Computer Supported Modeling and Reasoning

Dr. Jochen Hoenicke Jürgen Christ

WS13/14

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 1 (22th October 2013)

Propositional Logic This week's exercises will be on *propositional logic*. We will do proofs both using Isabelle and using paper and pencil as we have learned in the lecture.

Isabelle 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.

Isabelle is implemented in the functional programming language (Standard) ML. If you want to get to know (S)ML, consult [1, Appendix 2] for a short overview. See also the explanations at the end of this exercise sheet. Knowledge about (S)ML is not necessary for this course.

Isabelle can be used with different interfaces:

1. directly from a shell;
2. using an editor, e.g. emacs; this allows you to have better control of the files that are involved in an Isabelle session;
3. using a sophisticated specialized interface built on an editor; e.g., the Isabelle integration into jEdit.

In these exercises, we will opt for (3) and use the Isabelle/ISAR syntax.

Thus you should be aware of three layers when you are working with Isabelle: there is the programming language ML, then there is the system Isabelle built on top of it, and finally there is the interface for using Isabelle, which might be jEdit.

Configuring your system for Isabelle Create a working directory, move to it, and type `isabelle jedit`. `jedit` will start with the empty theory file `Scratch.thy`. Save this file as `sheet01.thy` which will be your first file containing Isabelle proofs.

Next we configure the Isabelle integration into`jedit`.

- Make sure the "Auto update" check-box is checked. Otherwise you have to use the "Update" button to see changes reported by Isabelle.
- Choosing the logic: We will use the logic FOL. Go to "Plugins → Plugin Options". This will open a new dialog. In this dialog, select "Isabelle → General". In the spin-box labeled with "Logic" select "FOL". Unfortunately the "Apply" button does not work as expected. So you have to restart jedit.

Files and Directories On the computers you are using, Isabelle version 2013 is installed under `/opt/Isabelle2013/`. You can find the sources for the different logics in the directory `/opt/Isabelle2013/src/`. For now, the folder `FOL` is the most interesting one. You can find the sources online at the `http://isabelle.in.tum.de/library/`.

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 (1) the definition of some syntax and (2) judgements that are postulated to be true. Point (2) will be clarified below. In Isabelle, this theory is contained in a file whose name ends in `.thy`.

There is no special theory file for propositional logic in our distribution, but rather we use the theory files of *first-order logic*, which is a superset of propositional logic. We have already taken care of that above by choosing the logic. The files are named `IFOL.thy` (for intuitionistic first-order logic) and `FOL.thy` (for classical first-order logic).

Syntax and Proofs in Isabelle Understanding the correspondence between a paper and pencil proof and a proof in Isabelle is difficult in the beginning and will take some time.

First, to start a proof script, we need to write an own theory file `sheet01.thy`. This proof script consists of a header, a set of theorems, lemmas, or corollaries with proofs, and an ending. The proof script generally looks like

```
theory sheet01
imports IFOL
begin

...

end
```

The name after the keyword `theory` has to be the same as the file name (without extension) of the proof script file. After the keyword `imports` we can list a number of theories we want to import. For the first sheet we will use `IFOL`. Later we will use different theories and (possibly) some theories created in previous exercises. The entries in the import list are separated by spaces. A theory file is terminated by the `end` keyword. Every other command has to be put between the keywords `begin` and `end`.

Next consider the syntax of formulae in Isabelle. The logical connectives are typed as `-->`, `\/` (or `|`), `/\` (or `&`), and `~`. Note that jedit can display those as the connectives used in paper and pencil proofs. Just type some formulae in `sheet01.thy` to see this (but then erase them again). Note that you have to use the tabulator key to get the symbol currently selected in the popup menu.

Now consider the inference rules. These are listed in `IFOL.thy` beneath the comment `(* Propositional logic *)`. E.g.

```
conjunct1:    "P&Q ==> P"
```

is the Isabelle `FOL` encoding of $\wedge$-*EL*. We want to understand the correspondence between the notation above and the Isabelle encoding of the rules, first by looking at the mere syntactic forms.

Try to suggest how the rule `conjunct1` corresponds to $\wedge$-*EL*. Do the same for some of: `conjunct2` vs. $\wedge$-*ER*, `disjI1` vs. $\vee$-*IL*, `disjI2` vs. $\vee$-*IR*, `FalseE` vs. $\bot$-*E*.

Now suggest how the rule `conjI` corresponds to $\wedge$-*I*, and how `mp` corresponds to $\rightarrow$-*E*.

Finally, suggest how `disjE` corresponds to $\vee$-*E*, and how `impI` corresponds to $\rightarrow$-*I*.

The inference rules `conjI`, `conjunct1` etc. are examples of 'judgements that are postulated to be true' (see above). Technically, each such rule is an expression of type `thm` ('theorem'). Simply type, e.g., `thm conjI;` in the file `sheet01.thy`. In the output buffer you will see `?P` $\Longrightarrow$ `?Q` $\Longrightarrow$ `?P`$\wedge$`?Q`, which gives the value of the expression `conjI`.

Theorems and Proofs A proof in Isabelle is started by stating the formula as a `theorem` or `lemma`, e.g.,

```
theorem conjSymm: "A /\ B --> B /\ A"
```

This starts a new theorem that will have the name `conjSymm`, when it is proved. The name is optional but is needed if the theorem should be referenced later. The formula has to be put in quotes. The theorem must be followed by a proof, otherwise Isabelle will complain. Thus Isabelle only allows us to write down theorems that are correct.

The proof starts with the keyword `proof` which is followed by a proof step. In the first sheets we will always use an application of a rule as the proof step. The rule is applied bottom-up, i.e., the conclusion of the rule is unified with the current goal, in this case the theorem. In our example we want to apply the rule $\rightarrow$-*I*, which corresponds to the theorem `impI` in Isabelle. The proof method `rule` applies a rule (which is a theorem in Isabelle) to the current goal.

```
proof (rule impI)
```

If you position the cursor after this line in the jedit window, you can see in the output window the result of the application of the proof rule. In this case we get the subgoal

$$1.\texttt{A} \wedge \texttt{B} \Longrightarrow \texttt{B} \wedge \texttt{A}$$

You may have to select the `Output` tab and check `Auto update`. The double arrow separates the assumptions from the conclusion. In this case the goal indicates that we have to show $\texttt{B} \wedge \texttt{A}$ from the assumption $A \wedge B$.

An assumption is introduced by the keyword `assume` and can be named with the same syntax used for theorems. The assumption obviously does not need a proof. Then we use the keyword `show` to indicate the goal we have to prove next. Since we never reference it again we do not name the formula we show.

```
  assume ab: "A /\ B"
  show "B /\ A"
    sorry
qed
```

Similarly to the keyword `theorem` the keyword `show` indicates a statement that needs a proof. Instead of giving the proof, we cheat here by writing `sorry`. In jedit the keyword is highlighted to remind us that we have cheated. However using `sorry` is useful to quickly check if the remainder of the proof is okay. With `qed` we can end the proof of the theorem `conjSymm`.

