Computer-supported Modeling and Reasoning — *Exercises and Solutions* — (Isabelle 2004)

Achim D. Brucker Burkhart Wolff
{brucker,bwolff}@inf.ethz.ch

Information Security
ETH-Zentrum
CH-8092 Zürich
Switzerland

1 Propositional Logic

In this lecture you will deepen your knowledge about *propositional logic*, you will prove your first theorems in an interactive theorem prover (Isabelle) and see how paper-and-pencil proofs are related to interactive theorem proving. In particular you will learn how to do forward-style and backward-style proofs (using Isabelle) and how to combine these two techniques.

1.1 Isabelle in a Nutshell

Isabelle is an interactive theorem prover. During an Isabelle session, you will construct proofs of theorems. A proof consists of a number of proof steps, and the Isabelle system will ensure that each step is correct, and thus ultimately that the entire proof is correct. Various degrees of automation can be realized in Isabelle: you can write each step of a proof yourself, or you can let the system do big subproofs or even the entire proof automatically. In the beginning, we will do the former, because we want to understand in detail what a proof looks like.

In the lecture we will use Isabelle 2004. [1] The graphical user interface (an instance of Proof General) is based on the editor (X)Emacs and can be started by typing[2] Proof General

```
Isabelle-2004
```

in a shell. An special configured (X)Emacs will start showing several Isabelle and Proof General related menus.

Hint: For a nice rendering of mathematical symbols, you should enable the X-Symbol package. For doing so, select the box ⟨`Proof-General` ▷ X-Symbol

[1] Isabelle is only supported on Unix-like operating systems (e.g. Linux, Solaris, MacOS X). You can download Isabelle from `http://isabelle.in.tum.de`. If you use Windows and do not want to install Linux on your hard disk, we recommend IsaMorph (`http://www.brucker.ch/projects/isamorph/`) which is a CD-based Linux that already provides Isabelle.

[2] If you have installed Isabelle yourself or if you are using IsaMorph, just type `Isabelle` instead of `Isabelle-2004`.

⟨Options ▷ X-Symbol⟩⟩. Now select ⟨Proof-General ▷ ⟨Options ▷ Save Options⟩⟩ to enable X-Symbol automatically on every startup.

Isabelle supports a variety of different logics, thus, before we can prove our first theorem, we have to choose the logic we want. As Isabelle does not provide a special setup for propositional logic, we choose first-order logic (FOL) by selecting ⟨Isabelle/Isar ▷ ⟨Logics ▷ FOL⟩⟩. FOL is a superset of propositional logic.

FOL

Hint: If you do not want to use the "default" logic (normally higher-order logic (HOL), you must select the logic on every startup of Isabelle.

We are now ready to prove theorems in propositional logic using Isabelle. While doing so, we have to keep several things in mind:

Isabelle rule names

- The rule names for propositional logic used by Isabelle differ from the names used in the lecture. For an overview of the rule names used by Isabelle, see Tab. 1.1.

theory

- Whenever we prove something in Isabelle (or in a paper and pencil fashion), we do so in the context of a *theory*. The essential parts of a theory are the definition of some syntax and judgments that are postulated to be true. In Isabelle, this theory is contained in a file whose name ends in `.thy`. We can start a new theory (building upon FOL) named `ex1` by creating a file with the first line

  theory ex1 = FOL:

symbol representation

- Isabelle uses several concrete syntaxes to represent mathematical symbols (see Tab. 1.2). With enabled X-Symbol mode, you will see the the mathematical symbols in the output of Isabelle. You can enter these symbols either by typing their ASCII representation (most of the symbols will be automatically converted to their mathematical representation) or entering their internal name. Also selecting the symbols within the ⟨X-Symbol ▷ ...⟩ menu is possible.

1.2 Our first theorem

Open a new file[3] `simple.thy` and enter the following skeleton of a theory file:

[3] Click on the menu ⟨File ▷ Open⟩ and enter `simple.thy` into the (X)Emacs Minibuffer (the very last line of the (X)Emacs window).

$$\frac{A \quad B}{A \land B}\ \texttt{conjI} \qquad \frac{A \land B}{A}\ \texttt{conjunct1} \qquad \frac{A \land B}{B}\ \texttt{conjunct2}$$

$$\frac{A}{A \lor B}\ \texttt{disjI1} \qquad \frac{B}{A \lor B}\ \texttt{disjI2} \qquad \frac{A \lor B \quad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \quad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}\ \texttt{disjE}$$

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \longrightarrow B}\ \texttt{impI} \qquad \frac{A \longrightarrow B \quad A}{B}\ \texttt{mp} \qquad \frac{}{A}\ \texttt{FalseE}$$

Table 1.1: Propositional Logic in Isabelle

<table>
<tr><td>$[\![$</td><td>[|</td><td>\<lbrakk></td><td>$\exists!$</td><td>EX!, :?!</td><td>\<exists>!</td></tr>
<tr><td>$]\!]$</td><td>|]</td><td>\<rbrakk></td><td>ϵ</td><td>SOME, @</td><td>\<epsilon></td></tr>
<tr><td>$\Longrightarrow$</td><td>==></td><td>\<Longrightarrow></td><td>$\circ$</td><td>o</td><td>\<circ></td></tr>
<tr><td>$\bigwedge$</td><td>!!</td><td>\<And></td><td>$\vert\ \vert$</td><td>abs</td><td>\<bar> \<bar></td></tr>
<tr><td>$\equiv$</td><td>==</td><td>\<equiv></td><td>$\le$</td><td><=</td><td>\<le></td></tr>
<tr><td>$\rightleftharpoons$</td><td>==</td><td>\<rightleftharpoons></td><td>$\times$</td><td>*</td><td>\<times></td></tr>
<tr><td>$\rightharpoonup$</td><td>=></td><td>\<rightharpoonup></td><td>$\in$</td><td>:</td><td>\<in></td></tr>
<tr><td>$\leftharpoondown$</td><td><=</td><td>\<leftharpoondown></td><td>$\notin$</td><td>~:</td><td>\<notin></td></tr>
<tr><td>λ</td><td>%</td><td>\<lambda></td><td>$\subseteq$</td><td><=</td><td>\<subseteq></td></tr>
<tr><td>$\Rightarrow$</td><td>=></td><td>\<Rightarrow></td><td>$\subset$</td><td><</td><td>\<subset></td></tr>
<tr><td>$\land$</td><td>&</td><td>\<and></td><td>$\cup$</td><td>Un</td><td>\<union></td></tr>
<tr><td>$\lor$</td><td>|</td><td>\<or></td><td>$\cap$</td><td>Int</td><td>\<inter></td></tr>
<tr><td>$\longrightarrow$</td><td>--></td><td>\<longrightarrow></td><td>$\bigcup$</td><td>UN,UNION</td><td>\<Union></td></tr>
<tr><td>$\neg$</td><td>~</td><td>\<not></td><td>$\bigcap$</td><td>INT, Inter</td><td>\<Inter></td></tr>
<tr><td>$\neq$</td><td>~=</td><td>\<noteq></td><td>*</td><td>^*</td><td>\<^sup>*</td></tr>
<tr><td>$\forall$</td><td>ALL, !</td><td>\<forall></td><td>$^{-1}$</td><td>^-1</td><td>\<inverse></td></tr>
<tr><td>$\exists$</td><td>EX, ?</td><td>\<exists></td><td></td><td></td><td></td></tr>
</table>

Table 1.2: Mathematical Symbols, Their ASCII-Equivalents and Internal Names

theory simple = FOL:

end

Hint: Isabelle requires, that the file name (without extension) is identical to the theory name.

You can now start the Isabelle process by clicking on the ⟨`Next`⟩ button; after a short startup time, the first line of your theory should be highlighted.

1.2.1 Backward-Style Reasoning in Isabelle

We will now prove $A \longrightarrow (B \longrightarrow A)$ in backward style. Therefore we begin by entering our proof goal

lemma

lemma first_theorem: "A --> (B --> A)"

as second line of your theory and click on ⟨`Next`⟩ (this processes your theory one step further). Now, Isabelle will also highlight this line and also will repeat the proof goal in its output window. As you know from the lecture, we have to apply $\rightarrow$-I as first proof step. Using Tab. 1.1 we see, that $\rightarrow$-I is called `impI` in Isabelle. You can look up Isabelle's definition of `impI` by clicking on the ⟨`Command`⟩ button and entering

impI

impI

thm

thm impI

Minibuffer

in the (X)Emacs Minibuffer (the very last line of the (X)Emacs window). Isabelle will print its version of the implication introduction rule in its output area. We apply this rule to the current proof state by writing

apply

rule

apply (*rule* impI)

and processing the theory one step. Can you explain the proof step after executing this rule? Applying `impI` *resolves* the rule `impI` and the previous goal $A \longrightarrow (B \longrightarrow A)$ to $A \Longrightarrow B \longrightarrow A$. Put very suggestively, our current state says: if we can prove $B \longrightarrow A$ under the assumption A, we are done.

At the moment, you may find it difficult to understand the difference between $\Longrightarrow$ and $\longrightarrow$, since both somehow seem to stand for implication. However, $\longrightarrow$ is a symbol of propositional logic, which is our *object logic*, i.e., the language we are talking about. In contrast, $\Longrightarrow$ is a symbol of the *meta-logic*, i.e., Isabelle's built-in logic in which other object logics (PL, FOL, HOL, ...) are formalized.

Now apply `impI` a second time (by repeating the above line), you should end up in a state where Isabelle requires to prove A under the assumption "A B". This holds trivially, in Isabelle, this is made explicit by the so-called *assumption* (tactic) method. Type

assumption

apply (*assumption*)

and after executing this line, Isabelle should reply with `No Subgoals` which means, there is nothing to prove anymore. The effect of the assumption method is to remove the first (and in this case only) subgoal provided the conclusion to be proven (in this case A) is one of the assumptions. This completes our proof of $A \longrightarrow (B \longrightarrow A)$. Try to see that we built the proof tree starting from the bottom. We can now close the proof be entering

No Subgoals

done

done

Now, `My first theorem` is a proven theorem which can be used in the same way as any other rule, e.g. `impI`.

Summarizing, you should end up with the following theory file:

```
theory simple = FOL:
lemma "My first theorem": "A --> (B --> A)"
  apply (rule impI)
  apply (rule impI)
  apply (assumption)
  done
end
```

1.2.2 Forward-Style Reasoning in Isabelle

We will now prove $A \longrightarrow (B \longrightarrow A)$ using forward style; Forward proofs mirror more or less directly the structure of a proof tree. Considering the proof tree for $A \longrightarrow (B \longrightarrow A)$, we conclude that applying `impI` twice is a valid proof. Let's start with: with

lemmas

lemmas forward_proof = impI

Convince yourself by executing the `thm forward_proof` that Isabelle is now aware of this theorem. Now let's undo the last step by clicking on undo and change the above line to

lemmas forward_proof = impI [**OF** impI]

where `OF` takes a list of theorems and applies them to the premises of the first `impI`. Again, check the result of this step by executing `thm forward_proof`. As you see, we are nearly done, we only have choose the right assignment for the meta variables. This can be done by changing the above proof to

OF

of

lemmas forward_proof = impI [**OF** impI, **of** A B A]

This results in:

$$([\![A; B]\!] \Longrightarrow A) \Longrightarrow A \longrightarrow B \longrightarrow A$$

discharging

where the first part of this formula is an artefact from discharging the assumptions (it says essentially that A has been introduced as assumption during the proof and that possible "candidates for discharge" are A and B.)

A further version of this proof adds a particular clean-up that performs the discharging:

```
lemmas forward_proof = impl [OF impl, of A B A, simplified]
```

which completes the discharge:

$$([\![A; B]\!] \Longrightarrow True) \Longrightarrow \ldots$$

but, unfortunately, has also the undesired effect to distroy also our conclusion in this case:

$$([\![A; B]\!] \Longrightarrow True) \Longrightarrow True$$

1.2.3 Combining Forward- and Backward-Style Reasoning

Note that one can arbitrarily mix forward- and backward-style reasoning in Isabelle, e.g.

```
lemma third_proof:  "A --> (B --> A)"
  apply (rule forward_proof)
  apply (assumption)
  done
```

or even

```
lemma third_proof:  "A --> (B --> A)"
  apply (rule impl [OF impl, of A B A] )
  apply (assumption)
  done
```

are valid proofs for $A \longrightarrow (B \longrightarrow A)$.

1.3 Exercises

1.3.1 Exercise 1

Choose four of the following theorems and prove them

- using paper and pencil,

- in Isabelle using backward style, and
- in Isabelle using forward style.

Choose suitable names for the proven theorems, i.e. choose names based on the exercise number, like `ex1_1` for the first one.

1. $A \longrightarrow B \longrightarrow A$
2. $A \wedge B \longrightarrow B \wedge A$
3. $A \wedge B \longrightarrow A \vee B$
4. $A \vee B \longrightarrow B \vee A$
5. $A \wedge (B \wedge C) \longrightarrow A \wedge C$
6. $(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow (A \longrightarrow C)$
7. $(A \wedge B) \vee C \longrightarrow (A \vee C) \wedge (B \vee C)$

Answer to Exercise 1

1. Proving $A \longrightarrow (B \longrightarrow A)$
   - using pencil and paper:

$$\dfrac{\dfrac{[A]^1}{B \longrightarrow A}\ {\scriptstyle \rightarrow\text{-}I}}{A \longrightarrow (B \longrightarrow A)}\ {\scriptstyle \rightarrow\text{-}I^1}$$

   - in Isabelle using backward style:

```
lemma ex_1_1: "A ⟶(B ⟶A)"
  apply(rule impI)
  apply(rule impI)
  apply(assumption)
  done
```

   - in Isabelle using forward style:

```
lemmas ex_1_1F = impI [OF impI, of A B A]
```

2. Proving $A \wedge B \longrightarrow B \wedge A$

- using pencil and paper:

$$\dfrac{\dfrac{[A \wedge B]^2}{B}\,\scriptstyle{\wedge\text{-}ER} \qquad \dfrac{[A \wedge B]^2}{A}\,\scriptstyle{\wedge\text{-}EL}}{\dfrac{B \wedge A}{A \wedge B \longrightarrow B \wedge A}\,\scriptstyle{\longrightarrow\text{-}I^2}}\,\scriptstyle{\wedge\text{-}I}$$

- in Isabelle using backward style:

```
lemma ex_1_2: "A ∧B ⟶B ∧A"
  apply(rule impI)
  apply(rule conjI )
  apply(rule conjunct2)
  apply(assumption)
    (* ... eliminated first subgoal ...                    *)
  apply(rule conjunct1)
  apply(assumption)
  done
```

- in Isabelle using forward style:

```
lemmas ex_1_2F = impI [OF conjI, OF conjunct1 conjunct2,
                       of "(A ∧B)" B A B A]
```

3. Proving $A \wedge B \longrightarrow A \vee B$

- using pencil and paper:

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^3}{A}\,\scriptstyle{\wedge\text{-}EL}}{A \vee B}\,\scriptstyle{\vee\text{-}IL}}{A \wedge B \longrightarrow A \vee B}\,\scriptstyle{\longrightarrow\text{-}I^3}$$

- in Isabelle using backward style:

```
lemma ex_1_3: "A ∧B ⟶A ∨B"
```

- in Isabelle using forward style:

```
lemmas ex_1_3F = impI [OF disjI1[OF conjunct1],
                       of "(A ∧B)" A B B]
```

4. Proving $A \vee B \longrightarrow B \vee A$

- using pencil and paper:

$$\dfrac{[A \vee B]^4 \quad \dfrac{[A]^5}{B \vee A}\ \vee\text{-}IR \quad \dfrac{[B]^5}{B \vee A}\ \vee\text{-}IL}{\dfrac{B \vee A}{A \vee B \longrightarrow B \vee A}\ \rightarrow\text{-}I^4}\ \vee\text{-}E^5$$

- in Isabelle using backward style:

```
lemma ex_1_4: "A ∨B ⟶B ∨A"
  apply( rule impI)
  apply( rule disjE )
  apply(assumption)
  apply( rule disjI2 )
  apply(assumption)
  apply( rule disjI1 )
  apply(assumption)
  done
```

- in Isabelle using forward style:

```
lemmas ex_1_4F = impI [OF disjE [OF _ disjI2 disjI1 ],
```

5. Proving $A \wedge (B \wedge C) \longrightarrow A \wedge C$

- using pencil and paper:

$$\dfrac{\dfrac{\dfrac{[A \wedge (B \wedge C)]^6}{A}\ \wedge\text{-}EL \quad \dfrac{\dfrac{[A \wedge (B \wedge C)]^6}{B \wedge C}\ \wedge\text{-}ER}{C}\ \wedge\text{-}ER}{A \wedge C}\ \wedge\text{-}I}{A \wedge (B \wedge C) \longrightarrow A \wedge C}\ \rightarrow\text{-}I^6$$

- in Isabelle using backward style:

```
lemma ex_1_5: "A ∧(B ∧C) ⟶A ∧C"
  apply( rule impI)
  apply( rule conjI )
  apply( rule conjunct1)
  apply(assumption)
  apply( rule conjunct2)
  apply( rule conjunct2)
  apply(assumption)
  done
```

- in Isabelle using forward style:

```
lemmas ex_1_5F = impI [OF conjI
                       [OF conjunct1 conjunct2[OF conjunct2]],
                        of "A ∧(B ∧C)" A "B ∧C" A B C]
```

6. Proving $(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow A \longrightarrow C$

- using pencil and paper:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A \longrightarrow B \longrightarrow C]^7 \quad [A]^9}{B \longrightarrow C}\ {\to}\text{-}E \qquad \dfrac{[A \longrightarrow B]^8 \quad [A]^9}{B}\ {\to}\text{-}E}{C}\ {\to}\text{-}E}{A \longrightarrow C}\ {\to}\text{-}I^9}{(A \longrightarrow B) \longrightarrow A \longrightarrow C}\ {\to}\text{-}I^8}{(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow A \longrightarrow C}\ {\to}\text{-}I^7$$

- in Isabelle using backward style:

```
lemma ex_1_6: "(A ⟶B ⟶C) ⟶(A ⟶B)⟶ (A ⟶C)"
  apply(rule impI)
  apply(rule impI)
  apply(rule impI)
  apply(rule mp)
  apply(rule mp)
  apply(assumption)
  apply(assumption)
  apply(rule mp)
  apply(assumption)
  apply(assumption)
  done
```

- in Isabelle using forward style:

```
lemmas ex_1_6F = impI [OF impI[OF impI[OF mp[OF mp mp]]],
                  of "A ⟶B ⟶C" "A ⟶B" A A B C A]
```

7. Proving $A \wedge B \vee C \longrightarrow (A \vee C) \wedge (B \vee C)$

- using pencil and paper:

$$
\begin{prooftree}
\AxiomC{$[A \wedge B \vee C]^{10}$}
\AxiomC{$[A \wedge B]^{11}$}
\RightLabel{\scriptsize $\wedge$-EL}
\UnaryInfC{A}
\RightLabel{\scriptsize $\vee$-IL}
\UnaryInfC{$A \vee C$}
\AxiomC{$[C]^{11}$}
\RightLabel{\scriptsize $\vee$-IR}
\UnaryInfC{$A \vee C$}
\RightLabel{\scriptsize $\vee$-E^{11}}
\TrinaryInfC{$A \vee C$}
\AxiomC{$[A \wedge B \vee C]^{10}$}
\AxiomC{$[A \wedge B]^{12}$}
\RightLabel{\scriptsize $\wedge$-ER}
\UnaryInfC{B}
\RightLabel{\scriptsize $\vee$-IL}
\UnaryInfC{$B \vee C$}
\AxiomC{$[C]^{12}$}
\RightLabel{\scriptsize $\vee$-IR}
\UnaryInfC{$B \vee C$}
\RightLabel{\scriptsize $\vee$-E^{12}}
\TrinaryInfC{$B \vee C$}
\RightLabel{\scriptsize $\wedge$-I}
\BinaryInfC{$(A \vee C) \wedge (B \vee C)$}
\RightLabel{\scriptsize $\longrightarrow$-I^{10}}
\UnaryInfC{$A \wedge B \vee C \longrightarrow (A \vee C) \wedge (B \vee C)$}
\end{prooftree}
$$

- in Isabelle using backward style:

```
lemma ex_1_7:"((A ∧B) ∨C) ⟶ (A ∨C) ∧ (B ∨C)"
  apply(rule impI)
  apply(rule conjI )
    (* this splits into two subgoals. We first ignore the second one *)
    (* and solve : (A ∧ B) ∨ C ⟹ A ∨ C *)
  apply(rule disjE )
  apply(assumption)
  apply(rule disjI1 )
  apply(rule conjunct1)
  apply(assumption)
  apply(rule disjI2 )
  apply(assumption)
    (* Now, we finish the other one subgoal in a similar way *)
  apply(rule disjE )
  apply(assumption)
  apply(rule disjI1 )
  apply(rule conjunct2)
  apply(assumption)
  apply(rule disjI2 )
  apply(assumption)
  done
```

- in Isabelle using forward style:

```
lemmas ex_1_7_aux1 = disjE [of "A ∧B" C "A ∨C",
                            OF _ disjI1 [OF conjunct1] disjI2 ,
                            of B, simplified ]
thm ex_1_7_aux1


lemmas ex_1_7_aux2 = disjE [OF _ disjI1 [OF conjunct2] disjI2 ,
```

```
                       of "(A ∧B)" C A B C, simplified ]
thm "ex_1_7_aux2"


lemmas ex_1_7F = impl [OF conjI, OF ex_1_7_aux1 ex_1_7_aux2,
                    of "A ∧B ∨C", simplified ]
thm "ex_1_7F"
```

1.3.2 Exercise 2

not_def
fold
unfold

Look up the definition of ¬ (hint: it's called `not_def`) and execute step-by-step the following proof script:

```
lemma "P ∧¬P ⟶R"
  apply (unfold not_def)
  apply (fold not_def)
  oops
```

oops

Explain the proof script in detail. Here, `oops` just abandons our proof.

Now proof the following lemmas using Isabelle (either forward or backward style):

1. $P \wedge \neg P \longrightarrow R$
2. $(A \vee B) \wedge \neg A \longrightarrow B$
3. $(A \vee \neg A) \longrightarrow ((A \longrightarrow B) \longrightarrow A) \longrightarrow A$

Keep the last theorem in mind, it will be useful later.

Answer to Exercise 2

1. Proving $P \wedge (P \longrightarrow \bot) \longrightarrow R$

   - using pencil and paper:

$$\dfrac{\dfrac{\dfrac{\dfrac{[P \wedge (P \longrightarrow \bot)]^1}{P \longrightarrow \bot}\,{\scriptstyle \wedge\text{-}ER} \quad \dfrac{[P \wedge (P \longrightarrow \bot)]^1}{P}\,{\scriptstyle \wedge\text{-}EL}}{\bot}\,{\scriptstyle \longrightarrow\text{-}E}}{R}\,{\scriptstyle \bot\text{-}E}}{P \wedge (P \longrightarrow \bot) \longrightarrow R}\,{\scriptstyle \longrightarrow\text{-}I^1}$$

- in Isabelle:

```
lemma "ex_2_1": "P ∧ ¬P ⟶ R"
  apply(unfold not_def)
  apply(rule impI)
  apply(rule FalseE)
  apply(rule mp)
  apply(rule conjunct2)
  apply(assumption)
  apply(rule conjunct1)
  apply(assumption)
  done
```

2. Proving $(A \vee B) \wedge \neg A \longrightarrow B$

- using pencil and paper:

$$\dfrac{\dfrac{\dfrac{[(A\vee B)\wedge\neg A]^2}{A\vee B}{\scriptstyle \wedge\text{-}EL} \quad \dfrac{\dfrac{\dfrac{[(A\vee B)\wedge\neg A]^2}{\neg A}{\scriptstyle \wedge\text{-}ER} \quad [A]^3}{\bot}{\scriptstyle \to\text{-}E}}{B}{\scriptstyle \bot\text{-}E} \quad [B]^3}{B}{\scriptstyle \vee\text{-}E^3}}{(A\vee B)\wedge\neg A\longrightarrow B}{\scriptstyle \to\text{-}I^2}$$

- in Isabelle:

```
lemma ex_2_2: "(A ∨B) ∧¬A ⟶B"
  apply(unfold not_def)
  apply(rule impI)
  apply(rule disjE )
  apply(rule conjunct1)
  apply(assumption)
  apply(rule FalseE)
  apply(rule mp)
  apply(rule conjunct2)
  apply(assumption)
  apply(assumption)
  apply(assumption)
  done
```

3. Proving $(A \vee \neg A) \longrightarrow ((A \longrightarrow B) \longrightarrow A) \longrightarrow A$

- using pencil and paper:

$$\dfrac{\dfrac{\dfrac{[(A \lor \neg A)]^4 \quad [A]^6 \quad \dfrac{[((A \longrightarrow B) \longrightarrow A)]^5 \quad \dfrac{\dfrac{\dfrac{[\neg A]^6 \quad [A]^7}{\bot}\,{\scriptstyle \longrightarrow\text{-}E}}{B}\,{\scriptstyle \bot\text{-}E}}{(A \longrightarrow B)}\,{\scriptstyle \longrightarrow\text{-}I^7}}{A}\,{\scriptstyle \longrightarrow\text{-}E}}{A}\,{\scriptstyle \lor\text{-}E^6}}{((A \longrightarrow B) \longrightarrow A) \longrightarrow A}\,{\scriptstyle \longrightarrow\text{-}I^5}}{(A \lor \neg A) \longrightarrow ((A \longrightarrow B) \longrightarrow A) \longrightarrow A}\,{\scriptstyle \longrightarrow\text{-}I^4}$$

- in Isabelle:

```
lemma ex_2_3:
  "(A ∨ ¬A) ⟶(((A⟶B)⟶A)⟶A)"
  apply(unfold not_def)
  apply(rule impI)
  apply(rule impI)
  apply(rule disjE)
  apply(assumption)
  apply(assumption)
  apply(rule mp)
  apply(assumption)
  apply(rule impI)
  apply(rule FalseE)
  apply(rule mp)
  apply(assumption)
  apply(assumption)
  done
```

1.3.3 Exercise 3

So far we only used rules of the intuitionistic propositional logic. We will now add one further rule

$$\dfrac{\begin{matrix}[\neg A]\\ \vdots \\ A\end{matrix}}{A}\,{\scriptstyle classical}$$

classical to obtain *classical* propositional logic. The characteristic of classical logic is that the principle of the excluded middle holds: $P \vee \neg P$.

We show that *classical* is equivalent to the principle of the excluded middle. excluded middle As above, do the proofs both using paper and pencil and in Isabelle.

1. $(\neg Q \longrightarrow P) \longrightarrow P \vee Q$ (hint: the main part of this proof is a proof of $P \vee Q$ using, among others, the assumption $\neg(P \vee Q)$, followed by an application of *classical*).

