Longrightarrow \vdash P\ (Cond\ b\ c1\ c2)\ Q$
proof
  assume *case-b*: $\vdash (P\ Int\ b)\ c1\ Q$ **and** *case-nb*: $\vdash (P\ Int\ -b)\ c2\ Q$
  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 Int 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 Int -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: |- (P Int b) c P
 shows |- P (While b X c) (P Int -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 Int -b
 proof (induct n arbitrary: s)
  case 0
  thus ?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 Int b by simp
  with body sem have s'' : P ..
  with iter show ?case by (rule Suc)
 qed
qed
```

12.3 Concrete syntax for assertions

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

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

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

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

syntax
```
  -quote       :: 'b => ('a => 'b)       ((.'(-').) [0] 1000)
  -antiquote   :: ('a => 'b) => 'b       (´- [1000] 1000)
  -Subst       :: 'a bexp ⇒ 'b ⇒ idt ⇒ 'a bexp
       (-[-'/´-] [1000] 999)
  -Assert      :: 'a => 'a set           ((.{-}.) [0] 1000)
  -Assign      :: idt => 'b => 'a com    ((´- :=/ -) [70, 65] 61)
  -Cond        :: 'a bexp => 'a com => 'a com => 'a com
       ((0IF -/ THEN -/ ELSE -/ FI) [0, 0, 0] 61)
  -While-inv   :: 'a bexp => 'a assn => 'a com => 'a com
       ((0WHILE -/ INV - //DO - /OD)  [0, 0, 0] 61)
  -While       :: 'a bexp => 'a com => 'a com
       ((0WHILE - //DO - /OD)  [0, 0] 61)
```

syntax (*xsymbols*)
```
  -Assert      :: 'a => 'a set           ((⦃-⦄) [0] 1000)
```

translations
```
  .{b}.                 => Collect .(b).
  B [a/´x]               => .{´(-update-name x (λ-. a)) ∈ B}.
  ´x := a                => Basic .(´(-update-name x (λ-. a))).
  IF b THEN c1 ELSE c2 FI => Cond .{b}. c1 c2
  WHILE b INV i DO c OD  => While .{b}. i c
  WHILE b DO c OD        == WHILE b INV arbitrary DO c OD
```

parse-translation ⟪
```
  let
    fun quote-tr [t] = Syntax.quote-tr -antiquote t
      | quote-tr ts = raise TERM (quote-tr, ts);
  in [(-quote, quote-tr)] end
```
⟫

As usual in Isabelle syntax translations, the part for printing is more complicated — we cannot express parts as macro rules as above. Don't look here, unless you have to do similar things for yourself.

print-translation ⟪
```
  let
    fun quote-tr' f (t :: ts) =
          Term.list-comb (f $ Syntax.quote-tr' -antiquote t, ts)
      | quote-tr' - - = raise Match;
```

```
  val assert-tr' = quote-tr' (Syntax.const -Assert);

  fun bexp-tr' name ((Const (Collect, -) $ t) :: ts) =
        quote-tr' (Syntax.const name) (t :: ts)
    | bexp-tr' - - = raise Match;

  fun upd-tr' (x-upd, T) =
    (case try (unsuffix RecordPackage.updateN) x-upd of
      SOME x => (x, if T = dummyT then T else Term.domain-type T)
    | NONE => raise Match);

  fun update-name-tr' (Free x) = Free (upd-tr' x)
    | update-name-tr' ((c as Const (-free, -)) $ Free x) =
        c $ Free (upd-tr' x)
    | update-name-tr' (Const x) = Const (upd-tr' x)
    | update-name-tr' - = raise Match;

  fun K-tr' (Abs (-,-,t)) = if null (loose-bnos t) then t else raise Match
    | K-tr' (Abs (-,-,Abs (-,-,t)$Bound 0)) = if null (loose-bnos t) then t else raise
Match
    | K-tr' - = raise Match;

  fun assign-tr' (Abs (x, -, f $ k $ Bound 0) :: ts) =
        quote-tr' (Syntax.const -Assign $ update-name-tr' f)
          (Abs (x, dummyT, K-tr' k) :: ts)
    | assign-tr' - = raise Match;
in
  [(Collect, assert-tr'), (Basic, assign-tr'),
    (Cond, bexp-tr' -Cond), (While, bexp-tr' -While-inv)]
end
⟫
```

12.4 Rules for single-step proof

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

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