To fix the proof we now have to replace `sorry` by a valid proof for $\texttt{B} \wedge \texttt{A}$. The next proof step is to apply the rule $\wedge$-I, which is named `conjI` in Isabelle. So we replace `sorry` by

```
  proof (rule conjI)
  qed
```

This is not accepted by Isabelle, because the proof is not yet complete. If you place the cursor between `proof` and `qed`, the output window shows the current goals $1.B$ and $2.A$. We can proof B as follows using the rule `conjunct2` which corresponds to $\wedge$-ER:

```
    show "B"
    proof (rule conjunct2)
    qed
```

If you now place the cursor again between `proof` and `qed`, you can see the next goal $\texttt{?P} \wedge \texttt{B}$. The variable $?P$ indicates a so called *metavariable*, which can stand for an arbitrary formula. This metavariable comes from the proof rule $\texttt{conjunct2} : \texttt{?P} \wedge \texttt{?Q} \Longrightarrow \texttt{?Q}$. When applying the rule bottom to top the metavariable `?Q` is unified with the goal B, but the variable `?P` does not occur in the conclusion of the rule. Thus we are still allowed to substitute it by any formula we like. In our case we want to show $\texttt{A} \wedge \texttt{B}$ as this was the assumption introduced by the first proof step. We can use the keyword `from` to reference a fact (theorem, assumption, intermediate proof step) that was introduced before and then show $\texttt{A} \wedge \texttt{B}$ by the proof method `assumption`.

```
      from ab show "A /\ B" by assumption
```

The syntax `by ...` is an abbreviation for `proof ... qed` and can be used if no more proof steps are necessary. This finishes the proof of B, however we have a second goal in the outer proof, namely A. This is indicated in jedit by the error symbols before the second `qed`. We use the keyword `next` to indicate that we need to proof another goal:

```
  next
```

After this keyword we can insert a similar proof for `A`. One can shorten the proof considerably to

```
    from ab show "A" by (rule conjunct1)
```

In this case we add the fact `ab` as an assumption for the goal A. Applying the rule $\texttt{conjunct1} : \texttt{?P} \wedge \texttt{?Q} \Longrightarrow \texttt{?P}$ will now instantiate `?P` with `A` and $\texttt{?P} \wedge \texttt{?Q}$ with the assumption $\texttt{ab} : \texttt{A} \wedge \texttt{B}$. The instantiation succeeds and since there is no other assumption the proof is discharged immediately.

Top-Down Proofs There are many ways to prove theorems in Isabelle. Another way to prove `B ∧ A` from `A ∧ B` starts by first proving `A` and `B` using the rules `conjunct1` and `conjunct2`. The keyword `have` is used to state intermediate facts. So another possible proof of `conjSymm` goes as follows.

```
theorem conjSymm: "A /\ B --> B /\ A"
proof (rule impI)
  assume ab: "A /\ B"
  from ab have a: "A" by (rule conjunct1)
  from ab have b: "B" by (rule conjunct2)
  from b a show "B /\ A" by (rule conjI)
qed
```

Shortcuts You can refer to the last fact (assumption or proven subgoal) using the special name `this`. Furthermore, instead of `from this` you can write `then`. If you want to conclude a fact from the last fact you can write `hence` instead of `then have` and `thus` instead of `then show`.

Exercise 1 (12 points)
Prove the following theorems using paper and pencil and using Isabelle.

For all theorems you prove (also in later exercises), use the number of the exercise as name, e.g. `ex1_1` for $A \wedge B \to B \wedge A$. Then, if needed in the proofs of later exercises, you can use the theorem using its fully qualified named, e. g., if you want to use `ex1_1` from theory file `Ex01.thy` you can reference it as `Ex01.ex1_1` after importing the theory `Ex01.thy`.

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

■

Exercise 2 (6 points)
Prove the following theorems involving negation with paper and pencil and in Isabelle. Look out in `IFOL.thy` for `not_def`. Typing

```
by (unfold not_def)
```

and

```
by (fold not_def)
```

replaces each $\neg\psi$ by $\psi \longrightarrow \bot$ or vice versa. You can also unfold the definition at the beginning of the proof using the proof prefix `unfolding`, i. e., you can start a proof with

```
lemma "...."
unfolding not_def proof ....
qed
```

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

■

Exercise 3 (2 points)
Explain why the following theorem is valid using natural language

$$(P \to A) \to (\neg P \to A) \to A$$

Now try to prove it in Isabelle. Why is this not working? What kind of rule is missing?

To close your proof use the Isar command `oops`. This command ends a proof attempt with a failure. Hence, no theorem is produced. ■

The Bluffer's guide to ML One clarification of [1, Appendix 2]: On top of page 281, it says:

> Patterns are expressions which contain only variables and constructors.

In Section A.2.5 it is explained what a constructor is. Constants are a special case of constructors, namely constructors with 0 arguments. Thus examples of constructors are the integer constants 0,1,2,..., and the list constructors [] and ::. Those happen to be predefined in ML. Other constructors may be defined by the user. Note also that a constructor is not the same thing as a *type* constructor. Sometimes one speaks of a *term* constructor to emphasize that one does not mean a type constructor.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 2 (29th October 2013)

When you do Isabelle proofs, one question is which subgoal should be selected first. In general, you should select the subgoal first whose conclusion is most instantiated. The reason is that the more it is instantiated, the smaller the chance is that it gets wrongly instantiated by unification. You may think: but if it is more instantiated it might be harder to prove. But the point is: each subgoal must be solved eventually anyway.

Classical reasoning So far we have been working in intuitionistic propositional logic. We will now add one further rule

$$\dfrac{\begin{array}{c}[\neg A]\\ \vdots\\ A\end{array}}{A}\ \textit{classical}$$

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

Initial proof step In the proofs done so far we always used an initial proof rule. In some cases it is easier to omit an initial proof step. Isabelle supports this by supplying the initial proof method `-`. Typically such cases are used when we already have a lemma that almost matches our current lemma but needs some reformulation. Assume you have a lemma `lemma_nnp_or_p_imp_r` that states $\neg\neg P \vee (P \to R)$ and you want to prove $\neg\neg P \vee \neg P$. You cannot use the lemma as initial proof step since $\neg P$ does not match with $P \to R$. Furthermore, if we use the proof prefix `unfolding not_def` we have to show $((P \to \mathit{False}) \to \mathit{False}) \vee (P \to \mathit{False})$. This time we cannot use the lemma as initial proof step since the left argument of the disjunction does not match with $\neg\neg P$ (the left disjunct of the lemma). One solution is to not apply an initial proof step and start the proof with

```
proof -
  have "~~P\/(P-->False)" by (rule lemma_nnp_or_p_imp_r)
```

Finishing this proof simply is a matter of folding `not_def`.

Files used for this sheet Since on this sheet we will use intuitionistic and classical logic, we will split the proofs in different files. Create a file `sheet02intuitionistic.thy` that later contains all proofs in intuitionistic logic. In this file, import the theory file `IFOL.thy` using the `imports` keyword in the header.

Do the classical proofs in the file `sheet02classical.thy` in which you import the theory `FOL.thy`. Note that the theory FOL is a superset of the theory IFOL and, among others, contains the rule `classical`.

If you want to use lemmas and theorems proved in a different theory file you can simply add the name of the theory file to the import list. Hence, if you want to use `sheet01.thy` in the file `sheet02intuitionistic.thy` use the import statement

```
imports IFOL sheet01
```

You can reference lemmas and theorems from a theory file by its name. If multiple lemmas have the same name but stem from different theory files, you have to explicitly qualify the theory. Hence, if you want to use theorem `ex01` from theory `sheet01`, but you already have a theorem `ex01` in you current theory, you have to refer to the first one as `sheet01.ex01`.

Exercise 1 (2 points)
We show that *classical* is equivalent to the principle of the excluded middle.

1. Prove $P \vee \neg P$ (hint: you only need the assumptions $\neg(P \vee \neg P)$ and P) in classical logic.
2. Prove $P \vee \neg P \to ((\neg P \to P) \to P)$ intuitionistically.