2. Using the previous theorem, prove $P \vee \neg P$ (hint: first prove $\neg P \vee P$).

3. Prove $P \vee \neg P \longrightarrow ((\neg P \longrightarrow P) \longrightarrow P)$ intuitionistically.

Answer to Exercise 3

1. Proving $(\neg Q \longrightarrow P) \longrightarrow P \vee Q$

   - using pencil and paper:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\neg Q \longrightarrow P]^1 \qquad \dfrac{\dfrac{[\neg(P \vee Q)]^2 \qquad \dfrac{[Q]^3}{P \vee Q}\,{\scriptstyle \vee\text{-}IR}}{\bot}\,{\scriptstyle \longrightarrow\text{-}E}}{\neg Q}\,{\scriptstyle \longrightarrow\text{-}I^3}}{P}\,{\scriptstyle \longrightarrow\text{-}E}}{P \vee Q}\,{\scriptstyle \vee\text{-}IL}}{P \vee Q}\,{\scriptstyle classical^2}}{(\neg Q \longrightarrow P) \longrightarrow P \vee Q}\,{\scriptstyle \longrightarrow\text{-}I^1}$$

   - in Isabelle:

```
lemma ex_3_1: "(¬Q ⟶P) ⟶(P ∨Q)"
  apply(rule impI)
  apply(rule classical )
  apply(rule disjI1 )
  apply(rule mp)
  apply(unfold not_def)
  apply(assumption)
  apply(rule impI)
  apply(rule mp)
  apply(assumption)
  apply(rule disjI2 )
```

```
  apply(assumption)
  done
```

2. Proving $\neg P \vee P$

   - using pencil and paper:

$$\dfrac{\dfrac{\boxed{\text{Ex. 1.4}}}{(\neg P \vee P) \longrightarrow (P \vee \neg P)} \qquad \dfrac{\dfrac{\boxed{\text{Ex. 3.1}}}{(\neg P \longrightarrow \neg P) \longrightarrow \neg P \vee P} \qquad \dfrac{[\neg P]^4}{\neg P \longrightarrow \neg P}\,{\scriptstyle \longrightarrow\text{-}I^4}}{\neg P \vee P}\,{\scriptstyle \longrightarrow\text{-}E}}{P \vee \neg P}\,{\scriptstyle \longrightarrow\text{-}E}$$

   - in Isabelle:

```
lemma ex_3_2: "P ∨¬P"
  apply( rule mp)
  apply( rule ex_1_4)
  apply( rule mp)
  apply( rule ex_3_1)
  apply( rule impI)
  apply(assumption)
  done
```

3. Instance of Exercise 2.3 (noting the equivalence of $\neg\phi$ and $\phi \longrightarrow \bot$). Using Isabelle we have the problem is that to apply "ex2_3" we would need to rewrite the first occurrence of P only. The solution is to prove the following auxiliary statement:

```
lemma ex_3_3_aux: "(A ∨(A ⟶False)) ⟶ ((A ⟶B) ⟶ A) ⟶ A"
  apply( fold not_def)
  apply( rule ex_2_3)
  done

lemma ex_3_3: "P ∨¬P ⟶((¬P ⟶P) ⟶P)"
  apply(unfold not_def)
  apply( rule ex_3_3_aux)
  done
```

1.3.4 Exercise 4

Prove the following classical theorem called *Peirce's law*, both using paper and pencil and in Isabelle:

$$((A \longrightarrow B) \longrightarrow A) \longrightarrow A$$

Hing: Use the proof of $P \vee \neg P$ from Ex. 3.

Answer to Exercise 4

Proving Peirce's law

- using pencil and paper:

$$\dfrac{\dfrac{\boxed{\text{Ex. 2.3}}}{(A \vee \neg A) \longrightarrow ((A \longrightarrow B) \longrightarrow A) \longrightarrow A} \qquad \dfrac{\boxed{\text{Ex. 3.2}}}{(A \vee \neg A)}}{((A \longrightarrow B) \longrightarrow A) \longrightarrow A} \;{\scriptstyle \longrightarrow\text{-}E}$$

- in Isabelle:

```
lemma "Peirce Law": "((A ⟶B) ⟶A) ⟶A"
  apply(rule mp)
  apply(rule ex_2_3)
  apply(rule ex_3_2)
  done
```

2 First-Order Logic

In this lecture you will deepen your knowledge about *first-order logic* (FOL). Theorem proving in FOL involves the issue of binding and substitution, which we treat at a fairly pragmatic level for the moment. (The issue will be revisited in the subsequent exercises on meta-theory and λ-calculus). We will learn to manage premises in backward proofs and tactic methods that manipulate assumptions in backward proofs.

2.1 More on Isabelle

2.1.1 Isabelle System Architecture

For using Isabelle it is sometimes helpful if one has a broad overview of Isabelle's system architecture (see Fig. 2.1). Isabelle is generic theorem prover providing a simple meta logic; it is implemented the functional language "Standard ML" (SML). On top of the Isabelle core, a variety of Isabelle instances are built, e.g. Isabelle/FOL (for first-order logic), Isabelle/HOL (for higher-order logic), or Isabelle/ZF (for Zermelo-Fränkel set theory). Isabelle instances can be programmed directly via programs written in SML or via the ISAR-proof language, which also provides powerful documentation facilities. On top of this, different user interfaces are provided. In this lecture, we use the most modern interface, called "Proof General", which itself builds upon the (X)Emacs editor family. SML Proof General

2.1.2 Assumptions in Backward Proof

In backward proof, Isabelle allows two notions for introducing assumptions into a proof context. For simple cases we can use assumptions

lemma *name*: "$[\![\, a_1; \ldots; a_n]\!] \Longrightarrow C$"

The latter format introduces assumptions as named objects that can be referenced identically to rules:

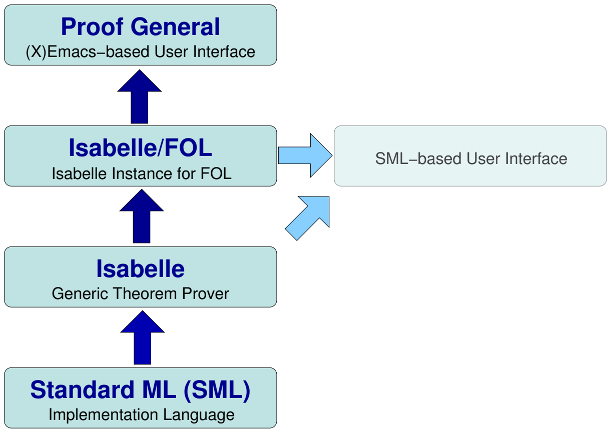

Figure 2.1: The System Architecture of Isabelle

lemma *name*:	**lemma** *name*:
assumes $name_1$: "a_1"	**assumes** $name_1$: "a_1"
$\vdots$	$\vdots$
assumes $name_n$: "a_n"	and $name_n$: "a_n"
shows "C"	**shows** "C"

assumes and *insert* One can use the **assumes** or and to enumerate several assumptions. Using this style, the assumptions are not automatically added to assumption list of the goals. If needed, you can insert them with the command *insert*.

Assumptions, derived rules, rules, axioms, theorems are all the same in Isabelle and can be combined in arbitrary ways in forward and backward proof! Remark: Internally, all these objects are represented as a particular abstract data type `thm`. The Isabelle kernel is a collection of SML-modules that implement this data type (provers of this system architecture are often referred as LCS-style-provers).

thm

LCS-style

2.1.3 New FOL Rules

The distinctive feature of FOL compared to PL are the quantifiers $\forall$ and $\exists$. Note that quantifiers have low priority, e.g., we have to write $(\forall x.p(x)) \longrightarrow (\exists x.p(x))$.

Recall the introduction and elimination quantifier rules from the lecture:

$$\frac{P(x)}{\forall x.\, P(x)}\ \forall\text{-}I^{1.} \qquad \frac{\forall x.\, P(x)}{P(t)}\ \forall\text{-}E \qquad \frac{P(t)}{\exists x.\, P(x)}\ \exists\text{-}I \qquad \frac{\exists x.\, P(x) \quad \begin{matrix}[P(x)]\\ \vdots \\ R\end{matrix}}{R}\ \exists\text{-}E^{2.}$$

quantifier rules

where the side conditions are:

1. x is not free in any assumption on which $P(x)$ depends.
2. x is not free in B or any assumption of the sub-derivation of B other than $A(x)$.

In Isabelle/FOL, these rules are represented (including the side conditions) as follows:

spec
allI
exI
exE

$$(\bigwedge x.P(x)) \Longrightarrow (\forall x.P(x)) \quad \texttt{allI} \qquad\qquad (\bigwedge x.P(x)) \Longrightarrow P(x) \quad \texttt{spec}$$

$$P(x) \Longrightarrow (\exists x.P(x)) \quad \texttt{exI} \qquad\qquad [\![\exists x.P(x); \bigwedge x.P(x) \Longrightarrow R]\!] \Longrightarrow R \quad \texttt{exE}$$

Where $\bigwedge$ is the meta-level universal quantification. If a goal is preceded by the meta-quantor $\bigwedge x. \ldots$, this means that Isabelle must be able to prove the subgoal in a way which is independent from x, i.e., without instantiating x. Another view on meta-level quantification is that they introduce "fresh free variables" on the fly (in fact, variables bound by outermost meta-level quantifiers were treated as free variables within substitutions).

meta-level universal quantifier

$\bigwedge$

Whenever an application of a rule leads to the introduction of meta variables in a goal preceded by $\bigwedge$, these introduced meta variables will be made dependent on x. You may also say that those meta-variables will be *Skolem* functions of x. When experimenting with rule applications introducing $\bigwedge$'s, you will notice that the order of these introductions is crucial.

2.1.4 Substitutions in Backward Proof

As mentioned in the lecture, Isabelle uses meta-variables $?X$,$?Y$,.... These meta-variables are *logically* treated as *free variables*, but may be instantiated either interactively or automatically by Isabelle itself.

Sometimes the automatic instantiation is not appropriate for a proof; then the user must provide it interactively. In forward proof, this can be done by the **of** command you have already got to know.

In backward proof, variants of proof commands were provided. Instead of

apply(*rule* $name$)

we might give several substitution during rule application : *rule_tac*

apply(*rule_tac* **of** x_1= "$term_1$" and ...and x_n= "$term_n$" **in** $rule$)

Where *rule_tac* may contain syntactic elements and free variables of the proof context. Note that

apply(*rule_tac* **of** x = "$term$" **in** $rule$)

is *not* the same as

apply(*rule* [**of** ..."$term$" ...])

Can you figure out why?

2.1.5 Manipulating Assumptions in Backward Proof

So far, we never changed the assumptions a_i of a goal $[\![a_1; \ldots; a_n]\!] \Longrightarrow C$. The command *rule* instantiates its argument rule such that its conclusion becomes equal to the conclusion C of the goal.

A collection of Isabelle tactic methods follows a different strategy:

erule

1. *erule* $rule$ constructs an instantiation such that the first assumption b_1 of $rule$ becomes equal to an a_i, and that the conclusion of $rule$ becomes equal to an C. a_i is erased from the assumptions.

drule

2. *drule* $rule$ constructs an instantiation such that the first assumption b_1 of $rule$ becomes equal to an a_i, and that the conclusion of $rule$ becomes a new assumption. a_i is erased from the assumptions.

frule

3. *drule* $rule$ works like *drule* $rule$ but does not erase a_i.

insert Moreover, with the command *insert*, an arbitrary theorem or assumption can be added to the assumption list.

Note that for some of these tactic methods are variants with explicit substitutions available: erule_tac, drule_tac, and frule_tac.

erule_tac
drule_tac
frule_tac

2.2 Exercises

2.2.1 Exercise 5

Derive the following rules in Isabelle:

$$\frac{P \wedge Q \qquad \begin{matrix}[P, Q] \\ \vdots \\ R\end{matrix}}{R}\,\texttt{conjE} \qquad \frac{P \longrightarrow Q \quad P \quad \begin{matrix}[Q] \\ \vdots \\ R\end{matrix}}{R}\,\texttt{impE} \qquad \frac{\begin{matrix}[P] \\ \vdots \\ \bot\end{matrix}}{\neg P}\,\texttt{notI} \qquad \frac{\neg P \quad P}{R}\,\texttt{notE}$$

conjE, impE
notI, notE

Answer to Exercise 5

```
lemma conjE: assumes major: " P ∧Q" assumes prem: " ⟦P; Q ⟧⟹R"
             shows  "R"
  apply(rule prem)
  apply(rule conjunct1[OF major])
  apply(rule conjunct2[OF major])
  done

lemma impE: assumes major: "P ⟶Q"
            assumes prem1: "P"
            assumes prem2: "Q ⟹R"
            shows "R"
  apply(rule prem2)
  apply(rule mp [OF major])
  apply(rule prem1)
  done

lemma notI: assumes prem: "(P ⟹False)" shows "¬P"
  apply(unfold not_def)
  apply(rule impI)
  apply(assumption)
  done

lemma notE: "⟦ ¬P;  P ⟧ ⟹  R"
  apply(unfold not_def)
  apply(erule FalseE [OF mp])
  apply(assumption)
  done
```

2.2.2 Exercise 6

Prove the following theorems using `erule` and `disjE` and `conjE` wherever possible.

1. $(A \wedge B) \wedge C \longrightarrow A \wedge B \wedge C$
2. $(A \wedge B) \wedge (C \wedge D) \longrightarrow (B \wedge C) \wedge (D \wedge A)$
3. $(A \vee B) \vee (C \vee D) \longrightarrow (B \vee C) \vee (D \vee A)$

Compare the first two proofs with the proofs without `erule` in Ex. 1.

Answer to Exercise 6

1. Proving $(A \wedge B) \wedge C \longrightarrow A \wedge B \wedge C$:

```
lemma ex6_1: "(A ∧B) ∧C ⟶A ∧B ∧C"
  apply(rule impI)
  apply(erule conjE)
  apply(erule conjE)
  apply(rule conjI)
  apply(assumption)
  apply(rule conjI)
  apply(assumption)
  apply(assumption)
  done
```

```
lemma ex6_1_wo_erule: "(A ∧B) ∧C ⟶A ∧B ∧C"
  apply(rule impI)
  apply(rule conjI)
  apply(rule conjunct1)
  apply(rule conjunct1)
  apply(assumption)
  apply(rule conjI)
  apply(rule conjunct2)
  apply(rule conjunct1)
  apply(assumption)
  apply(rule conjunct2)
  apply(assumption)
  done
```

2. Proving $(A \wedge B) \wedge (C \wedge D) \longrightarrow (B \wedge C) \wedge (D \wedge A)$:

```
lemma ex6_2: "(A ∧B) ∧(C∧D) ⟶(B ∧C) ∧(D ∧A)"
  apply(rule impI)
  apply(erule conjE)
  apply(erule conjE)
  apply(erule conjE)
  apply(rule conjI)
  apply(rule conjI)
  apply(assumption)
  apply(assumption)
  apply(rule conjI)
  apply(assumption)
  apply(assumption)
  done

lemma ex6_2_wo_erule: "(A ∧B) ∧(C∧D) ⟶(B ∧C) ∧(D ∧A)"
  apply(rule impI)
  apply(rule conjI)
  apply(rule conjI)
  apply(rule conjunct2)
  apply(rule conjunct1)
  apply(assumption)
  apply(rule conjunct1)
  apply(rule conjunct2)
  apply(assumption)
  apply(rule conjI)
  apply(rule conjunct2)
  apply(rule conjunct2)
  apply(assumption)
  apply(rule conjunct1)
  apply(rule conjunct1)
  apply(assumption)
  done
```

3. Proving $(A \vee B) \vee (C \vee D) \longrightarrow (B \vee C) \vee (D \vee A)$:

```
lemma ex6_3: "(A ∨B) ∨(C∨D) ⟶(B ∨C) ∨(D ∨A)"
  apply(rule impI)
  apply(erule disjE)
  apply(erule disjE)
  apply(rule disjI2)
```

```
apply(erule disjI2 )
apply(rule disjI1 )
apply(rule disjI1 )
apply(assumption)
apply(erule disjE)
apply(rule disjI1 )
apply(erule disjI2 )
apply(rule disjI2 )
apply(erule disjI1 )
done
```

2.2.3 Exercise 7

Derive the rule

$$\frac{\begin{matrix}[A] & [B] \\ \vdots & \vdots \\ B & A\end{matrix}}{A \longleftrightarrow B}$$

in Isabelle. Recall that $\longleftrightarrow$ is defined by:

$$P \longleftrightarrow Q \equiv (P \longrightarrow Q) \land (Q \longrightarrow P) \qquad \texttt{iff\_def}$$

Use *erule* and *drule* wherever you can.

Answer to Exercise 7

```
lemma iffI: assumes p1: "A ⟹B"
            assumes p2: "B ⟹A"
            shows "A ⟷B"
  apply(unfold iff_def )
  apply(rule conjI)
  apply(rule impI)
  apply(erule p1)
  apply(rule impI)
  apply(erule p2)
  done
```

2.2.4 Exercise 8

Prove the following theorems of first-order logic in Isabelle:

1. $(\forall x.p(x)) \longrightarrow \exists x.p(x)$
2. $((\forall x.p(x)) \vee (\forall x.q(x))) \longrightarrow (\forall x.(p(x) \vee q(x)))$
3. $((\forall x.p(x)) \wedge (\forall x.q(x))) \longleftrightarrow (\forall x.(p(x) \wedge q(x)))$
4. $(\exists x.\forall y.p(x, y)) \longrightarrow (\forall y.\exists x.p(x, y))$
5. $(\exists x.p(f(x))) \longrightarrow (\exists x.p(x))$

What about: $(\forall x.(p(x) \vee q(x))) \longrightarrow ((\forall x.p(x)) \vee (\forall x.q(x)))$? Can you prove it?

Answer to Exercise 8

1. Proving $(\forall x.p(x)) \longrightarrow \exists x.p(x)$

```
lemma ex8_1: "(∀x. p(x)) ⟶(∃x. p(x))"
  apply(rule impI)
  apply(rule exI)
  apply(erule spec)
  done
```

2. Proving $((\forall x.p(x)) \vee (\forall x.q(x))) \longrightarrow (\forall x.(p(x) \vee q(x)))$

```
lemma ex8_2: "(∀x. p(x)) ∨(∀x. q(x))  ⟶(∀x. p(x)  ∨q(x))"
  apply(rule impI)
  apply(rule allI )
  apply(erule disjE)
  apply(rule disjI1 )
  apply(rule spec)
  apply(assumption)
  apply(rule disjI2 )
  apply(rule spec)
  apply(assumption)
  done
```

3. Proving $((\forall x.p(x)) \wedge (\forall x.q(x))) \longleftrightarrow (\forall x.(p(x) \wedge q(x)))$

```
lemma ex8_3: "(∀x. p(x)) ∧(∀x. q(x))  ⟷(∀x. p(x)  ∧q(x))"
  apply(rule iffI )
  apply(rule allI )
  apply(rule conjI)
  apply(erule conjE)
  apply(erule spec)
```

```
  apply(erule conjE)
  apply(erule spec)
   (* Other direction of iff *)
  apply(rule conjI)
  apply(rule allI)
  apply(erule allE)
  apply(erule conjE)
  apply(assumption)
  apply(rule allI)
  apply(erule allE)
  apply(erule conjE)
  apply(assumption)
  done
```

4. Proving $(\exists x.\forall y.p(x,y)) \longrightarrow (\forall y.\exists x.p(x,y))$

```
lemma ex8_4: "(∃x. ∀y. p(x,y)) ⟶ (∀y. ∃x. p(x,y))"
  apply(rule impI)
  apply(erule exE)
  apply(rule allI)
  apply(rule exI)
  apply(rule spec)
  apply(assumption)
  done
```

5. Proving $(\exists x.p(f(x))) \longrightarrow (\exists x.p(x))$

```
lemma ex8_5: "(∃x. p(f(x))) ⟶ (∃x. p(x))"
  apply(rule impI)
  apply(erule exE)
  apply(erule exI)
  done
```

2.2.5 Exercise 9

Prove

$$\frac{}{(\forall x.A \longrightarrow B(x)) \longleftrightarrow (A \longrightarrow \forall x.B(x))}\ \texttt{all\_distr}$$

in Isabelle. Reuse Exercise 7.

In lecture '1.5 FOL: Natural Deduction" it was said that in the above theorem it is crucial that "A does not contain x freely". How does Isabelle take this into account? Try to prove: $p(x) \longrightarrow \forall x.p(x)$

Answer to Exercise 9

```
lemma all_distr: "(∀x. A ⟶ B(x)) ⟷ (A ⟶ (∀x. B(x)))"
  apply(rule iffI)
  apply(rule impI)
  apply(rule allI)
  apply(rule mp)
  apply(erule spec)
  apply(assumption)
  apply(rule allI)
  apply(rule impI)
  apply(rule spec)
  apply(erule mp)
  apply(assumption)
  done
```

2.2.6 Exercise 10

Prove the following theorem of first-order logic in Isabelle:

$$s\,(s\,(s\,(s\,(zero)))) = four \wedge p(zero) \wedge (\forall x.p(x) \longrightarrow p(s(s(x)))) \longrightarrow p(four)$$

Answer to Exercise 10

```
lemma ex8: "s(s(s(s(zero)))) = four ∧ p(zero) ∧ (∀x. p(x)
                                    ⟶ p(s(s(x)))) ⟶ p(four)"
  apply(rule impI)
  apply(erule conjE)
  apply(erule conjE)
  apply(erule subst)
  apply(rule mp)
  apply(rule_tac x = "s (s (zero))" in spec)
  apply(assumption)
  apply(rule mp)
  apply(rule_tac x = "zero" in spec)
  apply(assumption)
  apply(assumption)
  done
```

3 Naïve Set Theory

In this exercise, we will study a particular version of a set theory, called *Naïve Set Theory*, originally proposed by Frege and still implicitly used by many mathematicians. We introduce some of its axioms, notation and, properties. At the end, we use a Paradox due to Russel in order to show that Naive Set Theory is inconsistent.

3.1 More on Isabelle

3.1.1 Backward Proof Control Structures

Revising our first proof scripts, it becomes clear that proof-scripts contain considerable repetition. Thus, more automation can be achieved by introducing control structures in the ISAR-language. These are:

control structures
ISAR
sequential composition (,)
alternative (|)
option (?)
repetition (+)

1. M,M' *sequential composition*: try tactic M; if it succeeds try tactic M'.
2. M|M *alternative*: try tactic M; if it fails try tactic M'.
3. M? *option*: try tactic M; if it fails report success.
4. M+ *repetition*: try tactic M and repeat as long as no failure occurs.

For example, instead of:

```
apply(rule X)
apply(erule Y)
```

we may write:

```
apply(rule X, rule Y)
```

Further, instead of:

```
apply(drule mp)
apply(assumption)
apply(assumption)
apply(erule disjE)
apply(drule mp)
apply(drule conjunct1)
```

Usual notation	Isabelle
$\{1,2,3\}$	{1,2,3}
$\in$	:
$\notin$	~:
$\{y \mid P(y)\}$	{y. P(y)}
$A \cap B$	A Int B

Usual notation	Isabelle
$A \cup B$	A Un B
$A \subseteq B$	A <= B
$A \setminus B$	A Minus B
$\mathcal{P}(A)$	Pow(A)

Table 3.1: Notations used in `naive_set.thy`

we may write:

apply(*drule* mp,(*assumption*|*erule* disjE | *drule* conjunct1))+

3.1.2 Proof-State Massage

defer n
prefer n

The standard **apply**-command usually effects only the first subgoal. Thus, it may be desirable to rotate the list of subgoals in a proof state. The **defer** n or **prefer** n commands move a subgoal to the last or the first position.

rotate_tac n

For the choice of unifiers, the order of assumptions in a subgoal may be relevant. *rotate_tac* n rotates the assumptions of the first subgoal by n positions: from right to left if n is positive, and from left to right otherwise. The default value is one.

3.2 Naïve Set Theory

`naive_set.thy`

(Naïve) set theory has been formalized in Isabelle in the theory `naive_set.thy` (see allso Appendix 5.3). Tab. 3.1 shows some hints on the syntax used in this theory.

In lecture "Naïve Set Theory", we have seen four elementary rules of set theory

$$\frac{P(t)}{t \in \{x \mid P(x)\}} \in\text{-}I \qquad \frac{t \in \{x \mid P(x)\}}{P(t)} \in\text{-}E$$

$$\frac{\forall x. x \in A \leftrightarrow x \in B}{A = B} =\text{-I} \qquad \frac{A = B}{\forall x. x \in A \leftrightarrow x \in B} =\text{-E}$$

`ext`
`Collect`
`naive_set.thy`

In the provided Isabelle theory, instead of those inference rules, we have two axioms `ext` and `Collect` which have been encoded and derived in Isabelle.

For this exercise, download the `http://www.infsec.ethz.ch/education/`

`permanent/csmr/material/naive_set.thy` file and install it in the same directory where you store your work. Use the following template as starting point for your own Isabelle file:

```
theory exercise2 = naive_set.thy:

end
```

3.3 Exercises

3.3.1 Exercise 11

Prove in Isabelle that the subset relation is a partial order, i.e., it is reflexive, transitive and antisymmetric.

Answer to Exercise 11

1. Reflexivity:

```
lemma ex11_1: "A <= A"
  apply(unfold subset_def)
  apply(rule allI )
  apply(rule impI)
  apply(assumption)
  done
```

2. Transitivity:

```
lemma ex11_2: "(A <= B) ∧(B <= C) ⟶(A <= C)"
  apply(unfold subset_def)
  apply(rule impI)
  apply(rule allI )
  apply(rule impI)
  apply(erule conjE)
  apply(rule mp, erule spec)+
  apply(assumption)
  done
```

3. Anti-symmetry:

```
lemma ex11_3: "(A <= B ∧B <= A) ⟶A = B"
  apply(rule impI)
  apply(rule equalsI )
```

```
        apply(rule allI )
        apply(unfold iff_def )
        apply(rule conjI )
        apply( rule_tac x = "x" in spec)
        apply(fold subset_def )
        apply(erule conjunct1)
        apply( rule_tac x = "x" in spec)
        apply(fold subset_def )
        apply(erule conjunct2)
        done
```

3.3.2 Exercise 12

Prove $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ in Isabelle.

Answer to Exercise 12

```
lemma ex12: "(A ∩(B ∪C)) = ((A ∩B) ∪ (A ∩C))"
  apply(rule equalsI )
  apply(rule allI )
  apply(rule iffI )
  apply(unfold Int_def Un_def)
  apply(rule inI )
  apply(erule inE)
  apply(erule conjE)
  apply(erule inE)
  apply(erule disjE )
  apply(rule disjI1 )
  apply(rule inI )
  apply(rule conjI )
  apply(assumption)+
  apply(rule disjI2 )
  apply(rule inI )
  apply(rule conjI )
  apply(assumption)+
  apply(rule inI )
  apply(rule conjI )
  apply(erule inE)
  apply(erule disjE )
```

```
  apply(erule inE, erule conjunct1)+
  apply(erule inE)
  apply(rule inI)
  apply(erule disjE)
  apply(erule inE)
  apply(rule disjI1)
  apply(erule conjunct2)
  apply(rule disjI2)
  apply(erule inE)
  apply(erule conjunct2)
  done
```

3.3.3 Exercise 13

Prove $\mathcal{P}(A) \subseteq \mathcal{P}(B) \leftrightarrow A \subseteq B$ in Isabelle

Answer to Exercise 13

```
lemma ex13: "Pow(A) <= Pow(B) ⟷A <= B"
  apply(rule iffI)
  apply(unfold Pow_def subset_def)
  apply(rule allI)
  apply(rule impI)
  apply(erule allE)
  apply(erule impE)
  apply(rule inI)
  apply(rule allI)
  apply(rule impI)
  apply(assumption)
  apply(erule inE)
  apply(erule allE)
  apply(erule mp)
  apply(assumption)
    (*second half*)
  apply(rule allI)
  apply(rule impI)
  apply(rule inI)
  apply(rule allI)
  apply(rule impI)
  apply(erule inE)
  apply(erule allE, erule impE)+
```

```
  apply(assumption)+
  done
```

3.3.4 Exercise 14

Show that `NSet` is inconsistent, i.e., that ⊥ can be derived in it. You should start like this:

```
lemma ex14: "False"
  apply( rule_tac P = "{A. A ∉A} ∈{A. A ∉A}" in notE)
```

The rule *classical_dual* will be useful.