```
lemma [trans]: |- P c Q ==> P' <= P ==> |- P' c Q
  by (unfold Valid-def) blast
lemma [trans] : P' <= P ==> |- P c Q ==> |- P' c Q
  by (unfold Valid-def) blast

lemma [trans]: Q <= Q' ==> |- P c Q ==> |- P c Q'
  by (unfold Valid-def) blast
lemma [trans]: |- P c Q ==> Q <= Q' ==> |- P c Q'
```

by (*unfold Valid-def*) *blast*

lemma [*trans*]:
  $|-.\{'P\}.\ c\ Q ==> (!!s.\ P'\ s --> P\ s) ==> |-.\{'P'\}.\ c\ Q$
by (*simp add*: *Valid-def*)
lemma [*trans*]:
  $(!!s.\ P'\ s --> P\ s) ==> |-.\{'P\}.\ c\ Q ==> |-.\{'P'\}.\ c\ Q$
by (*simp add*: *Valid-def*)

lemma [*trans*]:
  $|-\ P\ c\ .\{'Q\}. ==> (!!s.\ Q\ s --> Q'\ s) ==> |-\ P\ c\ .\{'Q'\}.$
by (*simp add*: *Valid-def*)
lemma [*trans*]:
  $(!!s.\ Q\ s --> Q'\ s) ==> |-\ P\ c\ .\{'Q\}. ==> |-\ P\ c\ .\{'Q'\}.$
by (*simp add*: *Valid-def*)

Identity and basic assignments.[10]

lemma *skip* [*intro?*]: $|-\ P\ SKIP\ P$
proof −
  have $|-\ \{s.\ id\ s : P\}\ SKIP\ P$ **by** (*rule basic*)
  thus *?thesis* **by** *simp*
qed

lemma *assign*: $|-\ P\ ['a/'x]\ 'x := 'a\ P$
by (*rule basic*)

Note that above formulation of assignment corresponds to our preferred way to model state spaces, using (extensible) record types in HOL [2]. For any record field x, Isabelle/HOL provides a functions x (selector) and x-*update* (update). Above, there is only a place-holder appearing for the latter kind of function: due to concrete syntax $\acute{x} := \acute{a}$ also contains *x_update*.[11]

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

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

lemma *seq-assoc* [*simp*]: $(\ |-\ P\ c1;(c2;c3)\ Q) = (\ |-\ P\ (c1;c2);c3\ Q)$
by (*auto simp add*: *Valid-def*)

Conditional statements.

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

lemma [*trans*, *intro?*]:

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

[11] 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.

```
|- .{´P & ´b}. c1 Q
   ==> |- .{´P & ~ ´b}. c2 Q
   ==> |- .{´P}. IF ´b THEN c1 ELSE c2 FI Q
  by (rule cond) (simp-all add: Valid-def)
```

While statements — with optional invariant.

```
lemma [intro?]:
   |- (P Int b) c P ==> |- P (While b P c) (P Int -b)
  by (rule while)

lemma [intro?]:
   |- (P Int b) c P ==> |- P (While b arbitrary c) (P Int -b)
  by (rule while)


lemma [intro?]:
  |- .{´P & ´b}. c .{´P}.
   ==> |- .{´P}. WHILE ´b INV .{´P}. DO c OD .{´P & ~ ´b}.
  by (simp add: while Collect-conj-eq Collect-neg-eq)