(Hint: Consider Exercise 2 on Sheet 1.) ∎

Exercise 2 (2 points)
Prove the following classical theorem called *Peirce's law*, both using paper and pencil and in Isabelle:

$$((A \to B) \to A) \to A$$

(Hint: use Exercise 1.1 and another previously proven exercise.) ∎

From now on, all exercises will be Isabelle exercises unless it is clear from the context that they must be paper-and-pencil exercises or this is explicitly said. But keep in mind that the proofs done in Isabelle correspond to paper-and-pencil proofs. Hence, if you have problems solving an exercise, it might be helpful to draw a proof tree on paper.

Applying rules as introduction Let us consider what happens when applying a rule to a goal. We simplify matters by ignoring the issue of *instantiation* of rules, which was treated on Sheet 1. Suppose one of our current subgoals is ϕ which reads: if we are able to proof ϕ from the assumptions made before, we are done. We want to use a rule $[\![\phi_1; \ldots; \phi_m]\!] \Longrightarrow \phi$, which reads: if we have proofs of $\phi_1, \ldots, \phi_m$, we have a proof of ϕ. Now `rule` replaces the subgoal with new subgoals

$$\begin{array}{l} \phi_1 \\ \ldots \\ \phi_m \end{array} \tag{1}$$

Think about why this is correct: If we are able to prove all $\phi_1, \ldots, \phi_m$ from the previous assumptions, then, by the rule we have shown ϕ. Thus it is correct to say: If we have shown (1), we are done.

Assumption lists If you look for the rule `mp` in the file `IFOL.thy` you will find the rule `[|P --> Q; P|] ==> Q`. Now use the command `thm` to display the rule `mp` in Isabelle. You get `?P --> ?Q ⟹ ?P ⟹ ?Q`. Note that the format of the premises was changed from a list (indicated by [|...|]) to a sequence (indicated by $\Longrightarrow$). In fact, Isabelle makes no distinction between the two forms. Semantically, this corresponds to the equivalence of $A_1 \wedge \ldots \wedge A_n \to B$ and $A_1 \to \ldots \to A_n \to B$.

Hypothetical derivations A rule containing a hypothetical derivation, where assumptions are discharged, e. g.,

$$\dfrac{\begin{array}{c}[P] \\ \vdots \\ \bot\end{array}}{\neg P}\ \neg\text{-}I$$

is represented in Isabelle as `(P ⟹ False) ⟹ ¬P`. So the assumption is written as $P \Longrightarrow \mathit{False}$. You can intuitively read the $\Longrightarrow$ symbol as "the right hand side is derivable from the left hand side". In this example, if you can derive *False* from P, then from that fact you can derive $\neg P$ (without the assumption P). In principle, the $\Longrightarrow$ in Isabelle can be arbitrarily nested, but there is no corresponding natural deduction rule for $((A \Longrightarrow B) \Longrightarrow C) \Longrightarrow D$.

Derived Rules On Sheet 1, it was explained that (1) the inference rules like `conjI`, `conjunct1` etc., (2) proven formulae like $A \to A$, (3) definitions of syntactic conversions like `not_def`, are expressions of type `thm`. We will now see that it also possible to *derive* rules in Isabelle.

A rule is a lemma that contains premises. If a lemma in Isabelle contains the meta-level implication sign `==>` it is a rule. As example, lets derive the rule `conjSymmRule`:

$$\dfrac{B \wedge A}{A \wedge B}\ \wedge - \mathit{symm}$$

To formulate this rule in Isabelle, we write

```
lemma conjSymmRule: "A/\B==>B/\A"
```

We can reuse most parts of the proof from Sheet 1:

```
proof (rule conjI)
  assume ab: "A/\B"
  from ab show "A" by (rule conjunct1)
  from ab show "B" by (rule conjunct2)
qed
```

Compared to the proof of $A \wedge B \to B \wedge A$ we don't need the rule $\to$-*I*. If you compare the rule `conjSymmRule` to the lemma `conjSymm` from Sheet 1 you see that there is only one difference: the implication symbol `-->` is turned into the meta-level implication `==>`. Isabelle can do that conversion for you as well. With the command `lemmas` we can define lemmas that are slightly modified version of other lemmas. In our case we would like to convert the lemma `conjSymm` into a rule. We can use the attribute `rule_format` to accomplish this task, e. g.,

```
lemmas conjSymmRule = conjSymm[rule_format]
```

If we print this rule with `thm` we get `?A∧?B ⟹?B∧?A` which is indeed the rule we wanted.

There is one additional remark about derived rules. If we want to derive a rule containing a hypothetic derivation, e. g., (`P` $\Longrightarrow$ `False`) $\Longrightarrow$ `P`, you can assume `P` $\Longrightarrow$ *False* in Isabelle and give it a name, say `pfalse`. This can be used as a rule that reads: if we have a proof of `P`, we have a proof of *False*. Thus, if our current subgoal is *False* and we apply `(rule pfalse)` we get the new subgoal `P`.

Some common derived rules of propositional logic

$$\frac{\begin{array}{c}[P,Q]\\ \vdots\\ P\wedge Q \qquad R\end{array}}{R}\ \wedge\text{-}E \qquad \frac{\begin{array}{cc}\multicolumn{2}{c}{[Q]}\\ \multicolumn{2}{c}{\vdots}\\ P\to Q \quad P \quad R\end{array}}{R}\ \to\text{-}E' \qquad \frac{\begin{array}{c}[P]\\ \vdots\\ \bot\end{array}}{\neg P}\ \neg\text{-}I \qquad \frac{\neg P \quad P}{R}\ \neg\text{-}E$$

For $\wedge$-*E*, note that the notation means: discharge zero or more occurrences of P, and discharge zero or more occurrences of Q.

Exercise 3 (1 point)
The Isabelle encoding of $\to$-E' is

```
[| P-->Q;  P;  Q ==> R |] ==> R
```

Derive this rule intuitionistically in Isabelle. ∎

Actually, it already existed before. It is contained in `IFOL.thy` and called `impE`. In `IFOL.thy`, you will also find `conjE` ($\wedge$-*E*), `notI` ($\neg$-*I*), and `notE` ($\neg$-*E*). Note that these lemmas are proven using an older syntax.

Exercise 4 (4 points)

1. Derive the rule
$$\frac{\begin{array}{c}[A]\\ \vdots\\ A\to B\end{array}}{A\to B}$$
using paper and pencil, as well as in Isabelle. Do it intuitionistically! (Hint: besides the premise it requires only two rule applications.)

2. Derive the rule
$$\frac{\begin{array}{c}[A]\\ \vdots\\ \neg A\end{array}}{\neg A}\ \textit{classical\_dual}$$
using paper and pencil, as well as in Isabelle. Again, do it intuitionistically! (Hint: the previous exercise might be useful.)

The first part shows an interesting technique that may sometimes be useful for doing Isabelle proofs: Whenever you want to prove $A \to B$, you may use A as an additional assumption.

Concerning the second part, compare the derived rule to the rule *classical* (you do not need to comment on it, just compare for yourself)! ∎

Elimination tactic Elimination rules such as `conjE`, `impE`, `disjE` and `FalseE` are designed to be used in combination with elimination resolution. In the Isabelle proofs we mostly applied the proof rules "backwards": To *get rid* of a connective in the conclusion we want to prove, we used an introduction rule, and to *obtain* a connective in the conclusion, we used an elimination rule. Elimination resolution allows to use *eliminate a connective in the premises*. We will now explain this.