Answer to Exercise 14

```
lemma aux_lemma: assumes prem: "(A ⟹(A⟶B))" shows "(A⟶ B)"
  apply( rule impI)
  apply( rule mp)
  apply( erule prem)
  apply(assumption)
  done

lemma classical_dual : assumes prem: "(A ⟹(¬A))" shows "¬A"
  apply(unfold not_def)
  apply( rule aux_lemma)
  apply( fold not_def)
  apply( erule prem)
  done


lemma ex14: "False"
  apply( rule_tac P = "{A. A ∉A} ∈{A. A ∉A}" in notE)
  apply( rule  classical_dual )
  apply( erule inE) ;
  apply(assumption)
  apply( rule inI )
  apply( rule  classical_dual )
  apply( erule inE) ;
  apply(assumption)
  done
```

And now, proving gets much more easier, e.g., re-doing the proof of Ex. 12:

lemma "ex12'''": "(A ∩(B ∪C)) = ((A ∩B) ∪ (A ∩ C))"
  apply (*insert* ex14, *erule* FalseE)
  done

3.4 Encoding Naïve Set Theory in Isabelle

```
theory naive_set = FOL:

nonterminals i

types set = i

arities i :: "term"

consts
  "0"     :: i                    ("0")
  "1"     :: i                    ("1")
  "{}"    :: set                  ("{}")
  insert  :: "[i, set] ⇒ set"
  "op :"  :: "[i, set] ⇒ o"       ("(_/ : _)" [50, 51] 50)
  "<="    :: "[set, set] ⇒ o"     (infixl 50)
  Collect :: "[i ⇒ o] ⇒ set"
  INTER :: "[set, i ⇒ set] ⇒ set"
  UNION :: "[set, i ⇒ set] ⇒ set"
  Minus   :: "[set, set] ⇒ set" (infixl 65)
  Int     :: "[set, set] ⇒ set" (infixl 70)
  Un      :: "[set, set] ⇒ set" (infixl 65)
  Compl   :: "set ⇒ set"
  Pow     :: "set ⇒ set"
  UNIV    :: set
  Ball    :: "[set, i ⇒ o] ⇒ o"
  Bex     :: "[set, i ⇒ o] ⇒ o"

syntax
  "op :"      :: "[i,set] ⇒ o"                 ("op :")
  "op ~:"     :: "[i,set] ⇒ o"                 ("(_/ ~: _)" [50, 51] 50)
  "op ~:"     :: "[i,set] ⇒ o"                 ("op ~:")
  "@Collect" :: "[pttrn, o] ⇒ set"             ("(1{_./ _})")
  "@Finset"  :: "args => set"                  ("{(_)}")
  "@INTER" :: "[pttrn, set, set] ⇒ set"        ("(3INT _:_./ _)" 10)
  "@UNION" :: "[pttrn, set, set] ⇒ set"        ("(3UN _:_./ _)" 10)
  "*Ball"     :: "[pttrn, set, o] ⇒ o"         ("(3ALL _:_./ _)" [0, 0, 10] 10)
  "*Bex"      :: "[pttrn, set, o] ⇒ o"         ("(3EX _:_./ _)" [0, 0, 10] 10)

translations
  "UNIV"          == "Compl({})"
  "x ~: y"        == "~ (x : y)"
  "{x, xs}"       == "insert(x, {xs})"
  "{x}"           == "insert(x, {})"
  "{x. P}"        == "Collect(λx. P)"
  "INT x:A. B" == "INTER(A, (λx. B))"
  "UN x:A. B"  == "UNION(A, (λx. B))"
  "ALL x:A. P" == "Ball(A, (λx. P))"
  "EX x:A. P"  == "Bex(A, (λx. P))"

axioms
  ext:          "A = B ⟷(∀x. ((x:A) ⟷(x:B)))"
  Collect:      "(t : { x. P(x) }) ⟷(P(t))"

syntax ("" output)
  "_setle"         :: "[set, set] ⇒ o"          ("op <=")
  "_setle"         :: "[set, set] ⇒ o"          ("(_/ <= _)" [50, 51] 50)
  "_setless"       :: "[set, set] ⇒ o"          ("op <")
  "_setless"       :: "[set, set] ⇒ o"          ("(_/ < _)" [50, 51] 50)

syntax (symbols)
  "_setle"         :: "[set, set] ⇒ o"          ("op ⊆")
  "_setle"         :: "[set, set] ⇒ o"          ("(_/ ⊆ _)" [50, 51] 50)
  "_setless"       :: "[set, set] ⇒ o"          ("op ⊂")
  "_setless"       :: "[set, set] ⇒ o"          ("(_/ ⊂ _)" [50, 51] 50)
  "op Int"         :: "[set, set] ⇒ set"        (infixl "∩" 70)
  "op Un"          :: "[set, set] ⇒ set"        (infixl "∪" 65)
  "op :"           :: "['a, set] ⇒ o"           ("op ∈")
  "op :"           :: "['a, set] ⇒ o"           ("(_/ ∈ _)" [50, 51] 50)
  "op ~:"          :: "['a, set] ⇒ o"           ("op ∉")
  "op ~:"          :: "['a, set] ⇒ o"           ("(_/ ∉ _)" [50, 51] 50)
  "UN "            :: "[idts, o] ⇒ o"           ("(3⋃ _./ _)" 10)
  "INT "           :: "[idts, o] ⇒ o"           ("(3⋂ _./ _)" 10)
  "@UNION1"        :: "[pttrn, set] ⇒ set"      ("(3⋃ _./ _)" 10)
```

```
  "@INTER1"    :: "[pttrn, set] ⇒ set"         ("(3⋂ _./ _)" 10)
  "@UNION"     :: "[pttrn, set, set] ⇒ set"    ("(3⋃ _∈_./ _)" 10)
  "@INTER"     :: "[pttrn, set, set] ⇒ set"    ("(3⋂ _∈_./ _)" 10)
  "_Ball"      :: "[pttrn, set, o] ⇒ o"        ("(3∀ _∈_./ _)" [0, 0, 10] 10)
  "_Bex"       :: "[pttrn, set, o] ⇒ o"        ("(3∃ _∈_./ _)" [0, 0, 10] 10)

translations
  "op ⊆" == "op <= :: [_ set, _ set] => o"


defs
  subset_def: "A <= B          ≡∀x. x∈A ⟶x∈B"
  empty_def:  "{}              ≡{x. False}"
  Minus_def:  "A Minus B       ≡{x. x ∈ A ∧x ∉ B}"
  Un_def:     "A Un B          ≡{x. x ∈ A ∨x ∈ B}"
  Int_def:    "A Int B         ≡{x. x ∈ A ∧x ∈ B}"
  Ball_def:   "Ball(A, P)      ≡(∀ x. x∈A ⟶P(x))"
  Bex_def:    "Bex(A, P)       ≡(∃ x. x∈A ∧P(x))"
  Compl_def:  "Compl(A)        ≡{x. ¬x:A}"
  INTER_def: "INTER(A, B)     ≡{y. ALL x:A. y:B(x)}"
  UNION_def: "UNION(A, B)     ≡ {y. EX x:A. y:B(x)}"
  insert_def : " insert (a, B)  ≡{x. x=a} Un B"
  Pow_def:    "Pow(A)          ≡{B. B <= A}"


lemma inI: "(P(t)) ⟹(t ∈ { x. P(x) })"
  apply(rule iffD2)
  apply(rule Collect)
  apply(assumption)
  done

lemma "inE2": "(t ∈{ x. P(x) }) ⟹P(t)"
  apply(rule iffD1)
  apply(rule Collect)
  apply(assumption)
  done

lemma inE: assumes p1: "(t ∈{ x. P(x) })" assumes p2: "P(t) ⟹R" shows "R"
  apply(rule p2)
  apply(rule inE2)
  apply(rule p1)
  done

lemma equalsI: "(∀x. x∈A ⟷x∈B) ⟹A = B"
  apply(rule iffD2)
  apply(rule ext)
  apply(assumption)
  done

lemma equalsE2: "A = B ⟹(∀ x. x∈A ⟷x∈B)"
  apply(rule iffD1)
  apply(rule ext)
  apply(assumption)
  done

lemma equalsE: assumes p1: "A = B" assumes p2: "(∀ x. x∈A ⟷x∈B) ⟹R"
  shows "R"
  apply(rule p2)
  apply(rule equalsE2)
  apply(rule p1)
  done

end
```

4 FOL with Equality: Equational Reasoning

In this exercise, we will study elementary equational reasoning for groups and orders, and learn how to combine this with reasoning via case distinction. The technical level is deliberately rather low since elementary fall-back techniques are necessary if more automated tactics fail.

4.1 More on Isabelle

4.1.1 Backward Proof Control Structures

Revising our first proof scripts, it becomes clear that proof-scripts contain considerable repetition. Thus, more automation can be achieved by introducing control structures in the ISAR-language. These are:

control structures
ISAR

1. M,M' *sequential composition*: try tactic M; if it succeeds try tactic M'.

sequential composition (,)

2. M|M *alternative*: try tactic M; if it fails try tactic M'.

alternative (|)

3. M? *option*: try tactic M; if it fails report success.

option (?)

4. M+ *repetition*: try tactic M and repeat as long as no failure occurs.

repetition (+)

For example, instead of:

apply(*rule* X)
apply(*erule* Y)

we may write:

apply(*rule* X, *rule* Y)

Further, instead of:

apply(*drule* mp)
apply(*assumption*)
apply(*assumption*)
apply(*erule* disjE)

apply(*drule* mp)
apply(*erule* disjE)

we may write:

apply(*drule* mp,(*assumption*|*erule* disjE)+)+

4.1.2 FOL with Equality

In lecture, *first-order logic with equality* has been introduced as a logical system where the equality $x = y$ has been defined as a predicate on terms which represents a congruence relation. This is covered in Isabelle/FOL by the following rules:

equality $x = y$

refl

trans

sym

subst

```
refl :    "a=a"
trans:    "⟦ x=z; x=y ⟧ ⟹ y=z"
sym:      "y=x  ⟹  x=y"
subst:    "⟦ a=b;  P(a) ⟧ ⟹ P(b)"
```

Note that the substitutivity rule in Isabelle does not distinguish between "formulas" and "terms" as described in the lecture.

4.1.3 New Tactics

We introduce two new tactical commands for case splitting reasoning and performing one rewrite step. Both can be understood as abbreviation of previously introduced commands and/or rules. These are:

case_tac

1. *case_tac* "<form>", where <form> is a splitting formula. It is equivalent to: *insert* excluded_middle[**of** "<form>"], *erule* disjE

subst

2. *subst* *rule*, where *rule* is a (conditional) equation performed left-to-right. It is equivalent to: *rule* *subst*[**OF** sym[**OF** *rule*]]

Note that the *subst* chooses an arbitrary "position" where to perform a rewrite step; this lack of control may be sometimes undesirable. In such cases there may be no alternative to providing a more concrete substitution for meta variables, for example like *rule_tac* P = "λz. ?X ∗ z = e" **in** *subst*[**OF** *rule*]. Here, the λ-expression denotes a function that generates a term (with a "hole" $?X$). In general, giving too special substitutions is tedious and makes proof-scripts less robust; giving too general substitutions may result in a dead end of a proof.

λ-expression

symmetric

By the way, sym[**OF** *rule*] is also equivalent to *rule*[symmetric].

4.1.4 New Declaration Elements

In an ISAR theory file, proofs can be mixed with other syntactic elements such as type declarations, constant declarations, definitions and axioms (here only used as exercise!). Consider: type declarations constant declarations definitions axioms

typedecl <T>
arities <T> :: "term"

Here, the type <T> is declared; since Isabelle has a two-staged type system with "types of types" called *type classes*, the new type is declared to the class term introduced in the IFOL theory.

A standard *constant declaration* is given by an example:

```
consts
   If   :: "[o, i, i] ⇒ i"     ("(if (_)/ then (_)/ else (_))" [10] 10)
```

Here, If is declared to have type [o, i, i] ⇒ i which is notationally equivalent to o ⇒ i ⇒ i ⇒ i. The final phrase is a pragma to the Isabelle parser: the user is allowed to write if P then Q else R instead of If(P,Q,R).

There are two ways of possibilities to *define* declared constant. One is by *axioms* as in the following example:

```
axioms
   if_P :     "P ⟹ (if P then y else z) = y"
   if_notP : "¬P ⟹(if P then y else z) = z"
```

The other possibility is by a special type of axioms, called *definitions*:

```
defs
   if_def  :  "(if P then y else z) = <E>"
```

where <E> is a closed expression not containing the constant If (we do not have the semantic means to give a useful definition for If at the moment).

Use analogies to declarations in the IFOL and FOL theories of the Isabelle distribution. You can find these theories nicely formatted on the Isabelle website: `http://isabelle.in.tum.de/library/FOL/index.html`

4.1.5 Proof-State Massage

The standard **apply**-command usually effects only the first subgoal. Thus, it may be desirable to rotate the list of subgoals in a proof state. The **defer** n or **prefer** n commands move a subgoal to the last or the first position. **defer** n **prefer** n

For the choice of unifiers, the order of assumptions in a subgoal may be relevant. *rotate_tac* n rotates the assumptions of the first subgoal by n positions: *rotate_tac* n

from right to left if n is positive, and from left to right otherwise. The default value is one.

4.2 Exercises

4.2.1 Exercise 15

Derive the symmetry and transitivity rules for $=$

$$\frac{x = y}{y = x}\ \textit{sym} \qquad \frac{x = y \quad y = z}{x = z}\ \textit{trans}$$

using only applications of refl and *subst*.

Answer to Exercise 15

1. Proving symmetry:

```
lemma ex15_1: "x=y ⟹y=x"
  apply(erule subst)
  apply(rule refl )
  done
```

2. Proving transitivity:

```
lemma ex15_2: "⟦x=y;y=z⟧ ⟹x=z"
  apply(rule subst [of "y"])
  apply( frule refl )+
  done
```

4.2.2 Exercise 16

Prove the following group properties from the lecture *without* using the tactic command *subst*.

$$x^{-1} * x = e \text{ and } x * e = x$$

Hint: Declare a type i of sort *term* in Isabelle/FOL and the constants $\_^{-1}$, $\_*\_$ and e over i in your theory! (use analogies to declarations in the theories FOL and IFOL.

Hint: Take the "axioms" of group theory, namely *associativity*, *right identity* and *right inverse* as named assumptions in a backward proof.

Answer to Exercise 16

1. Declaration:

```
typedecl i
arities   i :: "term"

consts
  e             :: "i"
  inv           :: "i ⇒ i"
  "*"           :: "[i, i] ⇒ i"          (infixr 55)
```

2. Proving right-inverse:

```
lemma ex16_1 :
      assumes assoc   : "∀ x y z. x * (y * z) = (x * y) * z"
      assumes neutral : "∀ x. x * e = x"
      assumes inverse : "∀ x. x * inv(x) = e"
      shows             "inv(x) * x = e"
  apply( rule_tac P = "λ z. ?X * z = e" in subst[OF spec[OF neutral]])
  apply( rule_tac x1 = "inv(x)"
         in subst[OF spec[OF inverse]])
  apply( rule_tac x4 = "x"
           in subst[OF sym [OF spec[OF spec[OF spec [OF assoc]]]]])
  apply( rule_tac x2 = "x"
           in subst[OF sym[OF spec[OF inverse]]])
  apply( rule_tac x4 = "inv(x)"
           in subst[OF sym [OF spec[OF spec[OF spec [OF assoc]]]]])
  apply( rule_tac a = "inv(x)"
           in subst[OF sym[OF spec[OF neutral]]])
  apply( rule refl )
  done
```

3. Proving right-neutral:

```
lemma ex16_2 :
      assumes assoc   : "∀ x y z. x * (y * z) = (x * y) * z"
      assumes neutral : "∀ x. x * e = x"
      assumes inverse : "∀ x. x * inv(x) = e"
      shows             "e * x = x"
  apply( rule_tac x1 = "x" in subst[OF spec[OF inverse ]])
  apply( rule_tac x3 = "x"
```

```
        in subst[OF spec[OF spec[OF spec [OF assoc]]]])
  apply(rule subst[OF sym[OF ex16_1[OF assoc neutral inverse]]])
  apply(rule spec[OF neutral])
  done
```

An alternative, slightly more backward, slightly more automatic proof would be:

```
lemma ex16_2_alt :
      assumes assoc   : "∀ x y z. x ∗ (y ∗ z) = (x ∗ y) ∗ z"
      assumes neutral : "∀ x. x ∗ e = x"
      assumes inverse : "∀ x. x ∗ inv(x) = e"
      shows             "e ∗ x = x"
  apply(insert inverse assoc)
  apply(erule allE)+
  apply(rule_tac b = "e" in subst)
  apply(assumption)
  apply(rule_tac b = "(?x2 ∗ inv(?x2)) ∗ x" in subst)
  apply(assumption)
  apply(rule_tac b = "inv(?x2) ∗ x" in subst)
  apply(rule sym[OF ex16_1])
  prefer 4
  apply(insert neutral)
  apply(erule allE)+
  apply(assumption)
  apply(insert assoc neutral inverse, assumption)+
  done
```

4.2.3 Exercise 17

Declare a predicate _ <= _ of type i ⇒ i ⇒ o (similar to equality). Formalize that _ <= _ is total or antisymmetric and use this as assumption at need in the proofs.

Prove that:

1. ¬ x <= y ==> y <= x

2. ¬ y <= x ==> x <= y

3. ¬ y = x ==> ¬(x <= y)∨¬(y <= x)

4. y <= x ==> x = y ∨(¬x <= y)

Hint: Use *subst* and *case_tac* whenever possible.

Hint: Consider derived rules of classical logic like swap, contrapos and contrapos2.

swap
contrapos
contrapos2

Answer to Exercise 17

1. Declaration:

```
consts    "<="          :: "[i, i] ⇒ o"          (infixr 50)
```

2. Proving 1.:

```
lemma ex17_1:
      assumes total: "∀ x y. x <= y ∨y <= x"
      shows          "¬x <= y ⟹y <= x"
  apply(insert total)
  apply(erule allE)+
  apply(erule disjE | assumption | erule notE)+
  done
```

3. Proving 2.:

```
lemma ex17_2:
      assumes total: "∀ x y. x <= y ∨y <= x"
      shows          "¬y <= x ⟹x <= y"
  apply(insert total)
  apply(erule allE)+
  apply(erule disjE | assumption | erule notE)+
  done
```

4. Proving 3.:

```
lemma ex17_3:
      assumes antisymmetry: "∀ x y. x <= y ∧y <= x ⟶x = y"
      shows                 "¬y = x ⟹¬(x <= y) ∨¬(y <= x)"
  apply(insert antisymmetry)
  apply(erule swap)
  apply(erule allE)+
  apply(erule impE)
```

```
prefer 2
apply(assumption)
apply(rule conjI)
apply(erule swap)
apply(rule disjI2, assumption)
apply(erule swap)
apply(rule disjI1, assumption)
done
```

5. Proving 4.:

```
lemma ex17_4:
      assumes antisymmetry: "∀ x y. x <= y ∧y <= x ⟶x = y"
      shows                 "y <= x ⟹x = y ∨(¬x <= y)"
   apply(case_tac "y = x")
   apply(rule disjI1, rule sym, assumption)
   apply(drule ex17_3[OF antisymmetry])
   apply(erule disjE)
   apply(rule disjI2, assumption)
   apply(erule notE, assumption)
   done
```

4.2.4 Exercise 18

Declare the constant If (presented syntactically in mix-fix notation) and define it via the axioms:

```
if_P :     "P ⟹(if P then y else z) = y"
if_notP :  "¬P ⟹(if P then y else z) = z"
```

Assume in the sequel that _ <= _ is a partial order (i.e. reflexive, transitive, antisymmetric).

Declare and define the operation max based on _ <= _ and If.

Prove that max is

1. idempotent,
2. commutative
3. and left-idempotent (i.e. max(x,max(x,y)) = max(x,y))

Hint: Use *subst* and *case_tac* whenever possible.

Answer to Exercise 18

1. Declaration:

```
consts
    If          :: "[o, i, i] ⇒ i"      ("(if (_)/ then (_)/ else (_))" [10] 10)
    max         :: "[i, i] ⇒ i"

axioms
    if_P :     "P ⟹ (if P then y else z) = y"
    if_notP :  "¬P ⟹(if P then y else z) = z"

defs
    max_def:  "max(x,y) ≡( if x <= y then y else x)"
```

2. Proving idempotence:

```
lemma ex18_1 :
        assumes transitivity : "∀ x y z. x <= y ∧ y <= z ⟶x <= z"
        assumes reflexivity :  "∀ x. x <= x"
        assumes antisymmetry: "∀ x y. x <= y ∧y <= x ⟶x = y"
        assumes total:         "∀ x y. x <= y ∨y <= x"
        shows                   "max(x,x) = x"
    apply(unfold max_def)
    apply(subst if_P )
    apply( insert reflexivity )
    apply( erule allE , assumption)
    apply( rule refl )
    done
```

3. Proving commutativity:

```
lemma ex18_2 :
        assumes transitivity : "∀ x y z. x <= y ∧y <= z ⟶x <= z"
        assumes reflexivity :  "∀ x. x <= x"
        assumes antisymmetry: "∀ x y. x <= y ∧y <= x ⟶x = y"
        assumes total:         "∀ x y. x <= y ∨y <= x"
        shows                  "max(x,y) = max(y,x)"
    apply(case_tac "x = y")
    apply( rule_tac b = "y" in subst)
    apply(assumption)
    apply( rule refl )
```

```
apply(unfold max_def)
apply(case_tac "x <= y")
(*corresponds to:
  apply(insert excluded_middle[of "x <= y"], erule disjE) *)
apply(frule ex17_4[OF antisymmetry])
apply(erule disjE)
apply(erule notE, rule sym, assumption)
apply(subst if_notP)
(* corresponds to:
   apply(rule subst[OF sym[OF if_notP]]) *)
apply(assumption)
apply(subst if_P, assumption, rule refl)
apply(frule ex17_2[OF total])
apply(subst if_P, assumption)
apply(subst if_notP, assumption, rule refl)
done
```

4. Proving left-idempotence:

```
lemma ex18_3 :
      assumes transitivity : "∀ x y z. x <= y ∧y <= z ⟶x <= z"
      assumes reflexivity :  "∀ x. x <= x"
      assumes antisymmetry: "∀ x y. x <= y ∧y <= x ⟶x = y"
      assumes total:         "∀ x y. x <= y ∨y <= x"
      shows                  "max(x,max(x,y)) = max(x,y)"
  apply(case_tac "x = y")
  apply(rule_tac b = "y" in subst, assumption)
  apply(subst ex18_1[OF transitivity reflexivity antisymmetry total])
  apply(subst ex18_1[OF transitivity reflexivity antisymmetry total])
  apply(rule refl)
  apply(case_tac "x <= y")
  apply(unfold max_def)
  apply(subst if_P, assumption)+
  apply(rule refl)
  apply(subst if_notP, assumption)+
  apply(subst if_P, rule spec [OF reflexivity])
  apply(rule refl)
  done
```

5 λ-Calculus

In this exercise, we will will use Isabelle as a prototype tool to describe calculi (including binding) and to perform computations in them by using tactics involving backtracking. This will also deepen our understanding of the unification procedures used by Isabelle.

We will also introduce the concept of (parametric) Polymorphism which can be used to encode object languages including their type system.

5.1 Isabelle

5.1.1 The Context of this Exercise

In lecture "The λ-Calculus", we defined the syntax of the untyped λ-calculus by the following grammar: untyped λ-calculus

$$e \ ::= \ x \mid c \mid (ee) \mid (\lambda x.\, e)$$

together with conventions of left-associativity and iterated λ's in order to avoid cluttering the notation. Later, we defined a substitution on this raw syntax, and congruence relations on λ-terms such as α-, β- and η congruences.

In this exercise, we will use a particular representation technique for the untyped λ-calculus called *shallow embedding*. It can be found in the theory `http://www.infsec.ethz.ch/education/permanent/csmr/material/Lambda.thy` (which is based on FOL for purely technical reasons - the declaration part can be loaded even in `Pure`, the meta logic of Isabelle itself). Instead of e, we declare one universal type term — the presented calculus is thus untyped. The application is represented by the constant declaration "^" :: "[term, term] ⇒ term", consequently. Instead of defining an own substitution function, however, we define the abstraction as a constructor of a function; thus, it gets the type Abs :: "[term ⇒ term] ⇒ term" where ⇒ is the function space inherited from `Pure`. The notation lam x. P x is equivalent to Abs(λx . P x); recall that λ is the internal abstraction inherited from Isabelle/Pure. Thus, whenever we want to substitute a term into the body of an abstraction, we can just use the β-reduction provided by Isabelle/Pure (one also speaks

shallow embedding `Lambda.thy` term ^ Abs lam x. P x higher-order abstract syntax

of an "internalized" substitution provided by the shallow embedding of our language; or of using higher-order abstract syntax).

Our theory for the untyped λ-calculus also provides the β-reduction relation and the β-congruence by a set of axioms; note that we make no claims on the logical consistency of this exercise!

Further, it provides definitions for the standard combinators K,S and I and two versions of Y combinators.

In lecture it was said that the untyped λ-calculus is Turing-complete. We will show two core ingredients for such a proof: namely that data types (in particular: natural numbers) and fix-point combinators (enabling the presentation of recursive functional programs) can be represented inside the untyped λ-calculus.

5.1.2 Automated Proof Search Tactics

As mentioned in the lecture "Proof Search", Isabelle can organize proof-states in a tree-like fashion, which can therefore be searched according to depth-first or breadth-first strategies. The tactic command fast performs the former, according to introduction and elimination rules given to it. Introduction and Elimination rules are both subdivided into two classes:

safe rules

unsafe rule

1. *safe rules*, which transform a proof state into an equivalent one,
2. *unsafe rule*, which may transform a proof state into a logically weaker one.

Unsafe rules where tried in a limited way after safe rules did not succeed, and assumption is applied after no more unsafe rule applications are possible. Some syntactic variants for fast-commands are:

fast intro
fast elim

fast intro : *rules*
fast elim : *rules*

If the full context of assumptions should be included as well, one can append a ! to intro, elim, and dest, e.g.:

fast intro !: *rules*

5.2 Exercises

5.2.1 Exercise 18

As a warm-up, reduce the following terms to β-normal form in Isabelle.

1. SKK
2. SKS

Hint: Start with

```
lemma ex18_1: "S^K^K >−−> ?x"
```

In the end, the metavariable ?x should be instantiated to a term in β-normal form.

Hint: Do the proofs without using fast.

Answer to Exercise 18

1. Reducing SKK:

```
lemma ex18_1: "S^K^K >−−> ?x"
  apply(rule trans)
  apply(unfold K_def S_def)
  apply(rule appr)
  apply(rule beta)
  apply(rule trans)
  apply(rule beta)
  apply(rule trans)
  apply(rule epsi)
  apply(rule appr)
  apply(rule beta)
  apply(rule trans)
  apply(rule epsi)
  apply(rule beta)
  apply(fold I_def)
  apply(rule refl)
  done
```

2. Reducing SKS:

```
lemma ex18_2: "S^K^S >−−> ?x"
  apply(unfold K_def S_def)
  apply(rule trans)
  apply(rule appr)
  apply(rule beta)
```

```
apply(rule trans)
apply(rule beta)
apply(rule trans)
apply(rule epsi)
apply(rule appr)
apply(rule beta)
apply(rule trans)
apply(rule epsi)
apply(rule beta)
apply(fold I_def)
apply(rule refl)
done
```

5.2.2 Exercise 19

Automate the proofs from Ex. 18 using fast and the ISAR control structures. Thanks to automation, you should be able to show also the following reductions using the identical "proof script":

1. $SKKISS$
2. $SKIKISS$

Answer to Exercise 19

First we define a set of lemmas we want to apply

```
lemmas red_cs_isar = beta appl appr epsi
```

using fast we can now show SKK with the following script:

```
lemma "S^K^K >--> ?x"
  apply(unfold S_def K_def)
  apply(rule trans, fast intro!: red_cs_isar)+
  apply(fold I_def)
  apply(rule refl)
  done
```

This "script" also works for the other examples:

1. Reducing $SKKISS$:

```
lemma ex19_1: "S^K^K^I^S^S >-->?t"
  apply(unfold S_def K_def)
  apply(rule trans, fast intro!: red_cs_isar )+
  apply(fold S_def K_def I_def )
  apply(rule refl )
  done
```

2. Reducing $SKIKISS$:

```
lemma ex19_2: "S^K^I^K^I^S^S >-->?t"
  apply(unfold S_def I_def K_def)
  apply(rule trans, fast intro!: red_cs_isar )+
  apply(fold I_def S_def K_def )
  apply(rule refl )
  done
```

5.2.3 Exercise 20

Now show in Isabelle that for both Y-combinator versions enjoy a fix-point property, i.e. prove that:

1. $Y_T F >=< F(Y_T F)$ and
2. $Y_C F >=< F(Y_C F)$.