lemma [intro?]:
  |- .{´P & ´b}. c .{´P}.
   ==> |- .{´P}. WHILE ´b DO c OD .{´P & ~ ´b}.
  by (simp add: while Collect-conj-eq Collect-neg-eq)
```

12.5 Verification conditions

We now load the *original* ML file for proof scripts and tactic definition for the Hoare Verification Condition Generator (see http://isabelle.in.tum.de/library/Hoare/). As far as we are concerned here, the result is a proof method *hoare*, which may be applied to a Hoare Logic assertion to extract purely logical verification conditions. It is important to note that the method requires `WHILE` loops to be fully annotated with invariants beforehand. Furthermore, only *concrete* pieces of code are handled — the underlying tactic fails ungracefully if supplied with meta-variables or parameters, for example.

```
lemma SkipRule: p ⊆ q ⟹ Valid p (Basic id) q
  by (auto simp add: Valid-def)

lemma BasicRule: p ⊆ {s. f s ∈ q} ⟹ Valid p (Basic f) q
  by (auto simp: Valid-def)

lemma SeqRule: Valid P c1 Q ⟹ Valid Q c2 R ⟹ Valid P (c1;c2) R
  by (auto simp: Valid-def)

lemma CondRule:
  p ⊆ {s. (s ∈ b ⟶ s ∈ w) ∧ (s ∉ b ⟶ s ∈ w')}
   ⟹ Valid w c1 q ⟹ Valid w' c2 q ⟹ Valid p (Cond b c1 c2) q
  by (auto simp: Valid-def)
```

lemma *iter-aux*:
 $\forall s\ s'.\ Sem\ c\ s\ s' --> s : I\ \&\ s : b --> s' : I ==>$
 $(\bigwedge s\ s'.\ s : I \Longrightarrow iter\ n\ b\ (Sem\ c)\ s\ s' \Longrightarrow s' : I\ \&\ s' \sim: b)$
 apply(*induct n*)
 apply *clarsimp*
 apply (*simp* (*no-asm-use*))
 apply *blast*
 done

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

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

use ~~/*src/HOL/Hoare/hoare-tac.ML*

method-setup *hoare* = ⟪
 Method.no-args
 (*Method.SIMPLE-METHOD'*
 (*hoare-tac* (*simp-tac* (*HOL-basic-ss addsimps* [@{*thm Record.K-record-comp*}]
))))⟫
 verification condition generator for Hoare logic

end

13 Using Hoare Logic

theory *HoareEx* **imports** *Hoare* **begin**

13.1 State spaces

First of all we provide a store of program variables that occur in any of the programs considered later. Slightly unexpected things may happen when attempting to work with undeclared variables.

record *vars* =
 I :: *nat*
 M :: *nat*
 N :: *nat*

$S :: nat$

While all of our variables happen to have the same type, nothing would prevent us from working with many-sorted programs as well, or even polymorphic ones. Also note that Isabelle/HOL's extensible record types even provides simple means to extend the state space later.

13.2 Basic examples

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

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

lemma
$\vdash .\{\acute{}(N\text{-}update\ (\lambda\text{-}.\ (2 * \acute{}N))) : .\{\acute{}N = 10\}.\}.\ \acute{}N := 2 * \acute{}N\ .\{\acute{}N = 10\}.$
by (*rule assign*)

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

lemma $\vdash .\{True\}.\ \acute{}N := 10\ .\{\acute{}N = 10\}.$
by *hoare*

lemma $\vdash .\{2 * \acute{}N = 10\}.\ \acute{}N := 2 * \acute{}N\ .\{\acute{}N = 10\}.$
by *hoare*

lemma $\vdash .\{\acute{}N = 5\}.\ \acute{}N := 2 * \acute{}N\ .\{\acute{}N = 10\}.$