The pragmatics of elimination rules and elimination resolution are that they allow you to manipulate premises, or more precisely, break a premise into pieces. For example, when you have the premise $A \wedge B$, you may want to replace this premise with the two premises A and B. A simple example is the following proof excerpt:

```
assume "A/\B" then have "A" by (rule conjE)
```

But this looks exactly like what we have done until now. So when is the application of a rule an elimination?

Whenever you have assumption like $\texttt{A} \wedge \texttt{B}$ in the example above and a conclusion, Isabelle will unify the assumptions with the premises of the rule and the conclusion of the rule with the subgoal. Consider the example above. We want to apply the rule `conjE`, i. e., $[|\texttt{P} \wedge \texttt{Q}; \texttt{P} \Longrightarrow \texttt{Q} \Longrightarrow \texttt{R}|] \Longrightarrow \texttt{R}$ to the premise $\texttt{A} \wedge \texttt{B}$ and the conclusion `A`. Hence, Isabelle matches $\texttt{P} \wedge \texttt{Q}$ with $\texttt{A} \wedge \texttt{B}$ and, hence, instantiates `P` with `A` and `Q` with `B`. Furthermore she matches `R` with `A`. Since the first premise is an assumption, Isabelle closes this branch by assumption because we have a proof for $\texttt{A} \wedge \texttt{B}$, i. e., the assumption proof. The second assumption gets instantiated to $\texttt{A} \Longrightarrow \texttt{B} \Longrightarrow \texttt{A}$. If we replaced the command `by` in the example proof above by the command `proof` we would have to prove this fact. But since the proof is trivially done by assumption, we leave it to the `qed` command of Isabelle.

We now exemplify what elimination resolution does in the general case. Consider a rule $[\![\phi_1; \ldots; \phi_m]\!] \Longrightarrow \phi$. If we apply this rule to the goal ϕ without premises, each premise of the rule gives rise to a new subgoal (explained above in the paragraph on introduction rules).

Now assume we have premises $\psi_1; \ldots; \psi_n$ (where $n \leq m$). If we now apply our rule, all ψ_i for $1 \leq i \leq n$ have to be identical to ϕ_i. Then, all these premises do not introduce new subgoals since they are already solved by assumption. But the remaining premises of the rule $\phi_{n+1}; \ldots; \phi_m$ still give rise to new subgoals:

$$\begin{array}{l} \phi_{n+1} \\ \ldots \\ \phi_m \end{array} \tag{2}$$

You should compare this to (1).

In the non-simplified formulation, you must replace 'identical' with 'unifiable', but it is best you see this in examples.

Proof methods with repetition If we want to repeatedly apply an introduction rule, we can use the method `intro`. For example assume we want to show $\texttt{A} \wedge (\texttt{B} \wedge \texttt{C})$ from the premises `A, B, C`. This can be done by repeatedly applying `conjI`:

```
lemma "A==>B==>C==>A/\(B/\C)"
by (intro conjI)
```

The method `intro` repeatedly matches the conclusion of the given rule with the right hand side of the current goal. If it matches she applies the rule splitting the goal into new goals, one for each premise of the given rule. If the right hand side of one of the newly introduced goal also matches the rule, Isabelle applies the rule again. This is useful for introduction rules. In the example above it will remove all the conjunction symbols from the goals, and finally arrive at the following three sub-goals.

```
1. A==>B==>C==>A
2. A==>B==>C==>B
3. A==>B==>C==>C
```

Remember that `by (...)` is a shorthand for `proof (...) qed`. By default, the `qed` command tries to solve every sub-goal by assumption. That is, when you write `qed` and your current proof state is $\phi_1, \ldots, \phi_n \Longrightarrow \psi$, Isabelle tries to match ψ against any premise ϕ_i. If a match is found, the goal is closed and if all goals can be closed the proof finishes.

You can also give more then one rule to the method `intro`. In that case it will try each of the rules on every goal. Usually, for introduction rules only one of the rules is applicable on a specific goal, as they all introduce a different outermost symbol. The method `intro` should only be used with introduction rules, i.e., rules where the goal is syntactically strictly larger than any of its premises. Otherwise, the command may go into an infinite loop.

Assume you want to show `C` from the premise $\texttt{A} \wedge (\texttt{B} \wedge \texttt{C})$. In this case we need to split the premise into three premises. Then, we have to show `C` using, among others, the assumption `C`. Splitting the premise can be done by a repeated application of the rule `conjE`. Isabelle/Isar provides the proof method `elim` to repeatedly apply a rule using elimination resolution. Hence, our proof is really simple:

```
lemma "A/\(B/\C) ==> C"
by (elim conjE)
```

How does this work? The method `elim` checks for the current goal if the rule is applicable as follows. The rule must be of the form $[|\phi_1, \ldots|] \Longrightarrow \psi$, i. e., it contains at least one premise and the first premise is ϕ_1. Isabelle now tries to simultaneously match ψ with the right hand side of an open goal and ϕ_1 with one of its assumptions. If such a match is possible, she removes the assumption ϕ_1 and replaces ψ with the remaining premises ... (splitting the goal if the rule has three or more premises, or closing the goal if ϕ_1 was the only premise). Then she repeats on all newly created goals until the rule is no longer applicable.

In the example above, the elimination rule is $[|\texttt{P} \wedge \texttt{Q}, ([|\texttt{P}, \texttt{Q}|] \Longrightarrow \texttt{R}|] \Longrightarrow \texttt{R}$. The current goal is $\texttt{A} \wedge (\texttt{B} \wedge \texttt{C}) \Longrightarrow \texttt{C}$ So Isabelle matches `R` with the right hand side `C` and $\texttt{P} \wedge \texttt{Q}$ with the assumption (in this case there is only one assumption). A match is found with $\texttt{P} = \texttt{A}$, $\texttt{Q} = \texttt{B} \wedge \texttt{C}$, and $\texttt{R} = \texttt{C}$. Applying the elimination rule removes the assumption and replaces the current right hand side by the second premise of the elimination rule yielding $[|\texttt{A}, (\texttt{B} \wedge \texttt{C})|] \Longrightarrow \texttt{C}$. Now, the elimination rule is applicable again, since the second premise can match $P \wedge Q$ (and everything can match R). Applying the rule yields the goal $\texttt{A} \Longrightarrow [|\texttt{B}, \texttt{C}|] \Longrightarrow \texttt{C}$, which is the same as $\texttt{A} \Longrightarrow \texttt{B} \Longrightarrow \texttt{C} \Longrightarrow \texttt{C}$. This goal can then be closed by assumption.

In the first example, we can also use the method `elim` on an introduction rule. To see the differences you can replace the `by` command with the `proof` command to see the remaining sub-goals that are proven by assumption with the `qed` command. Isabelle does not show you the intermediate steps. They are as follows. In the first goal $\texttt{A} \Longrightarrow \texttt{B} \Longrightarrow \texttt{C} \Longrightarrow \texttt{A} \wedge (\texttt{B} \wedge \texttt{C})$ it matches $P \wedge Q$ with the conclusion $A \wedge (B \wedge C)$ and simultaneously tries to match the first premise P with any of the assumption. A match is found for $\texttt{P} = \texttt{A}$ and $\texttt{Q} = (\texttt{B} \wedge \texttt{C})$, the assumption A is removed and the goal is replaced by the second premise Q, yielding $\texttt{B} \Longrightarrow \texttt{C} \Longrightarrow \texttt{B} \wedge \texttt{C}$. In the second elimination step the assumption B is removed and the final goal is $C \Longrightarrow C$. Note that the method `elim` only works by chance here. If you put the parenthesis the other way round, $(A \wedge B) \wedge C$ the rule would fail. In general, `elim` should only be used on elimination rules.