Is it possible to show $Y_T F$-->$F(Y_T F)$ and $Y_C F$-->$F(Y_C F)$?

Answer to Exercise 20

1. $Y_T F >=< F(Y_T F)$:

```
lemma ex_20_1: "YT^F >=< F^(YT^F)"
  apply(unfold YT_def)
  apply(rule trans_sym)
  apply(rule appr_sym)
  apply(rule beta_sym)
  apply(rule beta_sym)
  done
```

2. $Y_C F >=< F(Y_C F)$:

```
lemma ex20_2: "YC^F >=< F^(YC^F)"
  apply(unfold YC_def)
  apply( rule trans_sym)
  apply( rule beta_sym)
  apply( rule trans_sym)
  apply( rule beta_sym)
  apply( rule symm_sym)
  apply( rule appl_sym)
  apply( rule beta_sym)
  done
```

5.2.4 Exercise 21

Following a proposal by Alonzo Church, natural numbers n were encoded as the term

$$\lambda f x.\ \underbrace{f(f \ldots (f}_{n \text{ times}}\, x) \cdots)\ ,$$

which we abbreviate by writing $\lambda f x.\, f^n x$. The successor function and addition are given by the λ-terms:

$$\begin{aligned} succ &\equiv \lambda u f x.\, f(u f x) \\ add &\equiv \lambda u v f x.\, u f (v f x) \end{aligned}$$

Write a theory of the Church-Numerals with constants for $C0$,$C1$,$C2$ and *succ* and *add*.

Convince yourself that *succ* and *add* are indeed the successor and addition function, by evaluating them symbolically (i.e, on "terms" $\lambda f x.\, f^n x$ and $\lambda f x.\, f^m x$) under a suitable assumption.

Answer to Exercise 21

```
consts
  (* Church numerals *)
  C0        :: "term"
  C1        :: "term"
  C2        :: "term"
  C3        :: "term"
  C4        :: "term"
  C5        :: "term"
```

```
tt        :: "term"
ff        :: "term"
(* primitive recursive functions *)
zero      :: "term"
succ      :: "term"
add       :: "term"
IF        :: "term"
mult      :: "term"
pred      :: "term"
tuple     :: "term"
first     :: "term"
second    :: "term"
tup_succ  :: "term"
next_tup  :: "term"
```

defs

```
tt_def :    "tt ≡lam x. lam y. x"
ff_def :    "ff ≡lam x. lam y. y"

C0_def:     "C0 ≡lam f. lam x. x"
C1_def:     "C1 ≡lam f. lam x. f^x"
C2_def:     "C2 ≡lam f. lam x. f^(f^x)"
C3_def:     "C3 ≡lam f. lam x. f^(f^(f^x))"
C4_def:     "C4 ≡lam f. lam x. f^(f^(f^(f^x)))"
C5_def:     "C5 ≡lam f. lam x. f^(f^(f^(f^(f^x))))"

zero_def : "zero ≡lam x. x^(lam y. ff)^tt"
succ_def : "succ ≡lam x. lam y. lam z. y^(x^y^z)"
add_def:   "add ≡lam u. lam v. lam f. lam x.  u^f^(v^f^x)"
if_def :   " IF    ≡lam b. lam m. lam n. b^m^n"

mult_def:      "mult     ≡lam u. lam v. lam f. u^(v^f)"
tuple_def :    "tuple    ≡lam x. lam y. lam f. f^x^y"
first_def :    " first   ≡lam t. t^tt"
second_def:    "second   ≡lam t. t^ff"
tup_succ_def : "tup_succ ≡lam
                t. tuple^(succ^( first ^t))^(succ^(second^t))"
next_tup_def : "next_tup ≡lam
                t. tuple^(second^t)^(succ^(second^t))"
(*(0,0) → (0,1). (0,1) → (1,2). (1,2) → (2,3) etc.*)
```

```
pred_def: "pred ≡lam u. first ^(u^next_tup^(tuple^C0^C0))"
```

5.2.5 Exercise 22

Reduce the following terms:

1. *succ* C_0
2. *add* C_3 C_2

Answer to Exercise 22

1. *succ* C_0

```
apply(unfold C0_def succ_def)
apply(rule trans_sym)
apply(rule beta_sym)
apply(rule trans_sym)
apply(rule epsi_sym)+
apply(rule appl_sym)
apply(rule appr_sym)
apply(rule beta_sym)
apply(rule trans_sym)
apply(rule epsi_sym)+
apply(rule appl_sym)
apply(rule beta_sym)
apply(fold C1_def)
apply(rule refl_sym )
done
```

2. *add* C_3 C_2

```
apply(unfold C2_def C3_def add_def)
apply(rule trans_sym)
apply(rule appr_sym)
apply(rule beta_sym, rule trans_sym)+
apply(rule epsi_sym)+
apply(rule appr_sym)
apply(rule beta_sym)
apply(rule trans_sym)
apply(rule epsi_sym)+
apply(rule beta_sym)
apply(rule trans_sym)
```

```
apply(rule epsi_sym)+
apply(rule appl_sym)
apply(rule appl_sym)
apply(rule appl_sym)
apply(rule appr_sym)
apply(rule beta_sym)
apply(rule trans_sym)
apply(rule epsi_sym)+
apply(rule appl_sym)
apply(rule appl_sym)
apply(rule appl_sym)
apply(rule beta_sym)
apply(fold C5_def)
apply(rule refl_sym )
done
```

5.2.6 Exercise 23 (*optional*)

When applying a rule, Isabelle uses a process that is called *higher-order unification* for finding instantiations for meta-variables. Consider the unification problem

$$?\mathtt{P}(?\mathtt{b}) \ =_{\alpha\beta\eta} \ \mathtt{y} = \mathtt{x}$$

which has the solutions:

$$[?\mathtt{P} \leftarrow (\lambda \mathtt{z}.\ \mathtt{z} = \mathtt{x}),\ ?\mathtt{b} \leftarrow \mathtt{y}]$$
$$[?\mathtt{P} \leftarrow (\lambda \mathtt{z}.\ \mathtt{y} = \mathtt{z}),\ ?\mathtt{b} \leftarrow \mathtt{x}]$$
$$[?\mathtt{P} \leftarrow (\lambda \mathtt{z}.\ \mathtt{y} = \mathtt{x}), ?\mathtt{b} \leftarrow t] \qquad \text{(for any } t\text{)}$$

We can simulate higher-order unification inside `Lambda.thy` on the basis of ?P ^ ?x >=< add ^ C3 ^ C4.

1. Synthesize at least two solutions. You may use local substitutions or back.

2. Try to unify lam x. add ^ ?P ^ C4 >=< lam x. add ^ x ^ C4 and lam x. add ^ (?P ^ x) ^ C4 >=< lam x. add ^ x ^ C4

Answer to Exercise 23

1. Synthesize unifiers:

```
lemma ex_23_1_1: "?X^?Y >=< add^C3^C2"
  apply(rule appr_sym)
  apply(rule refl_sym )
  done
(* corresponds to X -> add^C3 and Y -> C2, i.e. first order
    unification solution . *)

lemma ex_23_1_2: "?X^?Y >=< add^C3^C2"
  apply(rule appr_sym)
  apply(rule beta_sym)
  done
(* artefact : (lam x. x) ^ (add ^ C3) ^ C2 >=< add ^ C3 ^ C2 *)


lemma ex_23_1_3: "?X^?Y >=< add^C3^C2"
  apply(rule beta_sym)
  back
  done
(* (lam x. add ^ C3 ^ x) ^ C2 >=< add ^ C3 ^ C2 *)

lemma ex_23_1_4: "?X^?Y >=< add^C3^C2"
  apply(rule beta_sym)back back
  done

lemma ex_23_1_5: "?X^?Y >=< add^C3^C2"
  apply(rule beta_sym)
  back
  back
  back
  done
```

2. Unification under binding:

```
lemma ex_23_2_1: "lam x. add ^ ?P ^ C3 >=< lam x. add ^ x ^ C3"
  apply(rule epsi_sym)
  apply(rule appr_sym)
  apply(rule appl_sym)
```

```
  (* "apply( rule  refl_sym )" does not work:
     since ?P does not depend on x and any substitution would
     produce a name-capture wrt. to x bound by meta-quantifier.
     Thus, ?P specifies a pattern that does not contain x!        *)
  oops

lemma ex_23_2_2: "lam x. add ^ (?P ^ x) ^ C3 >=< lam x. add ^ x ^ C3"
  apply( rule epsi_sym)
  apply( rule appr_sym)
  apply( rule appl_sym)
  apply( rule beta_sym)
  done
```

5.3 Encoding the untyped λ-calculus in in Isabelle

```
theory Lambda = FOL:

(* common definition for both calculi *)
typedecl
  "term"

arities
  "term" :: logic

consts
  Abs      :: "[term ⇒ term] ⇒ term"      (binder "lam " 10)
  "^"      :: "[term, term] ⇒ term"       (infixl 20)

  K        :: "term"
  I        :: "term"
  S        :: "term"

  B        :: "term"

  YC       :: "term"
  YT       :: "term"

defs
  K_def:   "K ≡lam x. (lam y. x)"
  I_def:   "I ≡lam x. x"
  S_def:   "S ≡lam x. (lam y. (lam z. x^z^(y^z)))"

  B_def:   "B ≡S^(K^S)^K"

  YC_def:  "YC ≡lam f. ((lam x. f^(x^x))^(lam x. f^(x^x)))"
  YT_def:  "YT ≡(lam z. lam x. x^(z^z^x))^(lam z. lam x. x^(z^z^x))"


  (* reduction λ−calculus *)
consts
  Red      :: "[term, term] ⇒ prop"       ("(_ >−−> _)")

axioms
  beta:    "(lam x. f(x))^a >−−> f(a)"
  refl:    "M >−−> M"
  trans:   "⟦ M >−−> N; N >−−> L ⟧⟹M >−−> L"
  appr:    "M >−−> N ⟹M^Z >−−> N^Z"
  appl:    "M >−−> N ⟹Z^M >−−> Z^N"
  epsi:    "⟦ !!x. M(x) >−−> N(x) ⟧⟹(lam x. M(x)) >−−> (lam x. N(x))"

  (* equational λ−calculus *)
consts
  Conv :: "[term, term] ⇒ prop"       ("(_ >=< _)")

axioms
  beta_sym:    "(lam x. f(x))^a >=< f(a)"
  refl_sym:    "M >=< M"
  symm_sym:    "M >=< N ⟹N >=< M"
  trans_sym:   "⟦ M >=< N; N >=< L ⟧⟹M >=< L"
  appr_sym:    "M >=< N ⟹M^Z >=< N^Z"
  appl_sym:    "M >=< N ⟹Z^M >=< Z^N"
  epsi_sym:    "⟦!!x. M(x) >=< N(x) ⟧⟹lam x. M(x) >=< lam x. N(x)"

  (* syntax setup *)
syntax (symbols)
  "lam "      :: "[idts, term] ⇒ term"          ("(3λ_./ _)" [0, 10] 10)

end
```

6 PL in LF

In this exercise, we will use a very powerful meta-logic, introduced under the name *LF* ("logical framework"). Its purpose is to represent not only the syntax of propositional logics (PL), but the deductive system in form of its natural deduction system. As a consequence, we will deepen our understanding of notions like *proof objects* and the propositions-as-types principle.

By encoding PL in LF, we also give an intuition into Isabelle and its character as logical framework itself—at the end, Isabelle's built-in logic *Pure* is used to encode LF with the same techniques as we are studying PL in LF.

6.1 Background

6.1.1 Revisiting LF

We briefly revisit the LF system as presented in the lecture. LF is defined as a λ-calculus with dependent types; these were represented by a several mutual recursive judgments formalizing *signatures* Σ and *contexts* Γ. LF

The basic theory `http://www.infsec.ethz.ch/education/permanent/csmr/material/LF.thy` contains a *shallow embedding* of the raw terms—also called: pseudo terms—of the λ-calculus (i.e. substitution and generation of free variables is done by *Pure*). However, the type-system is represented by axioms that define the notion of signature and context. As in previous exercises, we make no statement about the logical consistency of our presentation. `LF.thy` *shallow embedding*

6.1.2 Signatures and Contexts

Generally, a *signature* specifies the "constant symbols" (as opposed to variables). A signature Σ is a sequence of pairs of the form $c : \tau$, where c is a constant symbol and τ is a type. *signature*

A *context* specifies the types of the variables used in an expression. A context Γ is a sequence of pairs of the form $x : A$, where $x \in \mathit{Var}$ and A is a raw term. *context*

The axioms for signatures and contexts define inductively the subset of *valid* signatures and contexts.

6.1.3 The judgments of LF

Valid signatures and contexts are defined via three (mutually recursive) kinds of judgments:

1. judgments stating that a signature is valid, $\vdash_{sig} \Sigma$;
2. judgments stating that a context is valid, $\vdash_{con} \Gamma$;
3. judgments stating that a term has a certain type; this is a relation between a signature Σ, a context Γ and an expression of the form $t : A$, written $\Gamma \vdash_\Sigma t : A$.

Note, however, that our implementation of LF in Isabelle differs from the presentation in lecture in that there is no Σ and $\vdash_\Sigma$. Statements for them were simulated by constant declarations and suitable axioms.

The judgments in LF are of the form $x_1 : X_1 \ \ldots\ x_n : X_n \ \vdash x : X$. An example for a judgment is x:o y:o |− x:o.

The following table shows how the various syntactical entities of LF are written in `LF.thy`:

LF	`LF.thy`
$\Pi x^A. b$	Prod(A, λx.B) or Pi x:A. B
$A \to B$	A−>B
$\lambda x^A. b$	Abs(A, λx.B) or Lam x:A. B
$F(a)$ (application)	F^A

The notations Prod(A, λx.B) and Abs(A, λx.B) may be parsed and printed alternatively by Isabelle. There are also some differences between the LF presentation in the lecture and the way the *rules* are encoded in Isabelle:

- There is no assumption rule, since signatures are mimicked by contexts and by theory extensions.
- The hypothesis rule requires that the type assignment to be proven is the first in the context (which is implicitly assumed to be a set). In Isabelle/LF, the context is more a list-like structure which makes the introduction of a weakening-rule necessary.

6.2 Exercises

6.2.1 Exercise 24

Prove three of the following judgments in LF. To learn more, you might want to try and guess the instantiation of the metavariable in advance:

1. $i : \mathit{Type} \vdash \Pi x^i.\, \mathit{Type} :?T$
2. $A : \mathit{Type}, B : \mathit{Type} \vdash A \to B :?T$
3. $A : \mathit{Type}, B : \mathit{Type} \vdash \lambda x^A.\, A \to B :?T$
4. $f : \Pi x^A.\, B, a : A \vdash f(a) :?T$ (note how Isabelle displays $\Pi x^A.\, B$!)
5. $A : \mathit{Type}, P : A \to \mathit{Type}, a : A \vdash P(a) :?T$
6. $A : \mathit{Type}, P : A \to \mathit{Type} \vdash \lambda a^A.\, \lambda b^{P(a)}.\, b :?T$

Answer to Exercise 24

1. $i : \mathit{Type} \vdash \Pi x^i.\, \mathit{Type} :?T$:

```
lemma ex24_1: "i : Type |- (Pi x:i. Type) : ?T";
  apply(rule formation)
  apply(rule hypothesis)
  apply(rule axiom)
  done
```

2. $A : \mathit{Type}, B : \mathit{Type} \vdash A \to B :?T$:

```
lemma ex24_2: "A:Type B:Type |- A -> B :?T"
  apply(rule formation)
  apply(rule hypothesis)
  apply(rule weakening)
  apply(rule weakening)
  apply(rule hypothesis)
  done
```

3. $A : \mathit{Type}, B : \mathit{Type} \vdash \lambda x^A.\, A \to B :?T$:

```
lemma ex24_3: "A:Type B:Type |- Lam x:A. (A -> B) :?T"
  apply(rule abstraction )
  apply(rule formation)
  apply(rule weakening)
  apply(rule hypothesis)
  apply(rule weakening)
  apply(rule weakening)
  apply(rule weakening)
  apply(rule hypothesis)
```

```
apply(rule formation)
apply(rule hypothesis)
apply(rule axiom)
done
```

4. $f : \Pi x^A. B, a : A \vdash f(a) :?T$ (note how Isabelle displays $\Pi x^A. B$!):

```
lemma ex24_4: "f:(Pi x:A. B) a:A |- f^a:?T"
  apply(rule application)
  apply(rule hypothesis)
  apply(rule weakening)
  apply(rule hypothesis)
  done
```

5. $A : \mathit{Type}, P : A \to \mathit{Type}, a : A \vdash P(a) :?T$:

```
lemma ex24_5: "A : Type P : A -> Type a : A |- P^a : ?T"
  apply(rule application)
  apply(rule weakening)
  apply(rule hypothesis)
  apply(rule weakening)
  apply(rule weakening)
  apply(rule hypothesis)
  done
```

6. $A : \mathit{Type}, P : A \to \mathit{Type}, a : A \vdash P(a) :?T$:

```
lemma ex24_6: "A : Type P : A -> Type |- Lam a : A. Lam b : P^a. b : ?T"
  apply(rule abstraction)
  apply(rule abstraction)
  apply(rule hypothesis)
  apply(rule formation)
  apply(rule application)
  apply(rule weakening)
  apply(rule weakening)
  apply(rule hypothesis)
  apply(rule hypothesis)
  apply(rule application)
  apply(rule weakening)
  apply(rule weakening)
  apply(rule weakening)
```

```
apply(rule hypothesis)
apply(rule weakening)
apply(rule hypothesis)
apply(rule formation)
apply(rule hypothesis)
apply(rule formation)
apply(rule application)
apply(rule weakening)
apply(rule weakening)
apply(rule hypothesis)
apply(rule hypothesis)
apply(rule application)
apply(rule weakening)
apply(rule weakening)
apply(rule weakening)
apply(rule hypothesis)
apply(rule weakening)
apply(rule hypothesis)
done
```

6.2.2 Exercise 25

Encode syntax and deductive system of propositional logic (PL) and call the resulting theory PL_in_LF. The cases for and and or are sufficient.

Example for the syntax:

```
consts
  "o"          :: "term"
  "imp"        :: "term"

axioms
  o_def:       "G|- o:Type"
  imp_def:     "G|- imp: o->o->o"
```

Example for the deductive system:

```
consts
  "pr"         :: "term"
```

```
    "impl"        :: "term"

  axioms
    pr_def:       "G|- pr: o->Type"

    impl_def:     "G|- impl:Pi A:o. Pi B:o. (pr^A->pr^B)->pr ^(imp^A^B)"
```

Do not forget the impE-rule!

Answer to Exercise 25

```
theory PL_in_LF = LF:

(* ****************************************************** *)
(* Encoding the syntax for PL in LF                       *)
(* ****************************************************** *)

consts
  "o"          :: "term"
  "imp"        :: "term"
  "and"        :: "term"
  "or"         :: "term"

axioms
  o_def:       "G|- o:Type"

  imp_def:     "G|- imp: o->o->o"
  and_def:     "G|- and: o->o->o"
  or_def:      "G|- or: o->o->o"


(* ****************************************************** *)
(* Encoding the deductive system for PL in LF             *)
(* ****************************************************** *)

consts
  "pr"         :: "term"

  "impl"       :: "term"
  "impE"       :: "term"
```

```
  "andI"      :: "term"
  "andE1"     :: "term"
  "andE2"     :: "term"

  "orI1"      :: "term"
  "orI2"      :: "term"
  "orE"       :: "term"


axioms
  pr_def :     "G|- pr: o->Type"

  impI_def :   "G|- impI:Pi A:o. Pi B:o. (pr^A->pr^B)->pr ^(imp^A^B)"
  impE_def:    "G|- impE:Pi A:o. Pi B:o. pr^(imp^A^B)->pr^A->pr^B"

  andI_def :   "G|- andI:Pi A:o. Pi B:o. pr^A->pr^B->pr^(and^A^B)"
  andE1_def:   "G|- andE1:Pi A:o. Pi B:o. pr^(and^A^B)->pr^A"
  andE2_def:   "G|- andE2:Pi A:o. Pi B:o. pr^(and^A^B)->pr^B"

  orI1_def :   "G|- orI1:Pi A:o. Pi B:o. pr^A->pr^(or^A^B)"
  orI2_def :   "G|- orI2:Pi A:o. Pi B:o. pr^B->pr^(or^A^B)"
  orE_def:     "G|- orE:Pi A:o. Pi B:o. Pi C:o.
                      pr^(or^A^B)->(pr^A->pr^C)->(pr^B->pr^C)-> pr^C"

end
```

6.2.3 Exercise 26

Prove in PL_in_LF that $\vdash \Pi x^o. \Pi y^o. pr\ (imp\ x\ y) \to (pr\ x) \to (pr\ y) : Type$.

Answer to Exercise 26

```
lemma ex26: "|- Pi x:o. (Pi y:o. pr^(imp^x^y)-> pr^x -> pr^y): Type"
  apply(rule formation, rule o_def )+
  apply(rule formation)
  apply(rule application )
  apply(rule pr_def )
  apply(rule application )
  apply(rule application )
  apply(rule imp_def)
```

```
  apply( rule hypothesis | ( rule weakening, rule hypothesis) )+
apply( rule formation)
apply( rule application )
apply( rule pr_def)
apply( rule weakening)
  apply( rule hypothesis | ( rule weakening, rule hypothesis) )+
apply( rule application )
apply( rule pr_def)
  apply( rule weakening)
  apply( rule hypothesis | ( rule weakening, rule hypothesis) )+
done
```

6.2.4 Exercise 27

Prove one of the following propositions in PL_in_LF:

1. $a \rightarrow a$
2. $a \rightarrow b \rightarrow a$

Hints:

1. One states that a proof for the goal is a term of type $pr(imp\ a\ a)$, and gives a proof object for it, i.e. one states

$$a : o \vdash ?t : pr(imp\ a\ a)$$

   for an appropriate, given t and proves this statement.

2. Alternatively, one *synthesizes* the $?t$ through the meta-level proof. Since the unifications for the `application`-rule are highly ambiguous (Isabelle may even be unable to find existing unifiers!), you will have to make tricky explicit instantiations. An (unsafe and incomplete) alternative is to use `back` until Isabelle has found the right unifier.

3. The proof object for the second exercise is:

   impl^a^(imp^b^a)^ (Lam x: pr^a. impl^b^a^(Lam xa:pr^b. x))

   The proof is difficult.

Answer to Exercise 27

1. $a \to a$:

```
(* Version with all crucial substitutions made explicit ... *)
lemma ex27_1a: "a:o |- impl^a^a^(Lam y:pr^a. y) : pr^(imp^a^a)"
  apply( rule_tac A = "pr^a -> pr^a" and
                  B = "%u. pr^(imp^a^a)"
         in application )
  apply( rule_tac a = "a" and
                  A = "o" and
                  B = "%u. (pr^a -> pr^u) -> pr^(imp^a^u)"
         in       application )
  apply( rule_tac a = "a"  and
                  A = "o" and
                  B = "?X"
         in       application )
  apply( rule impl_def)
  apply( rule hypothesis)+
  apply( rule abstraction )
  apply( rule hypothesis)
  apply( rule formation)
  apply( rule application )
  apply( rule pr_def)
  apply( rule hypothesis)
  apply( rule application )
  apply( rule pr_def)
  apply( rule weakening)
  apply( rule hypothesis)
  done



(* version more relaxed wrt. explicit substitutions ... *)
lemma ex27_1b: "a:o |- impl^a^a^(Lam y:pr^a. y) : pr^(imp^a^a)"
  apply( rule application )
  apply( rule_tac B = "%u. ?A2(u) -> pr^(imp^a^u)"
         in       application )
  apply( rule application )
  apply( rule impl_def)
  apply( rule hypothesis)+
```

```
  apply( rule abstraction )
  apply( rule hypothesis)
  apply( rule formation)
  apply( rule application )
  apply( rule pr_def )
  apply( rule hypothesis)
  apply( rule application )
  apply( rule pr_def )
  apply( rule weakening)
  apply( rule hypothesis)
  done
```

```
(* synthetic version *)
lemma ex27_1c: "a:o |- ?T : pr^(imp^a^a)"
  apply( rule_tac B = "%u. pr^(imp^a^a)" in application)
  apply( rule_tac B = "%u. ?A3(u) -> pr^(imp^a^u)" in application)
  apply( rule_tac B = "%u. Pi ua : ?A8(u).
                           Pi uaa: ?A7(u, ua).
                                   pr^(imp^u^ua)" in application )
  apply( rule impl_def )
  apply( rule hypothesis)+
  apply( rule abstraction )
  apply( rule hypothesis)
  apply( rule formation)
  apply( rule application )
  apply( rule pr_def )
  apply( rule hypothesis)
  apply( rule application )
  apply( rule pr_def )
  apply( rule weakening)
  apply( rule hypothesis)
  done
```

2. $a \to b \to a$:

```
lemma ex27_2:
"a:o b:o
|- impl^a^(imp^b^a)^
   (Lam x: pr^a. impl^b^a^(Lam xa:pr^b. x)) :
```

```
 pr^(imp^a^(imp^b^a))"
apply( rule_tac  A = "pr ^ a -> pr ^ (imp ^ b ^ a)" and
                 B = "% u. pr^(imp^a^(imp^b^a))"
      in  application )
apply( rule_tac  A = "o" and
                 a = "imp^b^a" and
                 B = "%x. (pr^a -> pr^x) -> pr^(imp^a^x)"
      in  application )
apply( rule_tac  A = "o" and
                 a = "a" and
                 B = "%xa. Pi x:o. ((pr ^ xa -> pr ^ x)
                                  -> pr^(imp^xa^x))"
      in  application )
apply( rule  impl_def )
  apply( rule  hypothesis  |  ( rule  weakening, rule  hypothesis )  )+
apply( rule  application )
apply( rule  application )
apply( rule  imp_def)
  apply( rule  hypothesis  |  ( rule  weakening, rule  hypothesis )  )+
apply( rule  abstraction )

apply( rule  application )
apply( rule  application )
apply( rule  application )
apply( rule  impl_def )
  apply( rule  weakening)
  apply( rule  hypothesis  |  ( rule  weakening, rule  hypothesis )  )+
apply( rule  abstraction )
  apply( rule  weakening)
  apply( rule  hypothesis )  (* the discharge *)
apply( rule  formation )
  apply( rule  application )
  apply( rule  pr_def )
  apply( rule  weakening)
  apply( rule  hypothesis  |  ( rule  weakening, rule  hypothesis )  )+
  apply( rule  application )
  apply( rule  pr_def )
  apply( rule  weakening)
  apply( rule  hypothesis  |  ( rule  weakening, rule  hypothesis )  )+
apply( rule  formation )
```

```
    apply( rule  application )
    apply( rule  pr_def )
    apply( rule  hypothesis )
    apply( rule  application )
    apply( rule  pr_def )
  apply( rule  weakening)
  apply( rule  application )+
  apply( rule  imp_def)
    apply( rule  hypothesis  |  ( rule  weakening, rule  hypothesis )  )+
  done
```

6.2.5 Exercise 28

Prove one of the following propositions in PL_in_LF:

1. $a \wedge b \rightarrow a$
2. $a \rightarrow b \vee a$

Answer to Exercise 28

1. $a \wedge b \rightarrow a$:

```
(* Version with all crucial substitutions made explicit ... *)
lemma ex28_1a:
  "a:o b:o |- impl ^ (and ^ ?a ^ ?b) ^ ?a ^
                (andE1 ^ ?a ^ ?b) : pr^(imp^(and^a^b)^a)"
  apply( rule_tac  A ="pr^(and^a^b) -> pr^a" and
                   B ="%u. pr ^ (imp ^ (and ^ a ^ b) ^ a)"
        in         application )
  apply( rule_tac  a = "a" and
                   A = "o" and
                   B ="%u. (pr^(and^a^b) -> pr^u) -> pr^(imp^(and^a^b)^u)"
        in         application )
  apply( rule_tac  a = "and ^ a ^ b" and
                   A = "o" and
                   B = "%xa. Pi x:o. (  (pr^(xa) -> pr^x)
                                   -> pr^(imp^(xa)^x))"
        in  application )
```

```
  apply(rule impl_def)
    apply(rule application )
    apply(rule application )
    apply(rule and_def)
    apply(rule hypothesis | (rule weakening, rule hypothesis) )+

  apply( rule_tac B = "%u. Prod(pr ^ (and ^ a ^ u), %uu. pr ^ a)"
         in application )
  apply(rule application )

  apply(rule andE1_def)
    apply(rule hypothesis | (rule weakening, rule hypothesis) )+
  done


(* version more relaxed wrt. explicit substitutions ... *)
lemma ex28_1b: "a:o b:o |- ?T : pr^(imp^(and^a^b)^a)"
  apply( rule_tac B ="%u. pr ^ (imp ^ (and ^ a ^ b) ^ a)"
         in application )
  apply( rule_tac B ="%u. ?A3(u) -> pr ^ (imp ^ (and ^ a ^ b) ^ u)"
         in application )
  apply(rule application )  back back

  apply(rule impl_def)
    apply(rule application )
    apply(rule application )
    apply(rule and_def)
    apply(rule hypothesis | (rule weakening, rule hypothesis) )+

  apply( rule_tac B = "%u. pr^(and^a^u) -> pr^a"
         in application )
  apply(rule application )  back back

  apply(rule andE1_def)
    apply(rule hypothesis | (rule weakening, rule hypothesis) )+
  done
```

2. $a \to b \vee a$:

```
lemma ex28_2: "a:o b:o |- ?T : pr^(imp^a^(or^b^a))"
```

```
apply( rule_tac  B =''%u. pr ^ (imp^a^(or^b^a))''
        in          application )
apply( rule_tac  B =''%u. ?A3(u) -> pr^(imp^a^u)''
        in          application )
apply( rule_tac  B=''%u. Pi ua:?X1(u). (?X2(u,ua) -> pr^(imp^u^ua))''
        in  application )
apply( rule  impl_def )
  apply( rule  hypothesis | ( rule  weakening, rule  hypothesis ) )+

apply( rule  application )+
apply( rule  or_def )
  apply( rule  hypothesis | ( rule  weakening, rule  hypothesis ) )+

apply( rule_tac  B = ''%u. pr^u -> pr^(or^b^u)''
        in  application )
apply( rule_tac  B = ''%u. Pi ua:?X1(u). (pr^ua -> pr^(or^u^ua))''
        in  application )
apply( rule  orI2_def )
   apply( rule  hypothesis | ( rule  weakening, rule  hypothesis ) )+
done
```

7 HOL: Derived Rules

In the lecture, standard and non-standard models of HOL have been presented in informal notation based on ZF set theory.