by *hoare simp*

lemma $\vdash .\{\acute{}N + 1 = a + 1\}.\ \acute{}N := \acute{}N + 1\ .\{\acute{}N = a + 1\}.$
by *hoare*

lemma $\vdash .\{\acute{}N = a\}.\ \acute{}N := \acute{}N + 1\ .\{\acute{}N = a + 1\}.$
by *hoare simp*

lemma $\vdash .\{a = a\ \&\ b = b\}.\ \acute{}M := a;\ \acute{}N := b\ .\{\acute{}M = a\ \&\ \acute{}N = b\}.$
by *hoare*

lemma $\vdash .\{True\}.\ \acute{}M := a;\ \acute{}N := b\ .\{\acute{}M = a\ \&\ \acute{}N = b\}.$
by *hoare simp*

lemma
$\vdash .\{\acute{}M = a\ \&\ \acute{}N = b\}.$
$\acute{}I := \acute{}M;\ \acute{}M := \acute{}N;\ \acute{}N := \acute{}I$
$.\{\acute{}M = b\ \&\ \acute{}N = a\}.$
by *hoare simp*

It is important to note that statements like the following one can only be proven for each individual program variable. Due to the extra-logical nature

of record fields, we cannot formulate a theorem relating record selectors and updates schematically.

lemma $|- .\{\acute{}N = a\}. \acute{}N := \acute{}N .\{\acute{}N = a\}.$
  by *hoare*

lemma $|- .\{\acute{}x = a\}. \acute{}x := \acute{}x .\{\acute{}x = a\}.$
  ⋮

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 $|- .\{\acute{}M = \acute{}N\}. \acute{}M := \acute{}M + 1 .\{\acute{}M \sim= \acute{}N\}.$
proof −
  have $.\{\acute{}M = \acute{}N\}. <= .\{\acute{}M + 1 \sim= \acute{}N\}.$
    by *auto*
  also have $|- ... \acute{}M := \acute{}M + 1 .\{\acute{}M \sim= \acute{}N\}.$
    by *hoare*
  finally show *?thesis* **.**
qed

lemma $|- .\{\acute{}M = \acute{}N\}. \acute{}M := \acute{}M + 1 .\{\acute{}M \sim= \acute{}N\}.$
proof −
  have $!!m\ n::nat.\ m = n \dashrightarrow m + 1 \sim= n$
    — inclusion of assertions expressed in "pure" logic,
    — without mentioning the state space
    by *simp*
  also have $|- .\{\acute{}M + 1 \sim= \acute{}N\}. \acute{}M := \acute{}M + 1 .\{\acute{}M \sim= \acute{}N\}.$
    by *hoare*
  finally show *?thesis* **.**
qed

lemma $|- .\{\acute{}M = \acute{}N\}. \acute{}M := \acute{}M + 1 .\{\acute{}M \sim= \acute{}N\}.$
  by *hoare simp*

13.3 Multiplication by addition

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

lemma
  $|- .\{\acute{}M = 0\ \&\ \acute{}S = 0\}.$

```
    WHILE ´M ~= a
    DO ´S := ´S + b; ´M := ´M + 1 OD
    .{´S = a * b}.
proof −
  let |− - ?while - = ?thesis
  let .{´?inv}. = .{´S = ´M * b}.

  have .{´M = 0 & ´S = 0}. <= .{´?inv}. by auto
  also have |− ... ?while .{´?inv & ~ (´M ~= a)}.
  proof
    let ?c = ´S := ´S + b; ´M := ´M + 1
    have .{´?inv & ´M ~= a}. <= .{´S + b = (´M + 1) * b}.
      by auto
    also have |− ... ?c .{´?inv}. by hoare
    finally show |− .{´?inv & ´M ~= a}. ?c .{´?inv}. .
  qed
  also have ... <= .{´S = a * b}. by auto
  finally show ?thesis .
qed
```

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

```
lemma
  |− .{´M = 0 & ´S = 0}.
    WHILE ´M ~= a
    INV .{´S = ´M * b}.
    DO ´S := ´S + b; ´M := ´M + 1 OD
    .{´S = a * b}.
  by hoare auto
```

13.4 Summing natural numbers

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

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