The method `elim` is useful to work top-down from the assumptions using an elimination rule that removes operators from the first premise. On the other hand the method `intro` is useful to work bottom-up from the conclusion using an introduction rule that introduces a logical operator in the conclusion (i.e., in the bottom-up reasoning the operator is removed).

Exercise 5 (2 points)
Prove the following intuitionistic theorems using elimination resolution and `disjE` and `conjE` wherever possible.

1. $A \wedge (B \wedge C) \rightarrow (A \wedge B) \wedge C$
2. $(A \vee C) \wedge (B \vee C) \rightarrow (A \wedge B) \vee C$

You may want to compare this to proofs without using elimination resolution. ∎

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 3 (5th November 2013)

Variable Occurrences In lecture "First-Order Logic", it was said that all occurrences of a variable in a term or formula are bound or free or binding. Please search for the definition in the slides.

Exercise 1 (1 point)
Mark each variable occurrence in

$$(\exists x.\, q(x) \vee p(b)) \vee p(c) \to p(x) \wedge \exists y.\, \forall z.\, r(y, f(x,b), g(x,y), g(b,y))$$

as bound (e.g., red), free (e.g., green) or binding (e.g., blue). For each bound occurrence, indicate the corresponding binding occurrence. Assume the common convention that x, y, z are variables and a, b, c are constants. ∎

Exercise 2 (1 point)
Apply the substitution $[z \leftarrow g(z)]$ to $(\forall x.\, p(y, z, g(x))) \wedge \exists z.\, r(g(z))$ following the definition of a substitution (in the chapter on "First-Order Logic" in the slides) step by step. ∎

Syntax extensions In lecture "First-Order Logic", it was mentioned that the syntax of propositional or first-order logic may be extended with further connectives such as $\leftrightarrow$, which provide shorthands for certain formulae. E.g., $\phi \leftrightarrow \psi$ stands for $(\phi \to \psi) \wedge (\psi \to \phi)$. In `IFOL.thy`, we have

```
definition iff (infixr "<->" 25) where "P<->Q == (P-->Q) & (Q-->P)"
```

This defines the equivalence of two syntactic forms in the same way as we have seen for negation in lecture "Propositional Logic". In Isabelle, you will sometimes have to use `unfold` and `fold` to convert between the two syntactic forms. The definition is implicitly named `iff_def`.

Exercise 3 (2 points)
Derive the rule

$$\dfrac{\begin{array}{cc}[B] & [A] \\ \vdots & \vdots \\ A & B\end{array}}{A \leftrightarrow B}$$

in Isabelle and using paper and pencil. (First think carefully about how to encode it in Isabelle; this should definitely involve the symbol `<->`). ∎

First Order Logic We now have some Isabelle exercises on first-order logic, and so we keep using the theory `FOL`, of which propositional logic is a subset. The quantifiers are written `ALL` resp. `!` and `EX` resp. `?` in `FOL`. For example, $\forall y.\exists x.p(x,y)$ is written `ALL y. EX x. p(x,y)` or `! y. ? x. p(x,y)` (where you have to replace `!` and `?` by the corresponding logical symbols as suggested by jedit). Note that quantifiers bind very weak, i. e., $(\forall x.p(x)) \to \exists x.p(x)$ has to be written `(ALL x. p(x)) --> (EX x. p(x))` or `(! x. p(x)) --> (? x. p(x))`, including the parentheses.

Please look up the rules for the quantifiers in the lecture slides! In `IFOL.thy`, these rules are encoded as follows:

```
allI: "(!!x. P(x)) ==> (ALL x. P(x))" and
spec: "(ALL x. P(x)) ==> P(x)" and
exI: "P(x) ==> (EX x. P(x))" and
exE: "[| EX x. P(x);  !!x. P(x) ==> R |] ==> R"
```

The `!!` is the metalevel universal quantification (jedit also displays this as $\bigwedge$). If a goal is preceded by $\bigwedge x$, this means that Isabelle must be able to prove the subgoal in a way which is independent of x, i. e., without instantiating x. Whenever you apply a rule to a subgoal that is preceded by $\bigwedge x$, then the metavariables of the rule, which will occur in the new subgoals through the resolution step, will be made dependent on x. You may also say that those variables will be *Skolem* functions of x.

Note that the rules with `!!` in Isabelle are the rules whose paper and pencil versions have side conditions.

When you are doing proofs in first-order logic, you should introduce metalevel universal quantification into your proofs as early as possible. In other words, rules with a side condition should be applied first. Intuitively, in these rules you have no choice so you cannot do anything wrong applying these rules. Also, if you introduce the variable early you can make the terms in the other rules depend on this variable. In the Isar syntax the `fix` command corresponds to the metalevel quantification. Hence, if your goal is $\bigwedge x.\phi(x)$ you should start your proof by `fix x` and then show $\phi(x)$ (without meta-level quantification). Note that the `fix` command must come before the `x` occurs for the first time.

To summarize, you should apply `allI` and `exE`, the two rules that contain `!!`, as early as possible. For `allI`, this is clear: If you have a conclusion $\forall x. \ldots$, to prove, rule `allI` is the obvious choice anyway. But for `exE`, the $\exists$ occurs in the *premises*. This means that whenever you have an $\exists$ in the premises of the current subgoal, you should use the rule `exE`, and by our explanations about elimination tactic, you may guess that you should apply this rule as elimination.

Having said that, the above are rules of thumb, so you cannot expect that they always work. But they work for all exercises of this sheet.

Exercise 4 (7 points)
Try to prove the following theorems of first-order logic in Isabelle. If a theorem cannot be proven, end the proof with the command `oops` and state why this lemma cannot be proven. You might put this statement in a comment in your theory file. Comments are started by `(*` and terminated by `*)` and can span multiple lines.

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

■

The following is a theorem from lecture "First-Order Logic".

Exercise 5 (1 point)
Prove $(\forall x.A \rightarrow B(x)) \leftrightarrow (A \rightarrow \forall x.B(x))$ in Isabelle. Save it under, say, `all_scoping`. ■

We have said that in the above theorem it is crucial that A does not contain x freely. You should have a careful look at Exercise 5 (theorem `all_scoping`) again. The metavariable `?A` does not depend on `x`. This formalizes in Isabelle that A does not contain x freely. If this condition was violated, we might be tempted to prove $p(x) \rightarrow \forall x.p(x)$, which is not valid. You may want to attempt a proof of $p(x) \rightarrow \forall x.p(x)$ in Isabelle and try to understand why this cannot work.

The next exercise illustrates that whether or not a sub-formula is within the scope of a quantifier may matter even if the sub-formula does not contain that quantifier.

Exercise 6 (2 points)
Attempt to prove $((\forall x.A(x)) \to B) \to (\forall x.A(x) \to B)$ in Isabelle. Does it work? If it does not work, give (on paper) a counterexample demonstrating that the formula is not valid (i.e., sketch a signature and a structure). ■

Forward resolution In Isabelle, there are several functions that can be used to combine existing rules into a new rule. In terms of proof trees, this is best understood as collapsing a fragment from a proof tree into a single application of a rule. For example, consider the following derivation tree, which might occur in some proof tree:

$$\dfrac{\dfrac{A \wedge B}{B}\ \wedge\text{-}ER \quad \dfrac{A \wedge B}{A}\ \wedge\text{-}EL}{B \wedge A}\ \wedge\text{-}I$$

We may want to replace this by

$$\dfrac{A \wedge B}{B \wedge A}\ \wedge - symm$$

We have seen on Sheet 2 how such a rule can be derived and an identifier be bound to it, but to build big proofs, one may not want to introduce identifiers for each fragment. In Isabelle, the expression

```
conjI[OF conjunct2 conjunct1]
```

combines the rules `conjunct2`, `conjunct1`, and `conjI` into the new rule

```
[|?P2 & ?P; ?Q & ?Q1|] ==> ?P & ?Q
```

More precisely, "combining" means that the conclusions of `conjunct2` and `conjunct1` are unified with the premises of `conjI`, respectively. The result is a rule whose premises are the premises of `conjunct2` and `conjunct1`, and whose conclusion is the conclusion of `conjI`, where the unifier is applied. This process is called *(forward) resolution*.