On this basis, a small set of axioms is justified, which serve as foundation of HOL. In this exercise, we will prove the basic logical rules of Higher-order logic (HOL) from these axioms and elementary definitions.

7.1 Background

7.1.1 Higher-order Logic

We have seen in lecture "HOL: Deriving Rules" how all well-known inference rules for logical connectives and quantifiers can be derived in HOL. We now want to do some of these proofs in Isabelle. Those rules are available by default since they are derived from the eight basic rules once and for all.

Of course, these rules are already proved in the standard Isabelle/HOL library. Nevertheless, do not to use library proofs for them and apply automated tactics only with your own derived rules.

Following general convention, the syntax for function application in HOL is just f x instead of f(x) as in FOL. function application

7.2 Isabelle/HOL

7.2.1 Technicalities

As for FOL you have to tell Isabelle that you want to work in HOL; choose HOL by selecting ⟨`Isabelle/Isar` ▷ ⟨`Logics` ▷ `HOL`⟩⟩. Within Isabelle/HOL the basic theory (on which you build your own theory) is called Main, thus your basic theory file for this exercise should look like: HOL Main

```
theory ex7 = Main:
lemma fun_cong: "f=g ⟹f(x) = g(x)"

end
```

7.2.2 The Logical Foundation

```
True_def:      "True          ≡((λx::bool. x) = (λx. x))"
All_def :      "∀ P ≡(P = (λx. True))"
Ex_def:        "∃ P ≡∀Q. (∀x. P x ⟶Q) ⟶Q"
False_def :    "False         ≡(∀P. P)"
not_def:       "¬ P     ≡P⟶False"
and_def:       "P ∧Q    ≡∀R. (P⟶Q⟶R) ⟶R"
or_def :       "P ∨Q    ≡∀R. (P⟶R) ⟶(Q⟶R) ⟶R"
if_def :       " If P x y     ≡THE z::'a. (P=True ⟶z=x)
                                ∧ (P=False ⟶z=y)"

eq_reflection : "(x=y) ⟹(x≡y)"

refl :         "t = (t::'a)"
subst:         "⟦ s = t; P(s) ⟧ ⟹ P(t::'a)"

ext:           "(⋀x::'a. (f x ::'b) = g x) ⟹ (λx. f x) = (λx. g x)"

the_eq_trivial : "(THE x. x = a) = (a::'a)"

impl:          "(P ≡Q) ⟹P⟶Q"
mp:            "⟦ P⟶Q; P ⟧ ⟹ Q"

iff :          "(P⟶Q) ⟶(Q⟶P) ⟶(P=Q)"
True_or_False : "(P=True) ∨(P=False)"
```

THE-operator

The axiom of the THE-operator seems to be obviously true, but somewhat pointless. In each type τ there is a function assigned to this operator, that chooses out of the set of possible values in the semantic domain of τ the element, that is equal to a. However, since we may write THE x. P x, the THE-operator may be used quite flexibly to define elements that are uniquely defined by a predicate P; in other words: the use of the operator boils down to the proof of uniqueness with respect to P.

7.2.3 Exercise 29

Derive the following rules:

1. $f = g \Longrightarrow f(x) = g(x)$ (*fun_cong*)
2. $x = y \Longrightarrow f(x) = f(y)$ (*arg_cong*)

Answer to Exercise 29

1. *fun_cong*:

```
lemma fun_cong: "f=g ⟹f(x) = g(x)"
  apply(erule subst)
  apply(rule refl )
  done
```

2. *arg_cong*:

```
lemma arg_cong: "x=y ⟹f(x) = f(y)"
  apply(rule subst [of x])
  apply(assumption)
  apply(rule refl )
  done
```

7.2.4 Exercise 30

Derive the following rules presented in the lecture:

1. transitivity and symmetry

$$s = t \Longrightarrow t = s \qquad (sym)$$
$$[\![r = s; s = t]\!] \Longrightarrow r = t \qquad (trans)$$

2. rules about *iff*:

$$[\![P \Longrightarrow Q; Q \Longrightarrow P]\!] \Longrightarrow P = Q \qquad (iffI)$$
$$[\![P = Q; Q]\!] \Longrightarrow P \qquad (iffD2)$$

3. rules about *True*:

$$True \qquad (TrueI)$$
$$P = True \Longrightarrow P \qquad (eqTrueE)$$
$$P \Longrightarrow P = True \qquad (eqTrueI)$$

4. rules about ∀:

$$(\bigwedge x. P\, x) \Longrightarrow \forall x. P\, x \qquad (allI)$$
$$(\forall x. P\, x) \Longrightarrow P\, x \qquad (spec)$$

5. rules about *False*:

$$\begin{aligned} \mathit{False} &\Longrightarrow P && (\mathit{FalseE}) \\ \mathit{False} = \mathit{True} &\Longrightarrow P && (\mathit{False\_neq\_True}) \\ \mathit{True} = \mathit{False} &\Longrightarrow P && (\mathit{True\_neq\_False}) \end{aligned}$$

6. rules about ¬:

$$\begin{aligned} (P \Longrightarrow \mathit{False}) \Longrightarrow \neg P &\quad (\mathit{notI}) \\ [\![\neg P; P]\!] \Longrightarrow R &\quad (\mathit{notE}) \\ \neg(\mathit{True} = \mathit{False}) &\quad (\mathit{True\_Not\_False}) \end{aligned}$$

7. rules about ∃:

$$\begin{aligned} P(x) \Longrightarrow \exists x. P\, x &\qquad (\mathit{exI}) \\ [\![(\exists x. P\, x); \bigwedge x. P\, x \Longrightarrow Q]\!] \Longrightarrow Q &\qquad (\mathit{exE}) \end{aligned}$$

8. rules about ∧:

$$\begin{aligned} [\![P; Q]\!] \Longrightarrow P \wedge Q &\qquad (\mathit{conjI}) \\ P \wedge Q \Longrightarrow P &\qquad (\mathit{conjEL}) \\ P \wedge Q \Longrightarrow Q &\qquad (\mathit{conjER}) \\ [\![P \wedge Q; [\![P; Q]\!] \Longrightarrow R]\!] \Longrightarrow R &\qquad (\mathit{conjE}) \end{aligned}$$

9. rules about ∨:

$$\begin{aligned} P \Longrightarrow P \vee Q &\qquad (\mathit{disjIL}) \\ Q \Longrightarrow P \vee Q &\qquad (\mathit{disjIR}) \\ [\![P \vee Q; P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R &\qquad (\mathit{disjE}) \end{aligned}$$

10. and finaly, exluded middle:

$$P \vee \neg P \quad (\mathit{excluded\ middle})$$

Answer to Exercise 30

1.

```
lemma sym: "s=t ⟹t=s"
  apply(erule subst)
  apply(rule refl)
  done

lemma trans: assumes pr1: "r=s" and pr2:"s=t" shows "r=t"
  apply(rule_tac t = "t" and s = "s" in subst)
  apply(rule pr2)
  apply(rule_tac t = "r" and s = "s" in subst)
  apply(rule sym)
  apply(rule pr1)
  apply(rule refl)
  done
```

2.

```
lemma iffI: assumes pr1: "P⟹Q" and pr2:"Q⟹P" shows "P=Q"
  apply(rule mp)
  apply(rule mp)
  apply(rule iff)
  apply(rule impI)
  apply(erule pr1)
  apply(rule impI)
  apply(erule pr2)
  done

lemma iffD2: "⟦P=Q;Q⟧ ⟹P"
  apply(rule subst)
  apply(rule sym)
  apply(assumption)+
  done
```

3.

```
lemma TrueI: "True"
```

```
  apply(unfold True_def)
  apply(rule refl )
  done

lemma eqTrueE: "P=True ⟹P"
  apply(rule iffD2)
  apply(assumption)
  apply(rule TrueI)
  done

lemma eqTrueI: "P⟹P=True"
  apply(rule iffI )
  apply(rule TrueI)
  apply(assumption)
  done
```

4.

```
lemma allI: assumes prem: "(⋀ x. P x)" shows "∀x. P x"
  apply(unfold All_def )
  apply(rule ext)
  apply(rule eqTrueI)
  apply(rule prem)
  done

lemma spec: "(ALL x. P x) ⟹P x"
  apply(unfold All_def )
  apply(rule eqTrueE)
  apply(erule fun_cong)
  done
```

5.

```
lemma FalseE: "False ⟹P"
  apply(unfold False_def )
  apply(erule spec)
  done

lemma False_neq_True: "False = True ⟹P"
```

```
  apply(rule FalseE)
  apply(erule eqTrueE)
  done

lemma True_neq_False: "True = False ⟹P"
  apply(rule FalseE)
  apply(rule eqTrueE)
  apply(erule sym)
  done
```

6.

```
lemma notI: assumes prem: "(P⟹False)" shows "¬P"
  apply(unfold not_def)
  apply(rule impI)
  apply(erule prem)
  done

lemma notE: "⟦ ¬P; P ⟧⟹Q"
  apply(unfold not_def)
  apply(rule FalseE)
  apply(rule mp)
  apply(assumption)+
  done

lemma True_not_False: "¬(True = False)"
  apply(rule notI)
  apply(erule True_neq_False)
  done
```

7.

```
lemma existsI: "P x ⟹ ∃x. P x"
  apply(unfold Ex_def)
  apply(rule allI )
  apply(rule impI)
  apply(rule mp)
  apply(erule spec)
  apply(assumption)
```

```
  done

lemma existsE: assumes p1: "(∃x. P x)" and p2: "⋀x. P x ⟹ Q"
  shows "Q"
  apply(rule p1 [unfolded Ex_def, THEN spec, THEN mp])
  apply(rule impI [THEN allI ])
  apply(erule p2)
  done
```

8.

```
lemma conjI: assumes p1: "P" and p2: "Q" shows "P ∧Q"
  apply(unfold and_def)
  apply(rule allI )
  apply(rule impI)
  apply(rule mp)
  prefer 2
  apply(rule p2)
  apply(rule mp)
  prefer 2
  apply(rule p1)
  apply(assumption)
  done

lemma conjEL: assumes prem: "P ∧Q" shows "P"
  apply(rule mp)
  prefer 2
  apply(rule prem)
  apply(rule impI)
  apply(unfold and_def)
  apply(rule mp)
  apply( rule_tac x = "P" in spec)
  apply(assumption)
  apply(rule impI)+
  apply(assumption)
  done

lemma conjER: assumes prem: "P ∧Q" shows "Q"
  apply(rule mp)
```

```
  prefer 2
  apply(rule prem)
  apply(rule impI)
  apply(unfold and_def)
  apply(rule mp)
  apply(rule_tac x = "Q" in spec)
  apply(assumption)
  apply(rule impI)+
  apply(assumption)
  done

lemma conjE: assumes p1: "P ∧Q" and p2: "⟦P;Q⟧⟹R"
  shows "R"
  apply(rule p2)
  apply(rule conjEL)
  apply(rule p1)
  apply(rule conjER)
  apply(rule p1)
  done
```

9.

```
lemma disjIL: assumes prem: "P" shows "P ∨Q"
  apply(rule mp)
  prefer 2
  apply(rule prem)
  apply(rule impI)
  apply(unfold or_def)
  apply(rule allI )
  apply(rule impI)+
  apply(rule mp)
  prefer 2
  apply(assumption)+
  done

lemma disjIR: assumes prem: "Q" shows "P ∨Q"
  apply(rule mp)
  prefer 2
  apply(rule prem)
```

```
  apply(rule impI)
  apply(unfold or_def)
  apply(rule allI)
  apply(rule impI)+
  apply(rule mp)
  prefer 2
  apply(assumption)+
  done

lemma disjE: assumes p1:"P ∨Q" and p2: "P⟹R" and p3: "Q⟹R"
  shows "R"
  apply(rule mp)
  prefer 2
  apply(rule p1)
  apply(rule impI)
  apply(unfold or_def)
  apply(drule spec)
  apply(rule mp)
  apply(rule mp)
  apply(assumption)
  apply(rule impI)
  apply(rule p2)
  apply(assumption)
  apply(rule impI)
  apply(rule p3)
  apply(assumption)
  done
```

10.

```
lemma excluded_middle: "P ∨¬P"
  apply(rule_tac P = "P=True" and Q = "P=False" in disjE)
  apply(rule True_or_False)
  apply(drule eqTrueE)
  apply(rule disjIL)
  apply(assumption)
  apply(rule disjIR)
  apply(unfold not_def)
  apply(rule_tac t = "P" and s = "False" in subst)
```

```
apply(rule sym)
apply(assumption)
apply(rule impI)
apply(assumption)
done
```

7.2.5 Exercise 31

Prove the following properties:

$$[\![P\ a; \bigwedge x.P\ x \Longrightarrow x = a]\!] \Longrightarrow (\text{THE } x.P\ x) = a \quad (\mathit{the\_equality})$$

Answer to Exercise 31

```
lemma the_equality: assumes prema: "P a" and premb: "⋀x. P x ⟹x=a"
      shows "(THE x. P x) = a"
  apply(rule trans)
  prefer 2
  apply(rule  the_eq_trivial )
  apply( rule_tac f="The" in arg_cong)
  apply(rule ext)
  apply(rule iffI )
  apply(erule premb)
  apply(erule ssubst)
  apply(rule prema)
  done
```

7.2.6 Exercise 32

Prove the following two properties of the if –then–else: if –then–else

$$Q = \mathit{True} \Longrightarrow (\text{if } Q \text{ then } x \text{ else } y) = x \qquad (\mathit{ite\_then})$$
$$Q = \mathit{False} \Longrightarrow (\text{if } Q \text{ then } x \text{ else } y\ ) = y \qquad (\mathit{ite\_else})$$

Answer to Exercise 32

```
lemma ite_then: "Q= True ⟹(if Q then x else y) = x"
  apply(unfold if_def )
  apply(rule the_equality )
  apply(rule conjI )
  apply(rule impI)
```

```
  apply(rule refl)
  apply(rule impI)
  apply(drule sym)
  apply(rotate_tac 1)
  apply(drule trans)
  apply(assumption)
  apply(erule True_neq_False)
  apply(erule conjE)
  apply(erule impE)
  apply(assumption)+
  done

lemma ite_else: "Q = False ⟹(if Q then x else y) = y"
  apply(unfold if_def)
  apply(rule the_equality)
  apply(rule conjI)
  apply(rule impI)
  apply(drule sym)
  apply(rotate_tac 1)
  apply(drule trans)
  apply(assumption)
  apply(erule False_neq_True)
  apply(rule impI)
  apply(rule refl)
  apply(erule conjE)
  apply(rotate_tac −1)
  apply(erule impE)
  apply(assumption)+
  done
```

8 HOL: Axiomatic Classes and Typed Set Theory

In this exercise, we will deepen our knowledge on a specific concept of theory structuring in Isabelle, namely axiomatic classes. We will extend conservative library constructions in typed set theory, and will lay the groundwork for inductive definitions.

Technically, we will apply automated proof procedures, be it on the level of rewriting or tableaux based procedures and combined methods such as auto.

8.1 Isabelle

8.1.1 Axiomatic Classes

Languages like Haskell have popularized the notion of type classes. In its simplest form, a type class is a set of types with a common interface: all types in that class must provide the functions in the interface. Isabelle offers a similar concept, called *axiomatic type classes*. An axiomatic type classes is something like a type class with axioms, i.e., an axiomatic specification of a class of types, thus a type 'a being in a class C (written 'a::C) must satisfy all axioms of C. Furthermore, type classes can be organized in a hierarchy. Thus there is the notion of a class D being a sub class of a class C, written D < C. This is the case if all axioms of C are also provable in D.

D < C

Isabelle/HOL already has a built-in type class ord that among others defines the <= symbol for orders. On top of ord we can introduce a type class reford which requires reflexivity for the order relation:

ord
<=
axclass

```
axclass reford < ord
  reford_refl : "x <= y"
```

For types being in the type class reford we now have an antisymmetric order and should be able to proof:

```
lemma "(x::'a::reford) <= x"
```

But for now, there are no concrete types in the type class reford.

8.1.2 Instances

instance

To bring life in our new type class reford we have to declare that a concrete type is an **instance** of our type class and we also have to define the meaning of <= over bool.

But first we prove that bool is an instance of the type class ord:

```
instance bool :: ord
  apply( intro_classes )
  done
```

intro_classes

Where intro_classes is a special method for doing "instance-proofs", i.e., every proof of a type being a instance of a type class should start with applying this method. Further, we define the meaning of our order <= over bool as implication (⟶):[1]

```
defs (overloaded)
  leq_bool_def : "p <= q ≡ p ⟶ q"
```

and prove that bool is a instance of the type class reford:

```
instance bool :: reford
   apply( intro_classes )
   apply(unfold leq_bool_def )
   apply( rule imp_refl )
   done
```

8.1.3 Using the Simplifier

The simplifier uses a "current simplifier set" available in a proof context. This can be modified in the ISAR-language by adding new rules (that must have the format the simplifier may process; i.e. it must be a higher-order pattern rule), deleting rules or by adding rules of a special format, e.g. splitter rules or congruence rules, which we will discuss in the future.

Examples for the syntax of the simplifier method are:

```
apply(simp add: A B C)
apply(simp_all del: B)
apply(simp only: A)
apply(simp addsplit: E)
apply(simp addcong: F)
```

[1]The (overloaded) keyword is used here because the syntax of <= is used in many different contexts and we "overload" it with our definition.

8.2 Exercises

8.2.1 Exercise 33

1. Define an axiomatic class "qorder" of quasi-orderings (these are structures with an ordering symbol op <= which are reflexive and transitive).

2. Define an axiomatic subclass " linqorder " of *linear quasi-orderings* which enjoy the additional property A <= B ∨B <= A

   Define the relation:

   A ~=~ B == A <= B ∧B <= A

   on it.

3. Show that linear quasi-orderings are equivalence relations and prove the following properties (min is inherited from class ord):

   ```
   lemma min_cong: "A ~=~ B ⟹min A B ~=~ B"
   lemma linear_order_CE [dest !]:
     "¬ (A::'a::linqorder) <= B ⟹B <= A"
   lemma min_com: "min (A::'a::linqorder) B ~=~ min B A"
   lemma min_sym "min (A::'a::linqorder) B ~=~ min B A"
   lemma le_split:
     "(A::'a::linqorder) <= B ⟹¬(B <= A) ∨(A ~=~ B)"
   lemma quasi_refl: "A ~=~ A"
   lemma quasi_sym: "A ~=~ B ⟹B ~=~ A"
   lemma "⟦ A ~=~ B; B ~=~ C ⟧⟹A ~=~ C"
   ```

4. Define the ordering op <= over pairs by conjoining the ordering on components of the pairs and prove

   ```
   lemma "(a::('a::qorder * 'b::qorder)) ~=~ b ⟹b ~=~ a"
   ```

Hint: Lookup the definition of the axiomatic class order in the HOL theory (`http://isabelle.in.tum.de/library/HOL/HOL.html`) and modify it!

Hint: Use simp, fast, auto!

Answer to Exercise 33

1. Defining a quasi-order:

```
axclass qorder < ord
  qorder_refl [iff]: "x ≤ x"
  qorder_trans [trans]: "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
```

2. Defining a linear quasi-order:

```
axclass linqorder < qorder
    linqorder_linear : " A ≤ B ∨ B ≤ A"
```

3. Defining an equivalence relation:

```
constdefs
  "~=~" :: "['a::qorder, 'a] ⇒ bool"                  (infixl 50)
  "A ~=~ B ≡ A ≤ B ∧ B ≤ A"
```

4. Proving basic properties:

```
lemma min_cong: "A ~=~ B ⟹ min A B ~=~ B"
  apply(unfold "op ~=~_def" min_def)
  apply(simp)
  done

lemma linear_order_CE [dest !]:
  " ¬ (A::'a::linqorder) ≤ B ⟹ B ≤ A"
   by (insert linqorder_linear , auto)

 lemma min_com: "min (A::'a::linqorder) B ~=~ min B A"
   apply(unfold "op ~=~_def" min_def)
   apply(auto)
   done

lemma linear_order_simp:
  " (¬ (A::'a::linqorder) ≤ B) ⟶ (B ≤ A)"
  apply (auto)
  done

lemma min_sym: "min (A::'a::linqorder) B ~=~ min B A"
  apply(unfold "op ~=~_def" min_def)
```

```
  apply(simp)
  apply(blast)
  done

lemma le_split: "(A::'a::linqorder) ≤B ⟹¬(B ≤A) ∨(A ~=~ B)"
  apply(unfold "op ~=~_def" min_def)
  apply (auto)
  done

lemma quasi_refl: "A ~=~ A"
  apply(unfold "op ~=~_def")
  apply(auto)
  done

lemma quasi_sym: "A ~=~ B ⟹B ~=~ A"
  apply(unfold "op ~=~_def")
  apply(auto)
  done

lemma "⟦ A ~=~ B; B ~=~ C ⟧⟹A ~=~ C"
  apply(unfold "op ~=~_def")
  apply(auto)
  apply(erule qorder_trans ,simp)
  apply(erule qorder_trans ,simp)
  done
```

5. Extending the quasi-order to pairs:

```
instance ∗ :: (ord, ord) ord
by( intro_classes )

defs (overloaded)
   leq_prod_def : "p <= q ≡ fst p <= fst q ∧ snd p <= snd q"
instance ∗ :: (qorder, qorder) qorder
  apply( intro_classes )
  apply(unfold leq_prod_def "op ~=~_def")
  apply(auto)
  apply(erule qorder_trans )
  apply(assumption)
  apply(erule qorder_trans )
```

apply(*assumption*)
done

```
lemma "(a::('a::qorder * 'b::qorder)) ~=~ b ⟹b ~=~ a"
  apply ( rule quasi_sym)
  apply (assumption)
  done
```

8.2.2 Exercise 34

1. Prove the following set-theoretic properties only using the simplifier (not fast, not blast, not auto):

   A ∪ (B ∪ A) ⊆ A ∪ B
   A = D ⟹ A ∪ (C ∪ B) ∪ D = C ∪ B ∪ A
   F = B ⟹ A ∩ (B ∪ C) = (C ∩ A) ∪ (B ∩ A ∩ F)

2. Prove the following set-theoretic properties with methods of your choice:

   Domain r = UNIV ⟹ Id ⊆ r\={} O r
   Domain r ≠ UNIV ⟹ ∃x. (x, x) ∉ r\={} O r
   ∃ x∈ A. X ⊆ B x ⟹ X ⊆ UNION A B

Hint: For the first task, set up the simplifier such that it computes ACI normal forms.

Answer to Exercise 34

1. Using the simplifier:

```
lemma ex34_1_1: "A ∪(B ∪A) ⊆A ∪B"
  apply(simp add: Un_ac)
  done

lemma ex34_1_2: "A = D ⟹A ∪(C ∪B) ∪D = C ∪B ∪A"
  apply(simp add: Un_ac)
  done

lemma ex34_1_3: "F = B ⟹A ∩(B ∪C) = C ∩A ∪B ∩A ∩F"
  apply(simp add: Un_ac  Un_Int_distrib2 )
  apply(auto)
  done
```

2. Using full automation:

```
lemma ex34_4: "Domain r = UNIV ⟹Id ⊆r⁻¹ O r"
  apply(auto)
  done

lemma ex34_5: "Domain r ≠UNIV ⟹∃x. (x, x) ∉r⁻¹ O r"
  apply(auto)
  done

lemma ex34_6: "∃x∈A. X ⊆B x ⟹X ⊆UNION A B"
  apply(auto)
  done
```

8.2.3 Exercise 35

We define a (tiny) fragment of the specification language Z.[2] Begin by defining the type of relations as sets of products using the type synonym:

```
types    ('a,'b) "<=>" = "('a*'b) set"    (infixr 20)
```

Define the Z constructs notational equivalent:

```
syntax
  dom     ::"('a <=> 'b) => 'a set"
  ran     ::"('a <=> 'b) => 'b set"

translations
  "dom r" == "Domain r"
  "ran r" == "Range r"
```

1. Define the following operators over sets A and B:

A <−−−> B	relation
A −|−> B	partial function
A −−−> B	total function
A >−|−> B	partial injection
A >−−> B	total injection
A −|−>> B	partial surjection
A >−−>> B	bijection

[2]You can find more information about Z on the "Z Notation Website": `http://archive.comlab.ox.ac.uk/z.html`.