```
declare atLeast0LessThan[symmetric,simp]

theorem
  |− .{True}.
    ´S := 0; ´I := 1;
    WHILE ´I ~= n
    DO
```

```
      ´S := ´S + ´I;
      ´I := ´I + 1
    OD
    .{´S = (SUM j<n. j)}.
 (is |- - (-; ?while) -)
proof -
 let ?sum = λk::nat. SUM j<k. j
 let ?inv = λs i::nat. s = ?sum i

 have |- .{True}. ´S := 0; ´I := 1 .{?inv ´S ´I}.
 proof -
  have True --> 0 = ?sum 1
   by simp
  also have |- .{...}. ´S := 0; ´I := 1 .{?inv ´S ´I}.
   by hoare
  finally show ?thesis .
 qed
 also have |- ... ?while .{?inv ´S ´I & ~ ´I ~= n}.
 proof
  let ?body = ´S := ´S + ´I; ´I := ´I + 1
  have !!s i. ?inv s i & i ~= n --> ?inv (s + i) (i + 1)
   by simp
  also have |- .{´S + ´I = ?sum (´I + 1)}. ?body .{?inv ´S ´I}.
   by hoare
  finally show |- .{?inv ´S ´I & ´I ~= n}. ?body .{?inv ´S ´I}. .
 qed
 also have !!s i. s = ?sum i & ~ i ~= n --> s = ?sum n
  by simp
 finally show ?thesis .
qed
```

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

```
theorem
 |- .{True}.
    ´S := 0; ´I := 1;
    WHILE ´I ~= n
    INV .{´S = (SUM j<´I. j)}.
    DO
      ´S := ´S + ´I;
      ´I := ´I + 1
    OD
    .{´S = (SUM j<n. j)}.
proof -
 let ?sum = λk::nat. SUM j<k. j
 let ?inv = λs i::nat. s = ?sum i

 show ?thesis
 proof hoare
```

```
  show ?inv 0 1 by simp
 next
  fix s i assume ?inv s i & i ~= n
  thus ?inv (s + i) (i + 1) by simp
 next
  fix s i assume ?inv s i & ~ i ~= n
  thus s = ?sum n by simp
 qed
qed
```

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

```
theorem
 |- .{True}.
    ´S := 0; ´I := 1;
    WHILE ´I ~= n
    INV .{´S = (SUM j<´I. j)}.
    DO
      ´S := ´S + ´I;
      ´I := ´I + 1
    OD
    .{´S = (SUM j<n. j)}.
 by hoare auto
```

13.5 Time

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

```
record tstate = time :: nat

types 'a time = (|time :: nat, ... :: 'a|)

consts timeit :: 'a time com => 'a time com
primrec
 timeit (Basic f) = (Basic f; Basic(λs. s(|time := Suc (time s)|)))
 timeit (c1; c2) = (timeit c1; timeit c2)
 timeit (Cond b c1 c2) = Cond b (timeit c1) (timeit c2)
 timeit (While b iv c) = While b iv (timeit c)

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

lemma lem: (0::nat) < n ==> 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;
   ´I := ´I − 1
 OD
) .{2*´time = i*i + 5*i}.
 apply simp
 apply hoare
     apply simp
    apply clarsimp
   apply clarsimp
  apply arith
  prefer 2
  apply clarsimp
 apply (clarsimp simp: nat-distrib)
 apply (frule lem)
 apply arith
 done

end
```

References

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

[2] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in Higher-Order Logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *LNCS*, 1998.

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

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle's Logics: HOL*.

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

[6] L. C. Paulson. *The Isabelle Reference Manual*.

[7] L. C. Paulson. A simple formalization and proof for the mutilated chess board. Technical Report 394, Comp. Lab., Univ. Camb., 1996. http://www.cl.cam.ac.uk/users/lcp/papers/Reports/mutil.pdf.

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

[9] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, LNCS 1690, 1999.

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