Exercise 7 (1 point)
Prove the theorem $A \wedge B \to B \wedge A$ in Isabelle using the rule

```
conjI[OF conjunct2 conjunct1]
```

■

Generally, when we write

$$rule\ [OF\ rule\_list]\ ,$$

rule_list may have fewer elements than the number of assumptions of *rule*. Try

```
thm conjI[OF conjunct2]
```

to see what happens in this case. The rules stated in *rule_list* are applied from left to right to the premises. The previous example showed how to omit premises at the end of the list of premises. If you want to skip a premise in the beginning of the list, you can use `_` as a rule. Try

```
thm conjI[OF _ conjunct2]
```

to see what happens in this case.

Since the result of applying `OF` to a rule is another rule, you can combine the result with other rules, say, e. g., `impI`. Try

```
thm impI[OF conjI[OF _ conjunct2]]
```

to see what happens in this case.

Exercise 8 (1 point)
Prove the theorem $A \wedge B \to B \wedge A$ in Isabelle using a combination of

`conjI[OF conjunct2 conjunct1]`

and `impI`, i. e., your proof should be

`by (rule ...)`

■

Actually, using `OF`, there is a very close correspondence between proofs in Isabelle and the trees we had in paper and pencil proofs. Each new rule you obtain by an `OF` expression corresponds to a subtree. You can look at a tree and construct a corresponding `OF` expression inductively. Suppose you have a horizontal bar labeled with *rule*, and the subtrees above that line are $T_1, \ldots, T_n$, and the `OF` expressions corresponding to those subtrees are $E_1, \ldots, E_n$. Then the `OF` expression for the subtree at this horizontal bar is simply

$$rule[\texttt{OF}\ E_1\ \ldots\ E_n]$$

Exercise 9 (1 point)
Consider again the theorem $(A \to B \to C) \to (A \to B) \to (A \to C)$ on Sheet 1 Exercise 1.5, and have a look at its paper and pencil proof. Then prove the theorem in Isabelle using a single expression that combines the (six) rule applications involved using `OF`, i. e., your proof should read

`by (rule ...)`

■

Note that `OF` can be used to abbreviate proofs but has the disadvantage of making the proof unreadable. For example compare your proof of the last exercise to the proof done in Exercise 1.5. The latter proof is easier to read and understand. However, sometimes when you do proofs in Isabelle/Isar, you can sometimes use `OF` to abbreviate a proof while preserving readability. For example, assume you have $\forall x.p(x) \to q(x)$ under the name `quant` and $p(a)$ under the name `p`. Then you can derive $q(a)$ by

```
from quant p have "q(a)" by (rule mp[OF spec])
```

This does not limit readability a lot, but compresses the proof.

Explicit Instantiation We have mentioned that when you apply (forward) resolution, Isabelle *unifies* the conclusion of your current subgoal with the conclusion of the rule you use. In general, this unification is not unique, and sometimes you will have to help Isabelle to find it. For example, suppose your current goal is `p(s(s(z)))` and you want to prove this from the assumption `ALL x. p(s(x))`. Intuitively, it should be clear that you can prove this, since `p(s(x))` holds for all `x`, and thus in particular for `x` = `s(z)`. But when you try `proof (rule spec)`, Isabelle does not know that you need `x` to be `s(z)`. There are some ways to do this prove: (1) write `by (rule spec)` to let Isabelle know that she only needs additional assumption steps to discharge this goal, (2) use the method `elim` to let her try all possible unifiers, (3) use elimination resolution to reduce the number of unifiers, or (4) tell her the right unifier. To tell Isabelle about a unifier you first need to find the meta-variable you want to instantiate. Look at the rule `spec`. It contains the meta-variables `?P` and `?x`. In our case we want to instantiate `?x` with `s(z)` (since `?P` should be instantiated to `P(s(·))`). The attribute `where` can be used to modify a rule/theorem. In our example, we want to apply the rule

```
spec[where x="s(z)"]
```

Exercise 10 (1 point)
Prove the following theorem of first-order logic in Isabelle:

$$s(s(s(s(s(zero))))) = five \land q(zero) \land (\forall x.q(x) \to q(s(x))) \to q(five)$$

Did your proof need manual instantiations? If yes, can you come up with a proof that does not need manual instantiations. Hint: You can shorten your proof using `OF` to combine two frequently needed rules. ■

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 4 (12th November 2013)

Creating own shortcuts is helpful in Isabelle/Isar to abbreviate proof scripts. In the Isar syntax you have to state all premises and conclusions explicitly. While this leads to readable proofs in many cases, sometimes it is just annoying since you have to repeat the same term over and over again. The Isar syntax helps you with special constructs that allow you to create new term bindings.

The easiest way to create a new term binding is the `let` command. You simply say

```
let "?x" = "some complicated term"
```

and you can now use `?x` whenever you would have to type `some complicated term`. Note that Isabelle/Isar uses matching to detect the correct abbreviation. Hence, if you type

```
let "some ?x term" = "some complicated term"
```

the variable `x` will be bound to `complicated`, i. e., a sub-term. Also note the syntax of the let command. It has a left-hand side that should contain variables (identifiers preceeded by `?`) and a right-hand side that should not contain variables. Both the left-hand side and the right-hand side need to be in quotes while the equality sign must not.

If you have just written down a term (e. g., after stating a lemma, theorem, assumption, intermediate fact, or goal) you do not need to repeat the whole term in a `let` command to create a term binding. Isabelle/Isar lets you abbreviate these terms by appending `(is "?x")` to this term. Assume you want to show `some complicated term`. Then, writing

```
show "some complicated term" (is "some ?x term")
```

is equivalent to the two commands

```
show "some complicated term"
  let "some ?x term" = "some complicated term"
```

and, hence, produces the same binding.

Generalized Elimination Isabelle supports a more general version of elimination resolution. This can be used, e. g., to split a conjunction into its individual parts or to retrieve a witness for an existential quantifier. The keyword `obtain` signalizes such a generalized elimination. For example if you want to split the conjunction $\texttt{A} \wedge \texttt{B} \wedge \texttt{C}$ into `A` and `B` and `C`, you can use the following Isar statements

```
assume "A/\B/\C"
then obtain "A" and "B" and "C" by (elim conjE)
```

To eliminate an existential quantifier and retrieve a witness term, i. e., a term `t` that justifies $\exists \texttt{x}.\ \texttt{p}(\texttt{x})$, use `obtain` in conjunction with elimination resolution:

```
assume "EX x. p(x)"
then obtain t where "p(t)" by (rule exE)
```

How does generalized elimination work? After writing what you want to obtain Isabelle shows you a goal. For the first example, the goal is

$$\bigwedge \texttt{thesis}.\ \texttt{A} \wedge \texttt{B} \wedge \texttt{C} \Longrightarrow (\texttt{A} \Longrightarrow \texttt{B} \Longrightarrow \texttt{C} \Longrightarrow \texttt{thesis}) \Longrightarrow \texttt{thesis}$$

Using `elim conjE` will split the conjunction and produce the goal

$$\bigwedge \texttt{thesis}.\ (\texttt{A} \Longrightarrow \texttt{B} \Longrightarrow \texttt{C} \Longrightarrow \texttt{thesis}) \Longrightarrow (\texttt{A} \Longrightarrow \texttt{B} \Longrightarrow \texttt{C} \Longrightarrow \texttt{thesis})$$

which can be solved by assumption. Elimination resolution is critical in this example.