2. Define the operator *override* A (+) B that takes two relations and "combines" them as follows:

   a) any (x,y):A is in the override, iff x ~: dom B,

   b) any (x,y):B is in the override, iff x ~: dom B.

3. Prove:

```
f : A -|-> B ⟹ f : A <--> B
f : A ---> B ⟹ f : A -|-> B
f : A >-|-> B ⟹ f : A -|-> B
f : A >--> B ⟹ f : A -|->> B ⟹ f : A >-->> B
A (+) A = A
(A (+) B) (+) C = A (+) (B (+) C)
```

Hint: Use simp, fast, auto as you like.

Hint: It might be useful to define a concept like "domain restriction" S <: A (cutting down a relation A by erasing all pairs, whose first component is in a given set S).

Answer to Exercise 35

1. Defining basic Z operators:

```
constdefs
 rel      :: "['a set, 'b set] ⇒ ('a <=> 'b) set" ("_ <--> _"[54,53] 53)
 "A <--> B ≡ Pow {(x, y). x ∈ A ∧ y ∈ B}"

consts
 dom_res:: "['a set , 'a <=> 'b] => 'a <=> 'b" ("_ <: _"  [71,70] 70)
 ran_res :: "['a <=> 'b, 'bset]   => 'a <=> 'b" ("_ :> _"  [65,66] 65)
 dom_sub:: "['a set , 'a <=> 'b] => 'a <=> 'b" ("_ <-: _"[71,70] 70)
 ran_sub :: "['a <=> 'b, 'b set]  => 'a <=> 'b" ("_ :-> _" [65,66] 65)

 pfun   :: "['a set ,'b set] => ('a <=> 'b) set" ("_ -|-> _" [54,53] 53)
 tfun   :: "['a set ,'b set] => ('a <=> 'b) set" ("_ ---> _" [54,53] 53)
 pinj   :: "['a set ,'b set] => ('a <=> 'b) set" ("_ >-|-> _"[54,53] 53)
 tinj   :: "['a set ,'b set] => ('a <=> 'b) set" ("_ >--> _" [54,53] 53)
 psurj  :: "['a set ,'b set] => ('a <=> 'b) set" ("_ -|->> _"[54,53] 53)
 tsurj  :: "['a set ,'b set] => ('a <=> 'b) set" ("_ -->> _" [54,53] 53)
```

```
  bije      ::" ['a set ,'b set] => ('a <=> 'b) set" ("_ >-->> _"[54,53] 53)

defs
  pfun_def:     "S -|-> R ≡{f. f ∈S <--> R
                          ∧ (∀x y1 y2. (x,y1)∈f
                          ∧ (x,y2)∈f ⟶y1 = y2)}"

  dom_res_def: "S <: R     ≡{(x, y). (x, y) ∈R ∧x ∈S}"
  ran_res_def : "R :> S    ≡{(x, y). (x, y) ∈R ∧y ∈S}"
  dom_sub_def: "S <-: R   ≡{(x, y). (x, y) ∈R ∧x ∉S}"
  ran_sub_def: "R :-> S   ≡{(x, y). (x, y) ∈R ∧y ∉S}"


defs
  tfun_def:     "S ---> R ≡{s. s ∈S -|-> R ∧dom s = S}"
  pinj_def :    "S >-|-> R ≡{s. s ∈S -|-> R
                           ∧ (∀x1 x2 y. (x1,y)∈s
                           ∧ (x2,y)∈s ⟶x1 = x2)}"
  tinj_def :    "S >--> R ≡(S >-|-> R) ∩(S ---> R)"
  tsurj_def :   "S -->> R ≡(S -|->> R) ∩(S ---> R)"
  psurj_def :   "S -|->> R ≡{s. s : S -|-> R ∧ran s = R}"
  bije_def :    "S >-->> R ≡((S -->> R) ∩(S >--> R))"
```

2. Defining overwrite:

```
consts
  override ::" ['a <=> 'b, 'a <=> 'b] => ('a <=> 'b)"("_ '(+') _"
                                                    [55,56] 55)
defs
  override_def :"S (+) R   ≡(dom R <-: S) ∪R"
```

3. Proving some properties:

```
lemma "f∈A -|-> B ⟹f∈A <--> B"
  apply(auto simp: pfun_def)
  done

lemma "f∈A ---> B ⟹f∈A -|-> B"
  apply(auto simp: tfun_def pfun_def)
  done
```

```
lemma "f∈A >-|-> B ⟹f∈A -|-> B"
  apply(auto simp: pinj_def pfun_def)
  done

lemma "f∈(A >--> B) ⟹f∈(A -|->> B) ⟹f ∈(A >-->> B)"
  apply(auto simp: bije_def tinj_def tsurj_def )
  done

lemma "A (+) A = A"
  apply(unfold override_def )
  apply(auto simp: override_def dom_sub_def)
  done

lemma "(A (+) B) (+) C = A (+) (B (+) C)"
  apply(auto simp: override_def dom_sub_def)
  done
```

9 HOL: Inductive Data Types

In this exercise, we will study the concept of the least fix-point operator lfp , its main theorems knaster_tarski and lfp_induct and its major application: providing semantics for inductive definitions. lfp

The importance of the concept of inductive definition will be revealed by applying it in three examples, ranging from closures, finite sets to natural numbers.

9.1 More on Isabelle/HOL

9.1.1 Inductive Definitions

The general syntactic scheme of an inductive definition is: **inductive**

inductive "expr"
  intros
  thmname_1: "H_1 ∈ expr"
  ...
  thmname_m: "⟦ Cond_1(expr); ...; Cond_n(expr)⟧ ⟹ H_m ∈ expr"

where expr must be a set of the form C var_1 ...var_k and where C is a previously declared, but not yet defined constant, and the list of variables var_i may be empty. After the keyword **intros**, introduction rules for the inductive set may be inserted, either with assumptions or not (both forms can be arbitrarily mixed). The conditions Cond_i may depend on expr or not.

Isabelle will process such statements and compile it to

1. a constant definition for C which can be referenced by C.**defs** C.**defs**
2. proofs for the introduction rules in the form given in the inductive statement; the theorems can be referenced by their given name thmname_i, and
3. proofs for the induction rules which can be referenced by C.induct C.induct

Note that introducing theorems via the **declare** statement (see the ISAR Ref- **declare** erence Manual[1]) allows to insert such rules once and for all into the appropriate "slots" of the proof engine; there are more syntactic variants in the inductive statement that have the same effect.

9.1.2 Constant Specifications

constant specification

There is an alternative conservative extension scheme supported by Isabelle, namely the *constant specification*. In contrast to the constant definition used so far, a "fresh" constant c may be specified by a syntacticly unlimited predicate P in an axiom $P\,x$. Of course, this axiom must be justified by the proof of the semantic side-condition $\exists x. P\,x$.

specification

The overall syntactic scheme of a constant specification in the ISAR language is:

```
specification (C)
  thmname: "P C"
  ...
  done
```

where C is a previously declared, but not yet defined constant, P a characterizing predicate that can be referenced by thmname, followed by a proof for the side-condition.

9.2 Exercises

9.2.1 Exercise 36

Prove the Knaster-Tarski theorem

$$mono\ f \Longrightarrow lfp\ f = f(lfp\ f)$$

using the presentation given in the lecture "HOL: Fixpoints", i.e., first prove the claims 1–4. Use whatever proof methods you like, but you should no use any theorem from the HOL library.

Answer to Exercise 36

```
lemma claim1: "f A ⊆A ⟹lfp  f ⊆A"
  apply(auto simp:  lfp_def )
  done
```

[1] http://isabelle.in.tum.de/dist/Isabelle2004/doc/isar-ref.pdf

```
lemma claim2: "(∀x. f x ⊆x ⟶ A ⊆x) ⟹ A ⊆ lfp f"
  apply(auto simp: lfp_def )
  done

lemma claim3: "mono f ⟹f(lfp f) ⊆ lfp f"
  apply( rule claim2)
  apply( rule allI )
  apply( rule impI)
  apply( rule order_trans )
  prefer 2
  apply(assumption)
  apply( erule monoD)
  apply(auto dest: claim1)
  done

lemma claim3': "mono f ⟹f( lfp f) ⊆ lfp f"
  apply( rule claim2)
  apply( rule allI , rule impI)
  apply( rule_tac y="f x" in order_trans )
  apply(auto elim !: monoD dest: claim1)
  done

lemma claim4: "mono f ⟹ lfp f ⊆f (lfp f)"
  apply( rule claim1)
  apply( frule monoD)
  prefer 2
  apply(assumption)
  apply( erule claim3)
  done

lemma claim4': "mono f ⟹ lfp f ⊆f (lfp f)"
  apply( rule claim1)
  apply( frule_tac A="f (lfp f)" and B="lfp f" in monoD)
  apply(auto elim !: claim3)
  done

lemma KnasterTarski: "mono f ⟹ lfp f = f( lfp f)"
  apply(auto dest: claim4 claim3)
  done
```

9.2.2 Exercise 37

1. Define inductively the function "Fin:: 'a set ⇒ 'a set set" that produces the set of all finite subsets.

2. Prove the following properties over set of all finite subsets:

   a) **lemma** "{1,2}∈Fin{1,2,3}"
   b) **lemma** "⟦a∈Fin A; b∈Fin A⟧⟹(a∪b) ∈ Fin A"
   c) **lemma** "⟦(A ∈Fin X) ∨ (A ∈ Fin Y)⟧ ⟹A ∈ Fin (X ∪Y)"
   d) **lemma** finite_InI : "⟦ b∈ Fin A⟧ ⟹(a∩b) ∈ Fin A"
   e) **lemma** "⟦A ∈Fin X⟧⟹Pow(A)∈ Pow(Fin X)"

Remark: The elements 1, 2, etc. do not imply that we have already numbers; they are constants in syntactic classes predefined in the library. As a result, Fin{1,2,3} has the type ('a::{one,zero,number})set and not nat set.

Answer to Exercise 37

1. Defining Fin:

```
consts Fin :: "'a set ⇒ 'a set set "

inductive "Fin(A)"
  intros
  emptyI [simp, intro !]:  "{} ∈ Fin(A)"
   insertI  [simp, intro !]:  "⟦ a∈A; b∈Fin(A) ⟧⟹ insert a b ∈ Fin(A)"
```

2. Proving properties over Fin:

```
lemma "{1,2} ∈Fin {0,1,2}"
  apply(simp)
  done

lemma ex37_1aux : "mono (λS. {x. x = {} ∨(∃a b. x = insert a b
                             ∧ a ∈ {0::('a::{zero,one,number}), 1::'a, 2::'a}
                             ∧ b ∈ S)})"
  apply(auto intro !: HOL.monoI)
  done
```

```
lemma "{1,2} ∈Fin {0,1,2}"
  apply(unfold Fin.defs)
  apply(subst lfp_unfold [OF ex37_1aux])
  apply(subst lfp_unfold [OF ex37_1aux])
  apply(subst lfp_unfold [OF ex37_1aux])
  apply(auto)
  done

lemma finite_UnI: "⟦ a∈Fin A; b∈Fin A⟧ ⟹(a∪b) ∈Fin A"
  apply (erule Fin.induct)
  apply (auto)
  done

lemma "⟦(A ∈Fin X) ∨ (A ∈Fin Y)⟧ ⟹A ∈Fin (X ∪Y)"
   apply(auto intro: Fin.induct)
   done

lemma finite_IntI : "⟦b∈Fin A⟧ ⟹(a∩b) ∈Fin A"
  apply(erule Fin.induct)
  apply(simp)
  apply(subst Int_insert_right )
  apply(auto)
  done

lemma "⟦A ∈Fin X⟧ ⟹Pow(A) ∈Pow(Fin X)"
  apply(erule Fin.induct)
  apply(simp)
  apply(subst Pow_insert)
  apply(auto)
  done
```

9.2.3 Exercise 38

1. Define the concept of a reflexive transitive closure as an inductive definition over the constant

```
consts
  rtc :: "('a × 'a) set ⇒ ('a × 'a) set" ("(_^**)" [1000] 999)
```

2. Prove the following properties, using the derived induction scheme (The last two are optional.):

   a) **lemma** rtc: "⋀p. p ∈ r ⇒ p ∈ r^**"

   b) **lemma** rtc_induct_pointwise:
      assumes a: "(a:: 'a, b) ∈ r^**"
      assumes base: "P a"
      assumes step: "⋀ y z. ⟦(a, y) ∈ r^**; (y, z) ∈ r; P y⟧ ⟹ P z"
      shows "P b"

   c) **lemma** ctr_trans: "⟦ (a,b)∈ r^**;(b,c)∈ r^** ⟧ ⟹ (a,c)∈ r^**"

   d) **lemma** rtc_is_closure : "(r^**)^** = r^**"

   e) **lemma** rtc_un_distr: "(R^** ∪S^**)^** = (R ∪S)^**"

   f) **lemma** rtc_un_distr: "R^** O R^** = R^**"

Hints:

1. Prove the lemmas in the given order.
2. You may unfold variables denoting pairs with the method: **apply**(simp only: split_tupled_all)
3. The crucial alternative induction scheme needs an additional assumption a = a ⟶P(b). You should add this assumption (using subgoal_tac) and prove it using the derived induction scheme with the instance P = λx y. x = a ⟶P y.

Answer to Exercise 38

1. Defining rtc:

```
consts
  rtc :: "('a × 'a) set ⇒ ('a × 'a) set" ("(_^**)" [1000] 999)

inductive "r^**"
  intros
    rtc_refl    : "(a, a) : r^**"
   rtc_compose : "⟦(a, b) : r^**; (b, c) : r⟧ ⟹ (a, c) : r^**"
```

2. Proving properties over rtc:

```
lemma rtc [intro]: "⋀ p. p ∈ r ==> p ∈r^**"
  apply(simp only:  split_tupled_all )
  apply(erule  rtc_refl  [THEN rtc_compose])
  done

lemma rtc_induct_pointwise :
  assumes a : "(a::'a, b) ∈ r^**"
  assumes base : "P a"
  assumes step : "⋀y z. ⟦(a, y) ∈ r^**; (y, z) ∈ r; P y⟧ ⟹ P z"
  shows "P b"
  apply(subgoal_tac "a = a ⟶P(b)")
  apply(blast)
  apply( rule_tac  P = "λ x y. x = a ⟶P y"  in  rtc . induct [OF a])
  apply(auto intro :  base  step)
  done


lemma ctr_trans : "⟦ (a,b)∈r^**;(b,c)∈r^** ⟧ ⟹ (a,c)∈r^**"
  apply( erule_tac  b = c in  rtc_induct_pointwise )
  apply(blast  intro !: rtc_compose)+
  done


lemma  rtc_is_closure  :  "(r^**)^** = r^** "
  apply(auto)
  apply( erule  rtc . induct)
  apply( rule  rtc_refl )
  apply(blast  intro :  ctr_trans )
  done

lemma rtc_un_distr: "(R^** ∪S^**)^** = (R ∪S)^**"
  oops

lemma rtc_un_distr: "R^** O R^** = R^**"
  oops
```

9.2.4 Exercise 39

State the axiom of infinity

```
axioms  infinity : "∃ f::ind ⇒ ind. inj f ∧ ¬ surj f"
```

and build a conservative theory extension deriving the core of the natural number theory, the Peano Axioms:

1. Declare the constants ZERO::ind and SUC::ind ⇒ind,
2. Use a constant specifications to specify ZERO and SUC appropriately, i.e., such that you can derive ZERO ≠SUC X and SUC X = SUC Y ⟹X = Y,
3. Define NAT as the inductive set built over ZERO and SUC
4. Show the "induction" theorem on NAT.

Answer to Exercise 39

```
axioms  infinity : "∃ f::ind ⇒ ind. inj f ∧ ¬ surj f"

consts
   ZERO :: ind
   SUC :: "ind ⇒ ind"


specification (SUC)
   SUC_charn: "inj SUC ∧¬ surj SUC"
   by ( rule infinity )

specification (ZERO)
   ZERO_charn: "ZERO ≠SUC X"
   by ( insert SUC_charn, auto simp: surj_def )

lemma SUC_inj : "SUC X = SUC Y ⟹X = Y"
   by( insert SUC_charn, auto elim:injD)

consts NAT :: "ind set"

inductive "NAT"
  intros
  ZERO_I: "ZERO ∈NAT"
  SUC_I : "⟦ x∈NAT ⟧⟹SUC x ∈NAT"

lemmas "induction" = NAT.induct
```

10 HOL: Well-founded and Primitive Recursion

In this exercise, we will deepen our knowledge on well-founded orderings and induction as well as its applications in form of recursive definitions.

10.1 Recursive Definitions

10.1.1 Primitive recursion

Isabelle provides a syntactic front-end for defining an important subclass of well-founded recursions, namely *primitive recursive* functions, e.g.: *primitive recursive*

primrec **primrec**

```
primrec
  add_0:    "0 + n = n"
  add_Suc:  "Suc m + n = Suc (m + n)"

primrec
  diff_0 :   "m - 0 = m"
  diff_Suc : "m - Suc n = (case m - n of
                               0    => 0
                          |  Suc k => k)"
```

The general form of a primitive recursive definitions in Isabelle is:

primrec

$name_1$: "*rule*"

$\vdots$

$name_n$: "*rule*"

where *rule* are *reduction rules* (as usual, the names $name_1 \dots name_n$ are optional). The reduction rules specify one or more equations of the form *reduction rules*

$$f\ x_1\ \dots\ x_n(C\ y_1\ \dots\ y_n)\ z_1\ \dots\ z_n = r$$

such that C is a constructor of the datatype (e.g. Suc in our first example), r contains only free variables on the left-hand side, and all recursive calls in r are of the form $f\ \dots\ y_i\ \dots$ for some i.

10.1.2 General Recursive Definitions

recdef Isabelle also offers a way for declaring functions using general well-founded recursion: **recdef**. Using **recdef**, you can declare functions involving nested recursion and pattern-matching, e.g. we can define the Fibonacci function:

```
consts fib  :: "nat ⇒ nat"
recdef fib " less_than"
       " fib 0 = 0"
       " fib 1 = 1"
       " fib (Suc(Suc x)) = (fib x + fib (Suc x))"
```

where les_than is the "less than" on the natural numbers.

The general form of a recursive definitions in Isabelle is:

primrec *function rule*
  congs "*rules*"
  simpset "*rules*"
$name_1$: "*rule*"
⋮
$name_n$: "*rule*"

where *function* is the functions name and *rule* a HOL expression for the well-founded termination relation (Isabelle provides several built-in relations such as less_than or measure). With the to *optional* arguments congs and simpset one can influence the set of congurences rules and the simpset used during the termination proof. Finally, the *rule*s are specifing the "computational" recursive equations.

10.2 Exercises

10.2.1 Exercise 40

Prove the following consequences of well-founded orderings:

1. a well-founded ordering is not symmetric:

   lemma wf_not_sym: "wf(r) ⟹∀a x. (a,x)∈r ⟶ (x,a)∉r"

2. a well-founded ordering contains minimal elements:

   lemma wf_minimal: "wf r ⟹∃x. ∀y. (y,x) ∉ r^+"

3. a subrelation of a well-founded ordering is well-founded:

```
lemma wf_subrel: "wf(p) ⟹ ∀ r. r ⊆ p ⟶ (∃ x. ∀ y. (y,x) ∉ r^+)"
```

4. a well-founded ordering satisfies characterization (1):

```
lemma wf_eq_minimal2:
"wf(p) = (∀ r. (r≠{} ∧r ⊆p) ⟶ (∃ x ∈ Domain r. (∀ y. (y,x) ∉ r)))"
```

Hint: Look up the various theorems about wellfounded orderings that Isabelle provides (wf_induct, wf_empty, wf_subset, wf_not_sym, wf_not_refl , wf_trancl , wf_acyclic , and wfrec_def) and use them as you like.

Answer to Exercise 40

1. a well-founded ordering is not symmetric:

```
lemma wf_not_sym: "wf(r) ⟹∀a x. (a,x)∈r ⟶ (x,a)∉r"
  apply(rule allI )
  apply( rule_tac a = "a" in wf_induct)
  apply(assumption)
  apply( blast )
  done
```

2. a well-founded ordering contains minimal elements:

```
lemma wf_minimal: "wf r ⟹∃x. ∀ y. (y,x) ∉ r^+"
  apply( rule_tac r = "r^+" in wf_induct)
  apply( erule wf_trancl )
  apply( rule disjE )
  prefer 2
  apply(assumption)
  apply( rule_tac [2] FalseE)
  apply(auto)
  done
```

3. a subrelation of a well-founded ordering is well-founded:

```
lemma wf_subrel: "wf(p) ⟹ ∀ r. r ⊆p ⟶ (∃ x. ∀ y. (y,x) ∉ r^+)"
  apply( rule allI )
  apply( rule impI)
  apply( rule wf_minimal)
  apply( erule wf_subset )
  apply(assumption)
  done
```

4. a well-founded ordering satisfies characterization (1):

```
lemma wf_eq_minimal2:
  "wf(p) = (∀r. (r≠{} ∧r⊆p) ⟶(∃x∈Domain r. (∀y. (y,x)∉r)))"
  apply(subst wf_eq_minimal)
  apply(unfold Domain_def)
  apply(auto)
  apply( erule_tac x = " (Domain r) Un (Range r) " in allE )
  prefer 2
  apply( erule_tac x = "p Int { (x,y) .x ∈Q}" in allE)
  apply(auto)
  done
```

10.2.2 Exercise 41

1. Define a the recursor iter f n in terms of the well-founded recursor wfrec and the theory of the natural numbers. Derive from your definition the properties:

```
lemma iter_0 : " iter 0 g = (λ x. x)"
lemma iter_Suc : " iter (Suc n) g = g ∘ ( iter n g)"
```

2. Define the addition add, the multiplication mult, the exponentiation exp and the sumup operation sumup (sumup 3 = 1 + 2 + 3) on natural numbers.

   Use in at least two definitions the iter -recursor and derive the usual computational equations; in the other cases, you may use a **primrec** construct.

Answer to Exercise 41

1. Defining iter :

```
constdefs
  iter :: "[nat, 'a ⇒ 'a] ⇒ 'a ⇒ 'a"
  " iter n g ≡(wfrec pred_nat (λ f x. if x = 0 then (λ x. x)
                                     else g ∘ (f(THE y. x = Suc y)))) n"

lemma iter_0 [simp]: " iter 0 g = (λ x. x)"
  apply(auto simp: iter_def )
  apply(simp add: wfrec [OF wf_pred_nat])
```

```
  done

lemma iter_Suc [simp] : "iter (Suc n) g = g ∘ (iter n g)"
   apply(auto simp: iter_def )
   apply(simp add: wfrec [OF wf_pred_nat])
   apply auto
   apply( rule_tac f = "(λx. g ∘ x)" in arg_cong)
   apply( rule trans)
   apply( rule cut_apply)
   apply(simp only: pred_nat_def)
   apply auto
   apply(simp add: wfrec [OF wf_pred_nat])
   apply( rule_tac f = "(λx. g ∘ x)" in arg_cong)
   apply( rule trans)
   apply( rule cut_apply)
   apply(simp only: pred_nat_def)
   apply auto
   apply(simp add: wfrec [OF wf_pred_nat])
   done

lemma iter_0' [simp] : " iter 0 g x = x"
   by (simp)


lemma iter_Suc' [simp] : " iter (Suc n) g x = g ( iter n g x)"
   by (simp)
```

2. Defining operations on natural numbers:

```
constdefs
  add :: "[nat, nat] ⇒ nat"
  "add n m ≡(iter n Suc) m"

lemma add_0 : "add 0 x = x"
  by (simp add: add_def)

lemma add_Suc : "add (Suc x) y = Suc (add x y)"
  by (simp add: add_def)
```

```
constdefs
  mult :: "[nat, nat] ⇒ nat"
  "mult n m ≡(iter n (λ x. add m x)) 0"

lemma mult_0 : "mult 0 x = 0"
  by (simp add: mult_def)

lemma add_Suc : "mult (Suc x) y = add y (mult x y)"
  by (simp add: mult_def)

consts exp :: "[nat,nat] ⇒ nat"

primrec
  exp_0 :    "exp k 0 = 1"
  exp_Suc : "exp k (Suc x) = mult k (exp k x)"

consts sumup :: "nat ⇒ nat"

primrec
  sumup_0 :   "sumup 0 = 0"
  sumup_Suc : "sumup (Suc x) = add (Suc x) (sumup x) "
```

10.2.3 Exercise 42 — "The approximation theorem of lfp"

In lecture "HOL: Fixpoints" we have seen the theorem:

$$(\forall S.\ f(\bigcup S) = \bigcup(f\ `\ S)) \Longrightarrow \bigcup_{n \in N} f^n(\emptyset) = lfp\ f$$

i.e. under a certain condition, a fix-point can be seen as a limit of an approximation process. This condition is also called *continuity of f*. Under an obvious alternative constraint, namely that the fix-point must be reachable after finitely many steps, this principle is of practical importance, for example in data-flow analysis algorithms (such as the Java Byte-code Verifier).

Prove one of the following versions of the approximation theorem:

1. **lemma** lfp_approximable_if_finite :
   ⟦mono f; ∃ m. f (iter m f {}) = (iter m f {})⟧
   ⟹ (UN n:UNIV. (iter n f {})) = lfp f

2. **lemma** lfp_approximable_if_cont :
   ⟦(⋀S. f (Union S) = Union (f ‘ S))⟧
   ⟹ (UN n:UNIV. (iter n f {})) = lfp f

For the first option, we suggest the following intermediate lemmas:

1. mono f ⟹(UN n:UNIV . (iter n f {})) ≤ lfp f
2. ⟦mono f; ∃ m. f (iter m f {}) = (iter m f {})⟧
   ⟹ lfp f ≤(UN n:UNIV. (iter n f {}))

For the second option, we suggest the following milestones:

1. mono f ⟹(UN n:UNIV . (iter n f {})) ≤ lfp f
2. (∀ S. f (Union S) = Union (f ‘ S)) ⟹ mono f
3. (UN n:UNIV. iter (Suc n) f {}) = (UN n:{m. 0 < m}. (iter n f {}))
4. (UN n:UNIV. g (n::nat)) = (UN n:{m. 0 < m}. (g n))Un (g 0)
5. (∀ S. f (⋃S)= ⋃f ‘S)
   ⟹ f ($\bigcup_n$ iter n f {}) = ($\bigcup_n$ iter n f {})
6. (∀ S. f (Union S) = Union (f ‘ S))
   ⟹ f (UN n:UNIV. (iter n f {})) = (UN n:UNIV. f (iter n f {}))
7. ⋀S. f (Union S) = Union (f ‘ S))⟹ lfp f ≤(UN n:UNIV. (iter n f {})
)

Hint: Look up the various theorems about the inclusion operation that Isabelle provides (rev_subsetD, lfp_unfold , monoD, Un_upper1, Un_absorb1, image_Collect) and use them as you like.

Answer to Exercise 42

1. Option (1):

```
(* Option 1: the finite case *)
lemma union_below_lfp_1:
  "mono f ⟹(UN n:UNIV . (iter n f {})) ≤ lfp f"
  apply(auto)
  apply(erule rev_subsetD)
  apply(induct_tac "n")
```

```
    apply(simp_all)
    apply(subst lfp_unfold)
    apply(simp)
    apply(erule_tac monoD)
    apply(simp)
    done

  lemma lfp_below_union_if_finite_2 :
    "⟦mono f; ∃ m. f (iter m f {}) = (iter m f {})⟧
     ⟹ lfp f <= (UN n:UNIV. (iter n f {}))"
    apply(auto)
    apply(rule_tac x = "m" in exI)
    apply(erule rev_subsetD)
    apply(rule lfp_lowerbound)
    apply(auto)
    done

  lemma lfp_approximable_if_finte :
     "⟦mono f; ∃ m. f (iter m f {}) = (iter m f {})⟧
     ⟹ (UN n:UNIV. (iter n f {})) = lfp f"
    by(auto elim!: union_below_lfp_1 lfp_below_union_if_finite_2 )
```

2. Option (2):

```
  (* Option 2: continuation *)
  lemma union_below_lfp_1:
    "mono f ⟹(UN n:UNIV . (iter n f {})) ≤ lfp f"
    apply(auto)
    apply(erule rev_subsetD)
    apply(induct_tac "n")
    apply(simp_all)
    apply(subst lfp_unfold)
    apply(auto elim!: monoD)
    done

  lemma distr_implies_mono_2:
    "(∀S. f (Union S) = Union (f ` S)) ⟹ mono f"
    apply(rule monoI)
    apply(erule_tac x = "{A,B}" in allE)
    apply(auto simp:Un_eq_Union [symmetric] Un_upper1 Un_absorb1 image_Collect)
```

```
  done

lemma shift_successor_3:
  "(UN n:UNIV. iter (Suc n) f {}) = (UN n:{m. 0 < m}. (iter n f {}))"
  apply(auto)
  apply( rule_tac x = "n - 1" in exI)
  apply( rule_tac t = "f ( iter  (n - 1) f {}) " in subst)
  apply( rule  iter_Suc ')
  apply(auto)
  done

lemma split_universal_union_4 :
  "(UN n:UNIV. g (n::nat)) = (UN n:{m. 0 < m}. (g n)) Un (g 0)"
  by(auto, case_tac "n=0",auto)

lemma cont_f_distributes_over_UN_5:
  "(∀ S. f (Union S) = Union (f ` S))
  ⟹ f (UN n:UNIV. (iter n f {})) = (UN n:UNIV. f (iter n f {}))"
  apply( rule_tac a = "f (Union {x . (∃ n. x = iter n f {})})" and
                 b = " (Union {f x | x. (∃ n. x = iter n f {}) }) "
        in box_equals)
  apply(subst image_Collect [symmetric])
  prefer 2
  apply( rule_tac f = "f" in arg_cong)
  apply(auto)
  done

lemma union_is_fixpoint_6:
" (∀ S. f (⋃S) = ⋃f `S)
  ⟹ f (⋃n iter n f {}) = (⋃n iter n f {})"
 apply( rule  trans)
 apply( erule  cont_f_distributes_over_UN_5 )
 apply(simp  only: iter_Suc '  [symmetric])
 apply( rule  trans)
 apply( rule  Un_empty_right [symmetric])
 apply( rule_tac P="%x. ?X Un x = ?Y" in subst)
 apply( rule_tac g="f" in iter_0 ')
 apply(simp  only:  shift_successor_3   split_universal_union_4 )
 done
```

```
lemma lfp_below_union_7:
  "(ALL S. f (Union S) = Union (f ` S))
   ==> lfp f <= (UN n:UNIV. (iter n f {}))"
  apply (rule lfp_lowerbound)
  apply (simp add: union_is_fixpoint_6 )
  done

lemma lfp_approximable_if_cont:
  "[|(!! S. f (Union S) = Union (f ` S))|]
   ==> (UN n:UNIV. (iter n f {})) = lfp f"
  apply(rule subset_antisym)
  apply(rule union_below_lfp_1 )
  apply(rule distr_implies_mono_2 ,simp)
  apply(rule lfp_below_union_7 ,simp)
  done
```

11 HOL: Hoare Logic

With this exercise, we turn now to applications of Isabelle/HOL in the field of (theoretical) computer science. We will reuse an existing encoding of an imperative toy-language for verifications of imperative programs. From the theorem proving side, we will introduce into structured proofs with ISAR.

11.1 More on Isabelle: Some ISAR Features

11.1.1 Structured Proofs with ISAR: An Introduction

Interactive theorem proving (as we introduced it in the course and as we — the authors — still believe is easier to understand comprehensively) has been dominated by a model of proof that goes back to the LCF system: a proof is a sequence of commands that manipulate an implicit proof state.

This model is reflected in the syntactic structure:

(**lemma** | theorem) [*name* :] <proposition> <**proof**>

where <**proof**> has is a sequence of **apply**(<method>) commands followed by **done** or just **by**(method).

Tactic-style proofs had been criticized for being very distinct from mathematics-like texts, unstructured and hard to maintain. Therefore, ISAR has been conceived to allow a more declarative proof-style that is claimed to be closer to mathematical texts (the reader may browse through the meanwhile quite rich corpus of structured proofs in the library in order to decide if this goal has really been achieved).

Structured proofs were introduced by a new alternative in the syntactic category <**proof**> which introduces a block structure:

<**proof**> ::= ...
| **proof** [<method> | −] <**proof**> {<statement>} **qed**

Here, a <statement> has the form:

<statement> ::= **fix** {name}
| **assume** [<fact>:] <propositions>
| [**from** {<fact>} | **this**} (**show** | **have**) <propositions>

fix
assume
from
this
show
have
note
let

```
   <proof>
 | note <fact> = <fact> | this
 | let  <meta-var> = "term"
```

and can thus again contain sub-proofs.

In the following, we discuss <statement> in more detail. The <**fix**>-statement serves as abstract means to introduce meta-quantified variables in a local proof goal, the <**assume**>-statement is used (similarly to the <**assumes** >-statement on the top-level) to introduce local assumptions and the <**have**>-statement to introduce the conclusion of a *local subgoal* of a proof. Thus, within a proof, local subgoals can be stated and proven. With the <**note**>-statement, the previous proposition (referenced by <**this**>) can be bound to a name, and in a <**let**> statement, a meta-variable may be bound to a particular *term*; since this meta-variable may be used in subsequent propositions, this may be used to reduce the size of local propositions and substitutions drastically. In connection with a pattern-match construct possible in any:

```
<proposition> ::= "term" [({is "<string>"})]
```

(where in the string, meta-variables may be used that can also be used in propositions and substitutions later), a means for systematic abbreviations in proof texts is provided.

Note that with the **proof**-directive, the current proof state is implicitly bound to a particular meta-variable ?thesis. Consequently, in order to conclude a sub-proof successfully, a proof will typically have the form:

```
proof -
  assume "the-assm"
  have   "concl" by (...)
  note   A = this
  assume "the-other-asm"
  have   "the-other-concl" by(...)
  note   B = this
  show   ?thesis
         <main proof>
```

Note that the – symbol stands for “do nothing”; if omitted, the default method is application of certain introduction rules controlled by the context.

Obviously, ISAR has been reduced to a kind of core-language here; a large number of abbreviations and syntactic variations exist. For example, there is an implicit fact management (pretty much inspired by PEARL) that makes most **note**-statements superfluous. We will describe some of these variations in subsequent exercise sheets.

11.2 Exercises

This exercise is based on IMP, in particular VC, which is not a IsabelleHOL built-in. You will need to extend Isabelle's search path such that Isabelle will be able to load the needed theory files at run-time. Therefore, start your theory file like:

```
ML {*
  add_path "$ISABELLE_HOME/src/HOL/IMP";
*}

theory HOL_Hoare = VC:
```

11.2.1 Exercise 43

Verify the program for computing the integer square root (from the lecture) in IMP from the lecture:

```
((   tm   :== (λs. 1));
 ((  sum  :== (λs. 1));
  ((i     :== (λs. 0));
    WHILE  (λs.  (s sum) <= (s a)) DO
        ((  i     :== (λs. (s  i ) + 1));
         ((tm    :== (λs. (s  tm) + 2));
          (sum  :== (λs. (s  tm) + (s sum)))))))
```

Verify this program using

1. the tactic-based method language
2. the structured ISAR language.

and compare the resulting proof scripts.

Hints:

- Use these given parenthesis's; the syntax setup of IMP is not really optimal this time !
- Do not forget to assume that the locations for i ,tm, sum and a are pairwise distinct.
- Use update_def in the simplifier set to handle updates.

Answer to Exercise 43

1. Using tactic-based method language:

```
constdefs
  squareroot :: "[loc,loc,loc,loc] => com"
  "squareroot tm sum i a ==
   ((  tm   :== (λs. 1));
    (( sum  :== (λs. 1));
     ((i    :== (λs. 0));
       WHILE (λs. (s sum) <= (s a)) DO
         (( i    :== (λs. (s i) + 1));
          ((tm   :== (λs. (s tm) + 2));
           (sum  :== (λs. (s tm) + (s sum)))))))
   )"

constdefs
  pre    :: assn
  "pre  == λx. True"
  post  :: "[loc,loc] ⇒ assn"
  "post a i == λs. (s i)*(s i)≤(s a) ∧ s a < (s i + 1)*(s i + 1)"

lemma sqrt_verify:
 assumes no_alias : "sum ≠i ∧ i ≠sum ∧tm ≠sum ∧
                     sum ≠tm ∧sum ≠a ∧ a ≠sum ∧
                     tm ≠i ∧ i ≠tm ∧tm ≠a ∧ a ≠tm ∧
                     a ≠ i ∧ i ≠a"
  shows "|- {pre} squareroot tm sum i a {post a i}"
  apply(unfold squareroot_def)
  apply(rule conseq)
  prefer 2
  apply(rule semi, rule ass)+
  apply(rule conseq)
  prefer 2
  apply( rule_tac P="λs. (s i + 1) * (s i + 1) = s sum ∧
    s tm = (2 * (s i) + 1) ∧
    (s i) * (s i) <= (s a) "
    in While)
  prefer 4
  apply(simp_all only: pre_def post_def)
  prefer 5
  apply(rule allI , rule impI, assumption)
  prefer 4
  apply(arith)
```

```
prefer 3
apply( rule allI , rule impI, assumption)
apply(simp add: update_def no_alias )
apply( rule_tac Q=" λs. (s i) ∗ (s i) = s sum ∧
                     s tm + 2 = (2 ∗ (s i) + 1) ∧
                     (s i) ∗ (s i) <= ((s a) + (2 ∗ (s i)) + 1) ∧
(s i) ∗ (s i) <= (s a)"
   in semi)
apply( rule conseq)
prefer 2
apply( rule ass)
prefer 2
apply( rule allI , rule impI, assumption)
apply(simp add: update_def no_alias )
apply( arith )
apply( rule_tac Q=" λs. s i ∗ s i = s sum ∧s tm = 2 ∗ s i + 1 ∧
s i ∗ s i ≤s a + 2 ∗ s i + 1
                   ∧ (s i) ∗ (s i) <= (s a)" in semi)
apply( rule conseq)
prefer 2
apply( rule ass)
prefer 2
apply( rule allI , rule impI, assumption)
apply(simp add:update_def no_alias )
apply( arith )
apply( rule conseq)
prefer 2
apply( rule ass)
prefer 2
apply( rule allI , rule impI, assumption)
apply(simp add: update_def no_alias )
apply( arith )
done
```

2. Using structured ISAR language:

```
lemma sqrt_verify_structured :
  assumes no_alias : "sum ≠i ∧ i ≠sum ∧tm ≠sum ∧
                     sum ≠tm ∧sum ≠a ∧ a ≠sum ∧
                     tm ≠i ∧ i ≠tm ∧tm ≠a ∧ a ≠tm ∧
```

```
                          a ≠ i  ∧ i  ≠ a"
  shows "|− {pre} squareroot  tm sum i a {post  a  i}"
proof −
   let ?inv  = "λs.(s  i  + 1) ∗ (s  i  + 1) = s sum
                        ∧ s  tm = (2 ∗ (s  i)  + 1)
                        ∧ (s  i)  ∗  (s  i)  <= (s a)"
   let ?post_tm = "λs. s tm = 1"
   have " |− {pre}  tm   :== (λs. 1) {?post_tm}"
     apply(unfold  pre_def)
     apply(rule  conseq)
     prefer  2
     apply( rule_tac  P="λs. s tm = 1"  in ass)
     apply(simp_all  add:  update_def  no_alias)
     done
   note init_tm=this
   let ?post_sum = "λs. s sum = 1 ∧s tm = 1"
   have " |− {λs. s tm = 1} sum   :== (λs. 1) {?post_sum}"
     apply(rule  conseq)
     prefer  2
     apply( rule_tac  P="?post_sum"  in ass)
     apply(simp_all  add:  update_def  no_alias)
     done
   note init_sum=this
   let ?post_i  = "λs. s  i  = 0 ∧s  sum = 1 ∧s tm = 1"
   have " |− {λs. s  sum = 1 ∧s tm = 1}  i  :== (λs. 0) {?post_i}"
     apply(rule  conseq)
     prefer  2
     apply( rule_tac  P="?post_i"  in ass)
     apply(simp_all  add:  update_def  no_alias)
     done
   note  init_i =this
   have " |− {λs. (?inv  s) ∧    s  sum ≤s  a}
              i  :== λs. s  i  + 1 ;  (tm :== λs. s tm + 2 ; sum :== λs. s tm + s sum )
           {?inv}"
   proof −
     let ?post_i  = "λs.(s  i)  ∗  (s  i)  = s sum
                         ∧ s  tm + 2 = (2 ∗ (s i)  + 1)
                         ∧ (s  i  − 1) ∗ (s  i  − 1) <= (s a) ∧0 < (s  i)  ∧
s  sum ≤s  a "
     have " |− {λs. (?inv  s) ∧  s  sum ≤s  a}
```

```
                i :== λs. s i + 1
               {?post_i}"
    apply(rule conseq)
    prefer 2
    apply(rule_tac P="?post_i" in ass)
    apply(simp_all add: update_def no_alias)
    done
  note while_i=this
  let ?post_tm = "λs. (s i) * (s i) = s sum
                    ∧ s tm - 2 = (2 * (s i) - 1)
                    ∧ (s i - 1) * (s i - 1) <= (s a)
                    ∧ (0 < (s i)) ∧ (1 < (s tm)) ∧ s sum ≤s a"

  have " |- {?post_i}
               tm :== λs. s tm + 2
               {?post_tm}"
    apply(rule conseq)
    prefer 2
    apply(rule_tac P="?post_tm" in ass)
    apply(simp_all add: update_def no_alias)
    done
  note while_tm = this
  have " |- {?post_tm}
               sum :== λs. s tm + s sum
               {?inv}"
    apply(rule conseq)
    prefer 2
    apply(rule_tac P="?inv" in ass)
    apply(simp_all add: update_def no_alias)
    apply(arith)
    done
  note while_sum = this
  show ?thesis
    apply(rule semi)
    apply(rule while_i)
    apply(rule semi)
    apply(rule while_tm)
    apply(rule while_sum)
    done
qed
```

```
    note whileInv = this
    show ?thesis (* main proof *)
      apply(unfold squareroot_def)
      apply(rule conseq)
      prefer 2
      apply(rule semi)
      apply(rule init_tm)
      apply(rule semi)
      apply(rule init_sum)
      apply(rule semi)
      apply(rule init_i)
      apply(rule conseq)
      prefer 2
      apply(rule_tac P="?inv" in While)
      apply(rule whileInv)
      apply(simp_all add: pre_def post_def update_def no_alias)
      prefer 2
      apply(rule allI, rule impI, assumption)
      apply(arith)
      done
  qed
```

11.2.2 Exercise 44

Verify the IMP-program of the previous exercise *without* using the Hoare-calculus explicitly. The idea is to use the verification condition generator vc in theory `VC.thy` running over an annotated program, i.e. the program enriched by the crucial invariants.

(Here, we do not need an in-depth understanding of vc, we just apply it).

The abstract syntax of annotated programs is given in `VC` by the datatype:

```
datatype acom = Askip
              | Aass   loc aexp
              | Asemi  acom acom
              | Aif    bexp acom acom
              | Awhile bexp assn acom
```

(The assn in the Awhile-case is the invariant).

Note: The crucial theorem vc_sound allows for the reduction of the Hoare-triple

```
|- {pre} squareroot tm sum i a {post a i}
```

to a HOL-formula generated by vc.

Hints:

- Do not forget to assume that the locations for i ,tm, sum and a are pairwise distinct.
- Give the annotated program aprog first (the let-statement may help here!).
- Prove the subgoal squareroot tm sum i a = astrip aprog, i.e. the annotated program must be the previously defined program squareroot if the annotations are "stripped away".
- Prove the subgoal pre = awp aprog (post a i), i.e. the weakest precondition computed from the program is equivalent to the precondition.
- apply theorem vc_sound.
- compute and solve the verification condition.
- Use update_def in the simplifier set to handle updates.

Answer to Exercise 44

- a structured verification proof for the annotated squareroot-program:

```
lemma sqrt_verify_vc:
  assumes no_alias : "sum ≠i ∧ i ≠sum ∧tm ≠sum ∧
                      sum ≠tm ∧sum ≠a ∧ a ≠sum ∧
                      tm ≠i ∧ i ≠tm ∧tm ≠a ∧ a ≠tm ∧
                      a ≠i ∧ i ≠a"
  shows "|− {pre} squareroot tm sum i a {post a i}"
  proof −
    (∗ composing the annotated program
       step by step ∗)
    let ?s1 =   "Aass tm (λs. 1)"
    let ?s2 =   "Aass sum(λs. 1)"
    let ?s3 =   "Aass i   (λs. 0)"
    let ? init  = "Asemi ?s1 (Asemi ?s2 ?s3)"
    let ?s4 =   "Aass  i  (λs. (s i) + 1)"
    let ?s5 =   "Aass  tm (λs. (s tm) + 2)"
    let ?s6 =   "Aass  sum (λs. (s tm) + (s sum))"
    let ?body = "Asemi ?s4 (Asemi ?s5 ?s6)"
    (∗ here comes the crucial invention : ∗)
    let ?inv  = "λs. (s i + 1) ∗ (s i + 1) = s sum ∧
```

```
                          s tm = (2 ∗ (s i) + 1) ∧
                          (s i) ∗ (s i) <= (s a) "
   let  ?cond = "λs. (s sum) <= (s a)"
   let  ?aprog = "Asemi ?s1 (Asemi ?s2 (Asemi ?s3
                     (Awhile ?cond ?inv ?body)))"

   have "squareroot tm sum i a = astrip ?aprog"
        by(simp add: squareroot_def)
   note A = this

   have "pre = awp ?aprog (post a i)"
        by(simp add: no_alias pre_def update_def)
   note B = this

   show ?thesis (∗ main proof ∗)
        apply(simp only : A B)
        apply(rule mp[OF spec[OF vc_sound]])
        apply(auto simp: update_def no_alias post_def)
             (∗ this is the beauty of this approach −
                the verification condition is computed and
                simply blown away by auto ∗)
        done
 qed

end
```

12 HOL: Specifying and Proving AVL-Trees

This exercises describes a small modeling and verification project going over two weeks. We will specify and analyze a widely-used data structure AVL-trees as a (purely functional) implementation.

As proof techniques, we will use automatic case splitting in the simplifier supporting reasoning over recursive functions with pattern matching (**recdef**). Finally, we use Isabelle's code generator the convert the function definitions into "real" SML programs.

12.1 The Problem: AVL trees

In 1962 Adel'son-Vel'skiĭ and Landis introduced a class of balanced binary search trees (called AVL trees) that guarantee that a tree with n internal nodes has height $O(\log n)$. The efficiency of AVL tree hinges on the fact that a tree should be *balanced* and *ordered*. Of course, when a node is inserted into or deleted from the tree, these properties must be maintained, by certain rotation operations on AVL trees. Note that unless a tree contains $2^n - 1$ nodes for some n, it cannot be "exactly" balanced. All we can expect is that the height of the left and right subtrees differ by at most one. In order to decide in which way a tree should be rotated, it is convenient to have a function *bal* that tells us if a tree is perfectly balanced, or heavier on the right, or heavier on the left. The necessary rotations are illustrated in Fig. 12.1-12.4.

AVL

bal

12.1.1 AVL tree insertion

We explain insertion of a node into an AVL tree in order to motivate the use of the rotation functions. Suppose we insert a node x into a (balanced ordered) tree $Node\, n\, l\, r$. If $x = n$, then x should not be inserted at all since the property of being ordered requires that the tree contains no duplicates. If $x < n$, we must insert x into l (to maintain the ordering). Let l' be the tree obtained by inserting x into l, and assume that it is balanced and ordered. As an

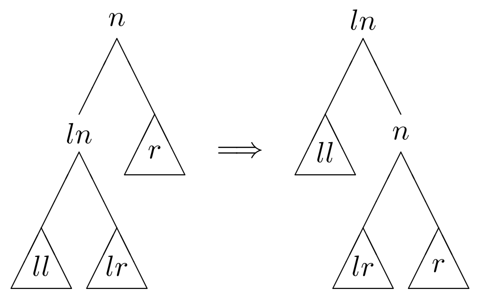

Figure 12.1: r_rot

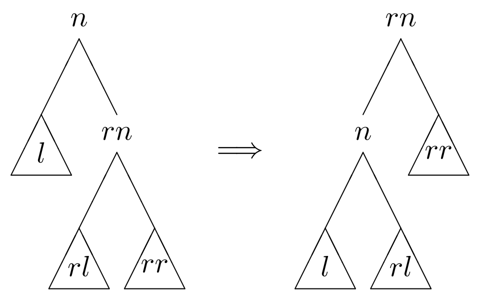

Figure 12.2: l_rot

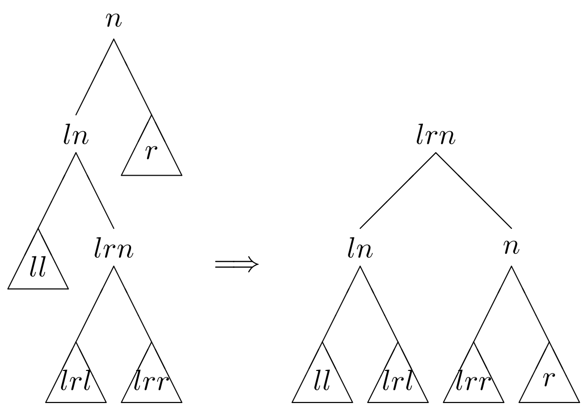

Figure 12.3: lr_rot

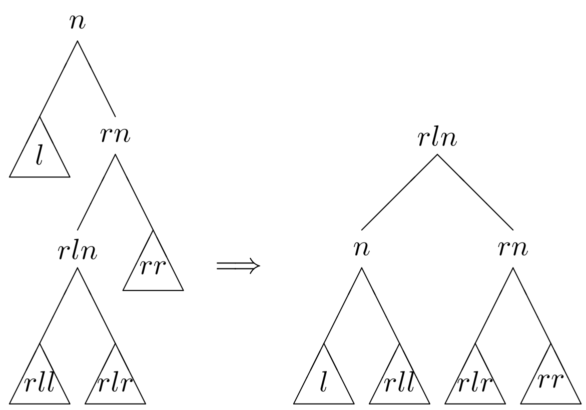

Figure 12.4: *rl_rot*

intermediate result, we have the tree $Node\, n\, l'\, r$. It is ordered, but it might not be balanced.

In fact, it might be the case that $height\ l = height\ r + 1$ and $height\ l' = height\ l + 1$. Then $height\ l' = height\ r + 2$ and so $Node\ n\ l'\ r$ is not balanced. Note that in all other cases, $Node\ n\ l'\ r$ is balanced.

So suppose that $height\ l' = height\ r + 2$. Then $Node\ n\ l'\ r$ looks as shown in the first picture of Figure 12.5, where either $height\ ll' = height\ r + 1$ or $height\ lr' = height\ r + 1$[1]. The tree is too heavy on the left, and rotation must rectify this. We distinguish two cases:

$bal\ l' = Right$. Since l' is balanced, this means that $height\ lr' = height\ r + 1$ and $height\ ll' = height\ r$. So, since lr' has height > 0, it follows that $Node\ n\ l'\ r$ actually looks as shown in the second picture of Figure 12.5, where both lrl' and lrr' have height $height\ r$ or $height\ r - 1$. Since all three trees ll', lrl', lrr' have the same height as r or one less, it follows that *lr_rot* produces a balanced tree (see Figure 12.3).

$bal\ l' \neq Right$. In this case, $height\ ll' = height\ r + 1$, and, since l' is balanced, $height\ lr' = height\ r + 1$ or $height\ lr' = height\ r$. One can easily see that *r_rot* produces a balanced tree (see Figure 12.1).

[1] Never both actually, but this is not needed in the proofs.

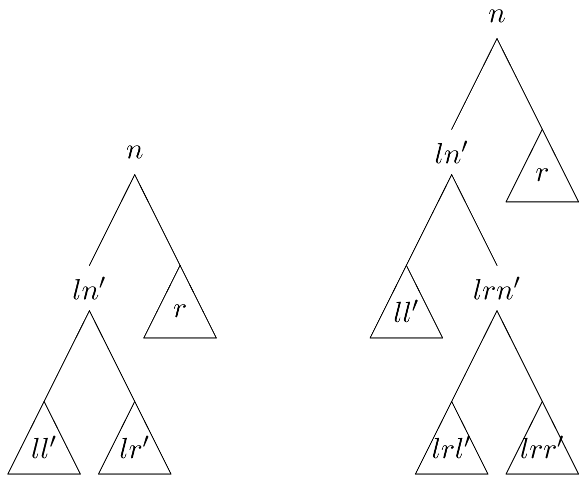

Figure 12.5: Too heavy on left

12.1.2 Efficient AVL trees

In the sequel, we discuss more efficient implementations of AVL-tree's. New function definitions and enriched data types were given; at the end, lemmas were proven that reveal the exact relationship between the new function versions processing new data to the old ones.

The overall scheme is also well-known as a *data refinement*.

The first inefficiency we noticed is that isin traverses the entire tree, which is unnecessary in case the tree is ordered. Note that for verification purposes, the more general but inefficient version is still sometimes in-dispensary.

Another inefficiency we noted is related to bal, which calls height for any node during insertion at the insertion path. A solution here is to store the result of bal as an additional attribute in an extended version of the AVL tree. As a consequence, this attribute must be kept consistent during operations on the enriched tree.

12.2 More on Isabelle

12.2.1 Some non-elementary constructs of ISAR

In the previous exercise, we have presented a core-language of ISAR, providing constructs such as **note** for binding (parts of) a current proof state as *fact* to a name that can be referenced later, or the **let** for introducing meta-variables as abbreviations of terms, which can also occur in propositions, substitutions or other pattern-match constructs such as (is <pattern>).

On top of this, we will now introduce a number of short-cuts that allow for an implicit management of facts and meta-variables.

also
moreover
ultimately
then
thus
with

```
also ≡ note calculation = this          (* initial case *)
also ≡ note calculation = r ∘ ( calculation @ this)
                                        (* for some rule r
                                           out of some given set of
                                           transitivity rules *)

     finally   ≡ also from calculation
    moreover   ≡ note calculation = calculation @ this
  ultimately   ≡ moreover from calculation
        then   ≡ from this
        thus   ≡ hence ≡ then show
with <facts>   ≡ from <facts> this
```

Here, calculation is a standard name for a list of facts,@ the concatenation on them, ∘ the forward resolution.

Similar to calculation , there is a generic name "..." which refers to the right-hand side of the most recent explicit fact statement. This allows to represent calculational sequences as follows:

```
have "x1 = x2" <proof>
also have " ... = x3" <proof>
also have " ... = x4" <proof>
 finally have x1 = x4 .
```

Note that the "." at the very end is again an abbreviation for **by**(**this**).

One of the more trickier constructs of ISAR is the case distinction construct, which works as well for case-splits as for inductions. For a current proof state, goal by goal, it allows for creating sub-proofs referenced by names. The details of this construction are quite involved (see Nipkow's Paper "Structured Proofs in ISAR/HOL" for details), here we give just an example:

```
lemma "length(tl xs) = length xs −1
proof (cases xs)
   case Nil           thus ?case by simp
next
   case (Cons y ys) thus ?case by simp
qed
```

12.3 Exercises

Get the template theory `http://www.infsec.ethz.ch/education/permanent/csmr/material/HOL_AVL_tmpl.thy` and complete it.