For the second example we have to show

$$\bigwedge \texttt{thesis}.\ (\exists \texttt{x}.\ \texttt{p(x)}) \Longrightarrow (\bigwedge \texttt{t}.\ \texttt{p(t)} \Longrightarrow \texttt{thesis}) \Longrightarrow \texttt{thesis}$$

Hence, without any assumption about `thesis`, assuming $\exists \texttt{x}.\ \texttt{p(x)}$ and a derivation of `thesis` from `p(t)` that is independent of `t` we have to show `thesis`. If we now use elimination resolution on the existential quantifier we get the goal

$$\bigwedge \texttt{thesis x}.\ (\bigwedge \texttt{t}.\ \texttt{p(t)} \Longrightarrow \texttt{thesis}) \Longrightarrow \texttt{p(x)} \Longrightarrow \texttt{thesis}$$

which can again be solved by assumption. Essentially, the `obtain` keyword corresponds to existential quantification on the meta-level. But we will not explain this encoding at this stage of the lecture.

Exercise 1 (7 points)
Prove the following lemmas in Isabelle:

1. $(\exists x.\ P(x) \vee Q(x)) \rightarrow (\exists x.\ P(x)) \vee (\exists x. Q(x))$
2. $(\forall x.\ \exists y.\ p(x, y)) \rightarrow (\forall u.\ \exists v.\ p(u, v))$
3. $\neg(\forall x.\ \neg p(x)) \rightarrow (\exists x.\ p(x))$
4. $(\exists x.\ p(x)) \rightarrow \neg(\forall x.\ \neg p(x))$
5. $\neg(\exists x.\ \neg p(x)) \rightarrow (\forall x.\ p(x))$
6. $(\forall x.\ p(x)) \rightarrow \neg(\exists x.\ \neg p(x))$
7. $\exists y.(\exists x. p(x)) \rightarrow p(y)$

Hint: Parts 3, 5, and 7 are classical. The other exercises can be solved without classical rules. ∎

Equality In lecture "First-Order Logic with Equality" we have learned that in *first-order logic with equality*, the predicate = is not just any predicate, but it has certain properties, namely it is an *equivalence* relation and a *congruence* on all terms and relations.

Note that we have no special theory file for first-order logic with equality, since it is already included in `IFOL.thy`. There, all we have is

```
refl:          "a=a" and
subst:         "a=b \<Longrightarrow> P(a) \<Longrightarrow> P(b)"
```

In fact, it turns out that transitivity and symmetry can be derived from reflexivity and congruence. Actually, even `subst` is a derived rule, but at this stage of the course we do not explain how.

Exercise 2 (4 points)
Derive rules that state that = is *symmetric* and *transitive*:

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

Do it both in Isabelle and using paper and pencil (here you may use the reflexivity axiom and the congruence rules from the lecture).

Hint: In the rule `subst`, `P(b)` stands for any formula that contains `b`; in particular, $\texttt{a} = \texttt{b}$ is such a formula. Both proofs are extremely short (not more than 5 lines). ∎

Equational Proofs In lecture "First-Order Theories" we have seen two examples of equational proofs. Isabelle/Isar provides special support for equational proofs through a set of keywords. These keywords maintain a special *calculation* that might finally hold the desired lemma.

The first important keyword is `also`. When this keyword is first encountered in a proof, it remembers the current fact as calculation. Every further occurrence of `also` composes the current calculation with the last fact using one *transitivity rule*. These rules are marked with the attribute `trans` and stored specially by Isabelle. You can view the set of transitivity rules with the command `print_trans_rules`. Note that *trans* is one of them.

After you have shown the last step of your calculation, the keyword `finally` does a transitivity step with the previous calculation and provides this fact as assumption.

With these keywords we can prove $\texttt{a} = \texttt{b} \Longrightarrow \texttt{b} = \texttt{c} \Longrightarrow \texttt{c} = \texttt{d} \Longrightarrow \texttt{a} = \texttt{d}$ as follows:

```
theorem "a=b ==> b=c ==> c=d ==> a=d"
proof -
  assume "a=b"
  also
  assume "b=c"
  also
  assume "c=d"
  finally
  show "a=d" .
qed
```

Placing the cursor after `also` resp. `finally` will display the current calculation.

When doing equational proofs on pen and paper we usually omit the individual transitivity steps by chaining equalities. Isabelle/Isar supports this syntax by proving the abbreviation ... for the right hand side of the last calculation. Hence, in the proof above, we could also write `assume ''...=d''` instead of `assume ''c=d''`.

Groups We will now have some exercises on groups. These exercises use the theory file `Groups.thy` available from the lecture page. This theory file provides an extension of first order logic called *group*. To use this extension in your theory you (1) have to import `Groups.thy` and (2) have to use `context group begin` and `end` to inform Isabelle that the lemmas and theorems proven between `begin` and `end` are only valid in `group`.

Now have a look at the file `Groups.thy`. It defines a function `f` (which can be written as `*` in Isabelle), a constant `e` (written as 1), and the function `inv` (written as `^-1`). Furthermore, it assumes `*` to be associative, 1 to be the right neutral element of `*`, and `^-1` to be the right inverse of `*`.

Additionally, it defines some lemmas that can be used to rewrite terms. The lemma `assoc_context` for example justifies a rewrite using associativity of `*` even if the term to rewrite is only a sub-term of the goal. Similarly, `rneutr_context` and `rinv_context` can be used to rewrite terms where the left hand side of an equality contains a sub-term of the form `a*1` resp. `a*a^-1` into `a` resp. `1`. Such rules are called *congruence rules*.

Since sometimes the term to rewrite is not contained in the left hand side, but in the right hand side, `Groups.thy` also provides symmetric version of these rules. They have the suffix `_sym` and are simply created by adding the attribute `symmetric` to the original rules.

Exercise 3 (4 points)

1. Show that `^-1` is left inverse, i. e., `x^-1 * x = 1`.
2. Show that `1` is also the left neutral element, i. e., `1 * x = x`.

   Hint: You might want to create a congruence rule from Part 1 to simplify this part.
3. Show that the inverse element is unique, i. e., `x*y=1 ==> y=x^-1`.
4. Show `(a*b)^-1 = b^-1*a^-1`.

■

A Groups.thy

```
1  theory Groups
2  imports FOL
3  begin
4
5  locale group =
6    fixes f :: "' a ⇒'a ⇒'a" ( infixl "∗" 70)
7    fixes e :: ' a ("1")
8    fixes inv :: "' a ⇒'a" ("(_^−1)" [79] 80)
9    assumes assoc : "a ∗ b ∗ c = a ∗ (b ∗ c)"
10   assumes rneutr: "a ∗ 1 = a"
11   assumes rinv: "a ∗ a^−1 = 1"
12
13 begin
14
15 (∗ This does not work as expected in jedit
16   notation
17       inv ("(_\<inverse>)" [79] 80)
18 ∗)
19
20   lemmas assoc_context = subst_context[OF assoc]
21     and rneutr_context = subst_context[OF rneutr]
22     and rinv_context = subst_context[OF rinv]
23
24   lemmas assoc_context_sym = assoc_context[symmetric]
25     and rneutr_context_sym = rneutr_context[symmetric]
26     and rinv_context_sym = rinv_context[symmetric]
27
28 end
29
30 end
```

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 5 (19th November 2013)

This week we will be using several Isabelle theories. You should have theories named `sheet05_NSet.thy` and `sheet05_lambda.thy`.