12.3.1 Exercise 46

Define the function *insert* :: "'a :: order ⇒ 'a tree <\Rightarrow> 'a tree". This involves the definition of:

- height, which computes the maximal number of nodes on a path from the root to a leaf,
- is_ord , which decides that for each node labeled n, all node labels in the left subtree are smaller, and all labels in the right subtree are greater,
- is_bal , which decides that it is either a leaf or a node with balanced subtrees where the height differs at most by one,
- is_in_eff which should provide a $O(lnn)$ implementation for ordered trees,
- the elementary rotation operations l_rot and lr_rot (analogously to the given functions r_rot and rl_rot),
- the balancing operation r_bal (analogously to the given functions l_bal),
- and finally the *insert* function.

Of course, your definitions should allow to prove the properties in the subsequent lemmas.

Hint: Use the general well-founded recursion mechanism:

```
recdef f "<wf_order"
  "f pat_1 = ... "
   ...
  "f pat_n = ... "
```

supporting pattern matching as in SML or Haskell whenever necessary.

Answer to Exercise 46

```
datatype 'a tree = ET | MKT 'a "'a tree" "'a tree"

consts
  height :: "'a tree ⇒ nat"
  is_in  :: "'a ⇒ 'a tree ⇒ bool"
  is_ord :: "('a::order) tree ⇒ bool"
  is_bal :: "'a tree ⇒ bool"

primrec
  "height ET = 0"
  "height (MKT n l r) = 1 + max (height l) (height r)"

primrec
  "is_in k ET = False"
  "is_in k (MKT n l r) = (k=n ∨ is_in k l ∨ is_in k r)"

primrec
 isord_base : "is_ord ET = True"
 isord_rec :  "is_ord (MKT n l r) = ((∀n'. is_in n' l ⟶ n' < n) ∧
                                      (∀n'. is_in n' r ⟶ n < n') ∧
                                      is_ord l ∧ is_ord r)"

primrec
  "is_bal ET = True"
  "is_bal (MKT n l r) = ((height l = height r ∨
                          height l = 1+height r ∨
                          height r = 1+height l) ∧
                         is_bal l ∧ is_bal r)"

datatype bal = Just | Left | Right

constdefs
  bal :: "'a tree ⇒ bal"
  "bal t ≡ case t of ET          ⇒ Just
              | (MKT n l r) ⇒ if height l = height r then Just
                              else if height l < height r then Right
                              else Left"
```

```
consts
  r_rot   :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  l_rot   :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  lr_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  rl_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"

recdef r_rot "{}"
  "r_rot (n, MKT ln ll lr, r) = MKT ln ll (MKT n lr r)"

recdef l_rot "{}"
  "l_rot (n, l, MKT rn rl rr)  = MKT rn (MKT n l rl) rr"

recdef lr_rot "{}"
  "lr_rot (n, MKT ln ll (MKT lrn lrl lrr), r) =
               MKT lrn (MKT ln ll lrl) (MKT n lrr r)"

recdef rl_rot "{}"
  "rl_rot (n, l, MKT rn (MKT rln rll rlr) rr) =
                 MKT rln (MKT n l rll) (MKT rn rlr rr)"


constdefs
  l_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
  "l_bal n l r ≡ if bal l = Right
                 then lr_rot (n, l, r)
                 else r_rot (n, l, r)"

  r_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
  "r_bal n l r ≡ if bal r = Left
                 then rl_rot (n, l, r)
                 else l_rot (n, l, r)"

consts
  insert :: "'a::order ⇒ 'a tree ⇒ 'a tree"

primrec
"insert x ET = MKT x ET ET"
"insert x (MKT n l r) =
   (if x=n
    then MKT n l r
```

```
else  if  x<n
     then  let  l '  =  insert  x  l
           in  if  height  l '  = 2+height r
               then  l_bal  n  l '  r
               else  MKT n l' r
     else  let  r '  =  insert  x  r
           in  if  height  r '  = 2+height l
               then  r_bal  n  l  r '
               else  MKT n l r')"
```

12.3.2 Exercise 47

Prove two of the following properties of your programs:

1.

```
lemma  is_in_insert :
" is_in  y ( insert  x  t)  = (y=x Z is_in  y  t)"
```

2.

```
lemma  is_in_eff_correct  [ rule_format  (no_asm)]:
" is_ord  t  _  ( is_in  k t  =  is_in_eff  k t)"
```

3.

```
lemma  is_ord_insert :
" is_ord  t  d  is_ord ( insert  (x ::' a :: linorder )  t)"
```

Hint: After giving the definitions in Exercise 46, the commented proof scripts should work again. Try to prove analogous cases and to flush out the **sorry**'s.

Answer to Exercise 47

1.

```
lemma  is_in_insert :
" is_in  y  ( insert  x  t)  = (y=x ∨ is_in  y  t)"
apply (induct  t)
 apply simp
```

```
apply (simp add: l_bal_def is_in_lr_rot is_in_r_rot r_bal_def
                 is_in_rl_rot is_in_l_rot )
apply blast
done
```

2.

```
lemma is_in_eff_correct [rule_format (no_asm)]:
" is_ord t ⟶ ( is_in k t = is_in_eff k t)"
apply ( induct_tac "t")
apply (simp (no_asm))
apply ( case_tac "k = a")
apply (auto);
done
```

3.

```
lemma is_ord_insert :
" is_ord t ⟹ is_ord ( insert (x ::' a :: linorder ) t)"
apply (induct t)
 apply simp
apply (cut_tac x = "x" and y = "a" in linorder_less_linear )
apply (fastsimp simp add: l_bal_def is_ord_lr_rot is_ord_r_rot r_bal_def
                          is_ord_l_rot is_ord_rl_rot is_in_insert )
done
```

12.3.3 Exercise 48 (*optional, tricky*)

A data refinement is provided by the new tree structure:

```
datatype 'a etree = EET | EMKT bal 'a "'a etree" "'a etree"
```

where the balancing information is directly stored in the tree.

1. Define a recursive definition of insertE on etree's that avoids the re-computation of height.
2. Speculate: What should be the crucial properties of this definition ? (state lemmas with sorry)!
3. Speculate: What could be a possible proof plan (state lemmas with sorry)?

12.4 Encoding AVL trees in in Isabelle (skeleton)

```
theory AVL = Main:

datatype 'a tree = ET | MKT 'a "'a tree" "'a tree"

consts
  height :: "'a tree ⇒ nat"
  is_in  :: "'a ⇒ 'a tree ⇒ bool"
  is_ord :: "('a::order) tree ⇒ bool"
  is_bal :: "'a tree ⇒ bool"


primrec
  "is_in k ET = False"
  "is_in k (MKT n l r) = (k=n ∨is_in k l ∨ is_in k r)"


(* ************************************************* *)
(* Define height, is_ord, is_bal, is_in_eff here ... *)
(* ************************************************* *)

datatype bal = Just | Left | Right

constdefs
  bal :: "'a tree ⇒ bal"
  "bal t ≡case t of ET          ⇒ Just
                  | (MKT n l r) ⇒if height l = height r then Just
                                 else if height l < height r
                                      then Right else Left"

consts
  r_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  l_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  lr_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  rl_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"


recdef r_rot "{}"
  "r_rot (n, MKT ln ll lr, r) = MKT ln ll (MKT n lr r)"

recdef rl_rot "{}"
  "rl_rot (n, l, MKT rn (MKT rln rll rlr) rr) =
          MKT rln (MKT n l rll) (MKT rn rlr rr)"


(* ***************************************************** *)
(* Define the analogous functions l_rot and lr_rot here *)
(* ***************************************************** *)


constdefs
  l_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
  "l_bal n l r ≡ if bal l = Right
                 then lr_rot (n, l, r)
                 else r_rot (n, l, r)"


(* ***************************************************** *)
(* Define the analogous function rbal here.             *)
(* ***************************************************** *)


(* ***************************************************** *)
(* Define the insert function for 'a::order on 'a tree. *)
(* ***************************************************** *)
consts
   insert :: "'a::order ⇒ 'a tree ⇒ 'a tree"

subsection "is-bal"

declare Let_def [simp]

lemma is_bal_lr_rot:
"⟦ height l = Suc(Suc(height r));
```

```
   bal l = Right; is_bal l; is_bal r ⟧
 ⟹ is_bal (lr_rot (n, l, r))"
sorry


lemma is_bal_r_rot:
"⟦ height l = Suc(Suc(height r));
   bal l ≠ Right; is_bal l; is_bal r ⟧
 ⟹ is_bal (r_rot (n, l, r))"
sorry


lemma is_bal_rl_rot:
"⟦ height r = Suc(Suc(height l));
   bal r = Left; is_bal l; is_bal r ⟧
 ⟹ is_bal (rl_rot (n, l, r))"
sorry


lemma is_bal_l_rot:
"⟦ height r = Suc(Suc(height l)); bal r ≠ Left; is_bal l; is_bal r ⟧
  ⟹ is_bal (l_rot (n, l, r))"
(*
apply (unfold bal_def)
apply (cases r)
 apply simp
apply (simp add: max_def split: split_if_asm)
done
 *)
sorry

text {* Lemmas about height after rotation *}

lemma height_lr_rot:
"⟦ bal l = Right; height l = Suc(Suc(height r)) ⟧
 ⟹ Suc(height (lr_rot (n, l, r))) = height (MKT n l r) "
(*
apply (unfold bal_def)
apply (cases l)
 apply simp
apply (rename_tac t1 t2)
apply (case_tac t2)
 apply simp
apply (simp add: max_def split: split_if_asm)
done
*)
sorry

lemma height_r_rot:
"⟦ height l = Suc(Suc(height r)); bal l ≠ Right ⟧
  ⟹ Suc(height (r_rot (n, l, r))) = height (MKT n l r) ∨
          height (r_rot (n, l, r)) = height (MKT n l r)"
sorry


lemma height_l_bal:
"height l = Suc(Suc(height r))
  ⟹ Suc(height (l_bal n l r)) = height (MKT n l r) |
          height (l_bal n l r) = height (MKT n l r)"
sorry


lemma height_rl_rot [rule_format (no_asm)]:
"height r = Suc(Suc(height l)) ⟶ bal r = Left
  ⟶ Suc(height (rl_rot (n, l, r))) = height (MKT n l r)"
sorry

lemma height_l_rot [rule_format (no_asm)]:
"height r = Suc(Suc(height l)) ⟶ bal r ≠ Left
 ⟶ Suc(height (l_rot (n, l, r))) = height (MKT n l r) ∨
          height (l_rot (n, l, r)) = height (MKT n l r)"
sorry



lemma height_r_bal:
```

```
"height r = Suc(Suc(height l))
  ⟹ Suc(height (r_bal n l r)) = height (MKT n l r) ∨
        height (r_bal n l r) = height (MKT n l r)"
(*
apply (unfold r_bal_def)
apply (cases "bal r = Left")
 apply (fastsimp dest: height_rl_rot)
apply (fastsimp dest: height_l_rot)
done
*)
sorry


lemma height_insert [rule_format (no_asm)]:
" is_bal t
  ⟶ height (insert x t) = height t ∨ height (insert x t) = Suc(height t)"
sorry
(*
apply (induct_tac "t")
 apply simp
apply (rename_tac n t1 t2)
apply (case_tac "x=n")
 apply simp
apply (case_tac "x<n")
 apply (case_tac "height (insert x t1) = Suc (Suc (height t2))")
  apply (frule_tac n = n in height_l_bal)
  apply (simp add: max_def)
  apply fastsimp
 apply (simp add: max_def)
 apply fastsimp
apply (case_tac "height (insert x t2) = Suc (Suc (height t1))")
 apply (frule_tac n = n in height_r_bal)
 apply (fastsimp simp add: max_def)
apply (simp add: max_def)
apply fastsimp
done
*)

lemma is_bal_insert_left:
"⟦height (insert x l) ≠ Suc(Suc(height r));
  is_bal (insert x l); is_bal (MKT n l r)⟧
 ⟹ is_bal (MKT n (insert x l) r)"
sorry


lemma is_bal_insert_right:
"⟦ height (insert x r) ≠ Suc(Suc(height l));
   is_bal (insert x r); is_bal (MKT n l r) ⟧
 ⟹ is_bal (MKT n l (insert x r))"
sorry

lemma is_bal_insert [rule_format (no_asm)]:
" is_bal t ⟶ is_bal (insert x t)"
sorry

subsection "is−in"

lemma is_in_lr_rot:
"⟦ height l = Suc(Suc(height r)); bal l = Right ⟧
 ⟹ is_in x (lr_rot (n, l, r)) = is_in x (MKT n l r)"
sorry
(*
apply (unfold bal_def)
apply (cases l)
 apply simp
apply (rename_tac t1 t2)
apply (case_tac t2)
 apply simp
apply fastsimp
done
 *)

lemma is_in_r_rot:
"⟦ height l = Suc(Suc(height r)); bal l ≠ Right ⟧
  ⟹ is_in x (r_rot (n, l, r)) = is_in x (MKT n l r)"
sorry
```

```
(*
apply (unfold bal_def)
apply (cases l)
 apply simp
apply fastsimp
done
 *)

lemma is_in_rl_rot:
"⟦ height r = Suc(Suc(height l)); bal r = Left ⟧
 ⟹ is_in x (rl_rot (n, l, r)) = is_in x (MKT n l r)"
sorry
(*
apply (unfold bal_def)
apply (cases r)
 apply simp
apply (rename_tac t1 t2)
apply (case_tac t1)
 apply (simp add: max_def le_def)
apply fastsimp
done
 *)

lemma is_in_l_rot:
"⟦ height r = Suc(Suc(height l)); bal r ≠ Left ⟧
  ⟹ is_in x (l_rot (n, l, r)) = is_in x (MKT n l r)"
sorry
(*
apply (unfold bal_def)
apply (cases r)
 apply simp
apply fastsimp
done
 *)

lemma is_in_insert:
"is_in y (insert x t) = (y=x ∨ is_in y t)"
sorry

lemma is_in_ord_l [rule_format (no_asm)]:
"is_ord (MKT n l r) ⟶ x < n ⟶ is_in x (MKT n l r) ⟶ is_in x l"
sorry

lemma is_in_ord_r [rule_format (no_asm)]:
"is_ord (MKT n l r) ⟶ n < x ⟶ is_in x (MKT n l r) ⟶ is_in x r"
sorry

subsection "is-in-eff"

lemma is_in_eff_correct [rule_format (no_asm)]:
"is_ord t ⟶ (is_in k t = is_in_eff k t)"
sorry


subsection "is-ord"

lemma is_ord_lr_rot [rule_format (no_asm)]:
"⟦ height l = Suc(Suc(height r));
   bal l = Right; is_ord (MKT n l r) ⟧
 ⟹ is_ord (lr_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases l)
 apply simp
apply (rename_tac t1 t2)
apply (case_tac t2)
 apply simp
apply simp
apply (blast intro: order_less_trans)
done
 *)

lemma is_ord_r_rot:
"⟦ height l = Suc(Suc(height r));
   bal l ≠ Right; is_ord (MKT n l r) ⟧
```

```
  ⟹ is_ord (r_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases l)
apply (simp (no_asm_simp))
apply (auto intro: order_less_trans )
done
 *)

lemma is_ord_rl_rot:
"⟦ height r = Suc(Suc(height l));
    bal r = Left; is_ord (MKT n l r) ⟧
 ⟹ is_ord (rl_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases r)
 apply simp
apply (rename_tac t1 t2)
apply (case_tac t1)
 apply (simp add: le_def)
apply simp
apply (blast intro: order_less_trans )
done
*)

lemma is_ord_l_rot:
"⟦ height r = Suc(Suc(height l)); bal r ≠ Left; is_ord (MKT n l r) ⟧
  ⟹ is_ord (l_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases r)
 apply simp
apply simp
apply (blast intro: order_less_trans )
done
*)


(* insert operation presreves is_ord property *)

lemma is_ord_insert:
"is_ord t ⟹ is_ord( insert (x::'a::linorder) t)"
sorry


subsection "An extended tree datatype with labels for the balancing information"

datatype 'a etree = EET | EMKT bal 'a "'a etree" "'a etree"


text {* Pruning, i.e. throwing away the balancing labels: *}
consts
  strip :: "'a etree ⇒ 'a tree"
primrec
  "strip EET = ET"
  "strip (EMKT b n l r) =
                            MKT n (strip l) (strip r)"

text {* Test if the balancing arguments are correct: *}
consts
   correct_labelled :: "'a etree ⇒ bool"
primrec
  " correct_labelled EET = True"
  " correct_labelled (EMKT b n l r) =
                     (b = bal (MKT n (strip l) (strip r))
                      ∧ correct_labelled l
                      ∧ correct_labelled r)"

text {* Add correct balancing labels: *}
consts
  label :: "'a tree ⇒ 'a etree"
primrec
 "label ET = EET"
```

```
 "label (MKT n l r) = EMKT (bal (MKT n l r)) n (label l)
                                           (label r)"

lemma correct_strip:
" correct_labelled (EMKT b n l r) ⟶(bal (strip (EMKT b n l r)) = b)"
apply (simp (no_asm_simp) add: bal_def)
done

subsection "Reversing of strip and label"

lemma prune_label: "strip (label t) = t"
apply (induct_tac "t")
apply (simp (no_asm))
apply (simp (no_asm))
apply (erule_tac conjI)
apply assumption
done

lemma label_prune: "correct_labelled t ⟹ label (strip t) = t"
apply (induct t)
apply auto
done

end
```

13 HOL: Using Specifications for Code Generation and Testing

This exercises describes two advanced techniques for using formal specifications: *code generation* and *random testing*.

The former is a viable approach to achieve correct functional programs and fast evaluation of complex expressions, the latter may be used for early validations of definitions and formulas.

13.1 More on Isabelle

13.1.1 Isabelle's Code Generator

Isabelle has an own code generator that attempts to convert many constructs occurring in a specification (such as **primrec** or **datatype** definitions) into SML code. Code generation out of verified theories for efficient datatype implementations is a viable approach to achieve correct, non-trivial (functional) programs with Isabelle. For example, you can generate code for the term "foldl op + (0::int) [1,2,3,4,5]" and store it in the file `test.sml` via generate_code

```
generate_code ("test.sml")
  test = "foldl op + (0::int) [1,2,3,4,5]"
```

The code generator can be configured both in more correctness oriented as well as pragmatic ways; it is possible, for example, to map the datatype nat on code resulting from the datatype definition in the theory Nat (thus on the free datatype generated by 0 and Suc) or simply on the SML-datatype int (thus reusing the machine integers based on two's complement representation).

Theories can contain highly generic function definitions that are not representable in a target programming language for a number of reasons:

1. a function may simply be not computable,

2. a function may have a type that is not representable in the target language.

An example for the former is a function definition involving a Hilbert-operator, an example for the latter is isord ('a :: ord) ('a tree) which is not representable in the SML type system but could be — in principle — represented in Haskell (note, however, that isord ('a :: order) ('a tree) could not even be represented in Haskell). In practice, the types of formulas to be converted into code must be sufficiently instantiated when configuring the code generator for a theory. You have mainly three options for configuring the code generator:

types_code

1. associate type constructors with specific SML code, e.g.:

   types_code
   "*"   ("(_ */ _)")

consts_code

2. associate constants with specific SML code, e.g.:

   consts_code
   "Pair"   ("(_,/ _)")

[code]

3. register theorems for code generation. This can be done using the **declare** statement, e.g.

   declare less_Suc_eq [code]

   or the code attribute:

   lemma [code]: "((n::nat) < 0) = False" **by**(simp)

   The used theorem should be either an equation (with only constructors and distinct variables on the left-hand side) or a horn-clause (in the same format as introduction rules of inductive definitions). The latter should denoted by using [code ind].

[code ind]

Finally note, if you omit the ("filename") part of the generate_+code statement, the generated code will be immediately available within Isabelle's ML-environment.

13.1.2 Quickcheck

quickcheck

Inspired by the success of random testing tools (e.g. Quickcheck for Haskell) a similar mechanism for testing lemmas was build into Isabelle: the **quickcheck** command. For example, if we try to prove

lemma rev_append: "rev (xs @ ys) = rev xs @ rev ys" we will have a hard day (caused by a simple typo). Now we can try to find a counter example:

```
lemma rev_append: "rev (xs @ ys) = rev xs @ rev ys"
quickcheck
```

Doing this, Isabelle will respond with:

```
Counterexample found:
xs = [0]
ys = [1]
```

Thus our lemma does not even hold for lists of length one. After fully understanding why this assignment is a counter-example, we can reformulate our lemma:

```
lemma rev_append: "rev (xs @ ys) = rev ys @ rev xs"
```

and prove it.

Note that **quickcheck** uses internally the code generator which means that **quickcheck** can only be used if the code generator is already configured correctly!

13.2 Exercises

13.2.1 Exercise 49

Create a version of your AVL tree specification that works over integers, e.g., *insert* should have the type

```
consts
  insert :: "int ⇒ tree ⇒ tree"
```

and use it for code generation. Store your SML program in a file `avl.sml`. Create a file `avl-test.sml` with the following content:

```
Control.Print.printDepth := 100;   (* only for sml/NJ *)
Control.Print.printLength := 100;  (* only for sml/NJ *)

use "avl.sml";
val elements = [1,5,3,4,8,2,4,6];
val t = foldl (fn (e,t) => insert e t) ET elements;
```

Now start open a shell (i.e., in a xterm) and start the SML Interpreter by typing SML and load your file by executing `use "avl-test.sml"`. Try to understand the shown tree representation and validate that your code produced a correct AVL tree with the elements $1, 2, 3, 4, 5, 6, 8$. Note, that 4 should be only stored once in your tree.

Hints:

- For datatype nat, please write Suc(n) instead of 1+n.
- The code generator will need some hints for the polymorphic max function. Therefore prove the following two theorems and declare them to the code generator:

  ```
  lemma [code]: "((x::nat) <= y) = ((x < y) ∨(x=y))"
  lemma [code]: "(max (a::nat) b) = (if (a <= b) then b else a)"
  ```

- The first two lines in your `avl-test.sml` file configure the pretty printer of New Jersey SML to show more details.

Answer to Exercise 49

1. A version of AVL trees that only works over integers:

```
datatype tree = ET | MKT int "tree" "tree"

consts
  height :: "tree ⇒ nat"
  is_in  :: "'a ⇒ tree ⇒ bool"
  is_ord :: "tree ⇒ bool"
  is_bal :: "tree ⇒ bool"

primrec
  "height ET = 0"
  "height (MKT n l r) = Suc(max (height l) (height r))"

primrec
  " is_bal ET = True"
  " is_bal (MKT n l r) = ((height l = height r ∨
                             height l = Suc(height r) ∨
                             height r = Suc(height l)) ∧
                            is_bal l ∧ is_bal r)"

datatype bal = Just | Left | Right

constdefs
  bal :: "tree ⇒ bal"
```

```
"bal t ≡case t of ET          ⇒ Just
                | (MKT n l r) ⇒ if height l = height r then Just
                                else if height l < height r then Right
                                     else Left"
consts
  r_rot  :: "int × tree × tree ⇒ tree"
  l_rot  :: "int × tree × tree ⇒ tree"
  lr_rot :: "int × tree × tree ⇒ tree"
  rl_rot :: "int × tree × tree ⇒ tree"

recdef r_rot "{}"
  "r_rot (n, MKT ln ll lr , r) = MKT ln ll (MKT n lr r)"

recdef l_rot "{}"
  "l_rot (n, l, MKT rn rl rr)  = MKT rn (MKT n l rl) rr"

recdef lr_rot "{}"
  "lr_rot (n, MKT ln ll (MKT lrn lrl lrr ), r) =
             MKT lrn (MKT ln ll lrl) (MKT n lrr r)"

recdef rl_rot "{}"
  "rl_rot (n, l, MKT rn (MKT rln rll rlr) rr) =
             MKT rln (MKT n l rll) (MKT rn rlr rr)"
constdefs
  l_bal :: "int ⇒ tree ⇒ tree ⇒ tree"
  "l_bal n l r ≡ if bal l = Right
                 then lr_rot (n, l, r)
                 else r_rot (n, l, r)"

  r_bal :: "int ⇒ tree ⇒ tree ⇒ tree"
  "r_bal n l r ≡ if bal r = Left
                 then rl_rot (n, l, r)
                 else l_rot (n, l, r)"

consts
  insert :: "int ⇒ tree ⇒ tree"

primrec
  "insert x ET = MKT x ET ET"
  "insert x (MKT n l r) =
```

```
( if x=n
 then MKT n l r
 else if x<n
      then let l' = insert x l
           in if height l' = Suc(Suc(height r))
              then l_bal n l' r
              else MKT n l' r
      else let r' = insert x r
           in if height r' = Suc(Suc(height l))
              then r_bal n l r'
              else MKT n l r')"
```

2. Preparing the code generator and generating code:

```
lemma [code]: "((x::nat) <= y) = ((x < y) ∨(x=y))"
  by(auto)

lemma [code]: "(max (a::nat) b) = (if (a <= b) then b else a)"
  by(simp add: max_def)


generate_code ("avl.sml")
  insert = "insert"
```

13.2.2 Exercise 50

Use the **quickcheck** command for testing your AVL tree specification "testing" your lemmas. Modify (i.e., introduce bugs) your specifications and try if **quickcheck** finds it. Find at least one example for a bug

- where quickcheck finds a non-trivial counter-example.
- where quickcheck fails in detecting the bug.

Answer to Exercise 50

1. An error in the structure of AVL trees is detected by **quickcheck**:

```
consts
  insert1 :: "int ⇒ tree ⇒ tree"
```

```
primrec
  "insert1 x ET = MKT x ET ET"
  "insert1 x (MKT n l r) =
  (if x=n
   then MKT n l r
   else if x<n
        then let l' = insert1 x l
             in if height l' = 2+height r
                then r_bal n l' r  (* Correct: then l_bal n l' r *)
                else MKT n l' r
        else let r' = insert1 x r
             in if height r' = 2+height l
                then r_bal n l r'
                else MKT n l r')"

lemma is_bal_insert1: "is_bal t ⟹ is_bal (insert1 x t)"
quickcheck
(*
  Counterexample found:
  t = MKT 0 (MKT 3 (MKT -1 ET ET) (MKT 0 ET ET)) (MKT 3 ET ET)
  x = -3
*)
oops
```

2. However, the current implementation of **quickcheck** only generates fairly small integers for testing:

```
consts
  insert2 :: "int ⇒ tree ⇒ tree"

primrec
"insert2 x ET = MKT x ET ET"
"insert2 x (MKT n l r) =
     (if x=n
      then if (10 < x) then (MKT n l ET) else (MKT n l r)
      else if x<n
           then let l' = insert2 x l
                in if height l' = Suc(Suc(height r))
                   then l_bal n l' r
                   else MKT n l' r
```

```
              else let r' = insert2 x r
                   in if height r' = Suc(Suc(height l))
                      then r_bal n l r'
                      else MKT n l r')"

lemma is_bal_insert2: " is_bal t ⟹ is_bal (insert2 x t)"
quickcheck
(* no counter example found! *)
oops

lemma is_bal_insert2: "∃ x.∃t. is_bal (t) ∧ ¬ is_bal (insert2 x t)"
  apply(rule exI)
  apply(rule_tac t="insert2 12" in subst)
  apply(simp)
  apply(rule exI)
  apply(rule_tac t="insert2 12
    (MKT 12 (MKT 4 (MKT 3 ET ET) ET) (MKT 8 ET ET ))" in subst)
  apply(simp add: Let_def l_bal_def bal_def)
  apply(auto)
  done
```