Naïve Set Theory has been formalized in Isabelle in `NSet.thy`. Here are some notes on the syntax (if this list is incomplete, you may have a look at `NSet.thy`):

Usual math. notation	Isabelle
$\{1, 2, 3\}$	`{1,2,3}`
$\in$	`:`
$\notin$	`~:`
$\{y \mid P(y)\}$	`{y. P(y)}`
$A \cap B$	`A Int B`
$A \cup B$	`A Un B`
$A \subseteq B$	`A <= B`
$A \setminus B$	`A Minus B`
$\mathcal{P}(A)$	`Pow(A)`

Have a look at the rules from the lecture on "Naïve Set Theory". In `NSet.thy`, instead of those inference rules we have two axioms `ext` and `Collect`. Load `NSet` (with `import`) in `sheet05_NSet.thy`. You might want to take a look at the lemmas proven in `NSet.thy`.

Exercise 1 (3 points)
Prove in Isabelle that the subset relation is a partial order, i.e., it is reflexive, transitive and antisymmetric. ■

Exercise 2 (1 point)
Prove $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ in Isabelle.

■

Exercise 3 (1 point)
Prove $A \subseteq B \leftrightarrow \mathcal{P}(A) \subseteq \mathcal{P}(B)$ in Isabelle

■

Exercise 4 (2 points)
Show that `NSet` is inconsistent, i.e., that $\bot$ can be derived in it.

Hint: recall lecture "Naïve Set Theory". You should start like this:

```
lemma "False"
proof (rule notE[where P="{A. A ~: A} ~: {A. A ~: A}"])
```

Our version of naïve set theory is based on `IFOL`. Inconsistency can still be shown. If, however, you prefer classical reasoning, import `FOL` and do your prove.

Hint: The rule *classical_dual* from Exercise 4 on Sheet 2 will be useful. ■

Untyped λ-calculus Recall that the syntax of the untyped λ-calculus is defined by the following grammar:

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

We introduced conventions of left-associativity and iterated λ's in order to avoid cluttering the notation.

Exercise 5 (2 points)
Write out the following λ-terms with full bracketing and without iterated λ's:

1. $(\lambda xyz.\, xz(yz))(\lambda xy.\, x)(\lambda xyz.\, x(zy)z)$
2. $(\lambda u.\, uu)(\lambda xz.\, (\lambda v.\, x)(\lambda xy.\, xy))(c(\lambda w.\, wc))$

■

Exercise 6 (2 points)
Rewrite the following λ-term using left-associativity and iterated λ's:

1. $((\lambda x.\, (\lambda y.\, (\lambda z.\, (z(yz)(w(xz)))))))(\lambda x.\, (xx)))$
2. $((\lambda x.\, ((\lambda w.\, v)(\lambda y.\, (yz))))((\lambda z.\, x)(\lambda x.\, (\lambda z.\, (\lambda u.\, (((xy)z)(uz)))))))$

■

In a λ-term, a subterm of the form $(\lambda x.\, M)N$ is called a *redex* (plur. *redices*). It is a subterm to which β-reduction can be applied.

Exercise 7 (3 points)
Reduce the terms $(\lambda xy.\, (\lambda z.\, wz)(\lambda w.\, w)x)y$, $(\lambda x.\, yx)(\lambda y.\, y)$, and $(\lambda xyz.\, zyx)(\lambda x.\, x)((\lambda v.\, x)c)$ to β-normal form (on paper), underlining the redex in each step. ■

In Isabelle, the untyped λ-calculus has been implemented in `lambda.thy` (for β-reduction and β-conversion).

Concerning the syntax, λ is written `lam`, and application MN is written `M^N`. Reduction is denoted by `>-->` (note that `jedit` might replace this with $>\longrightarrow$ which is not what we want). Conversion is denoted by `>=<`. Moreover, for some frequently used λ-terms called *combinators* [1, chapter 2], letters are introduced as a means of abbreviation: $I \equiv \lambda x.\, x$, $K \equiv \lambda xy.\, x$, $S \equiv \lambda xyz.\, xz(yz)$, $Y_C \equiv \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$, $Y_T \equiv (\lambda zx.\, x(zzx))(\lambda zx.\, x(zzx))$.

The reduction calculus is implemented in the context `RED`, the conversion calculus in the context `CONV`. Both provide the basic rules and the combinators mentioned above. Make sure you put the exercises done in `RED` between `context RED begin` and `end` and the exercises done in `CONV` between `context CONV begin` and `end`.

Schematic lemmas When doing β-reductions you typically do not know what the resulting term will be. In this case you would like to use a schematic variable like `?x` to capture the result value. Isabelle however does not let you state a lemma containing a variable for obvious reasons. However, the Isabelle/Isar syntax contains the keyword `schematic_lemma` which allows you to state lemmas containing schematic variables. For example, we could write

```
schematic_lemma "(lam x y. x ^ y) ^ (lam x. x) ^ z >--> ?x"
```

if we did not know the β-normal form. Note that there is an easy proof using reflexivity, but this clearly does not produce β-normal form! Hence, we have to do a series of derivation steps. These can be done with equational reasoning. Note that neither `RED` nor `CONV` provides rules to substitute a sub-term. Instead you have to use the congruence axioms of these theories. But sometimes isolating the redex can be done with a combination of multiple axioms using `OF`. An example β-reduction proof of the above theorem is

```
proof -
  have "(lam x y. x ^ y) ^ (lam x. x) ^ z >-->
        (lam y. (lam x. x) ^ y) ^ z" by (rule appr[OF beta])
  also have "... >--> (lam x. x) ^ z" by (rule beta)
```

```
  also have "... >--> z" by (rule beta)
  finally show "(lam x y. x ^ y) ^ (lam x. x) ^ z >--> z" .
qed
```

Import `lambda` in `sheet5_lambda.thy`. The following exercises will be in the context `RED`.

Exercise 8 (3 points)
Reduce the following terms to β-normal form in Isabelle:

1. SKK
2. SKS
3. SK

Your result should only contain lambdas and combinators, but no redices.

Hint: You should use schematic lemmas. In the end, the metavariable used in your schematic lemma should be instantiated to a term in β-normal form. Use `fold` to turn a lambda-term back into the corresponding combinator. ■

Exercise 9 (2 points)

1. Prove $Y_T F$>-->$F(Y_T F)$.
2. Try to prove $Y_C F$>-->$F(Y_C F)$. Does it work? If not state at which point you stopped and why you stopped there.

■

Exercise 10 (2 points)
Prove the following goals in the context `CONV`:

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

■

References

[1] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1990.

A Encoding of naïve set theory

```
theory NSet
imports IFOL
begin

declare [[ eta_contract = false ]]

typedecl i

type_synonym set = i

 arities  i  :: " term"

consts
   "zero " ::  i  ("0")
   "one " ::  i  ("1")
   "empty" :: set  ("{}")
    insert  :: "[ i , set ]⇒set"
   "member" :: "[i, set ]⇒o"
```

```
  "lesseq " :: "[ 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 "\<inter>" 70)
  "Un" :: "[set , set ]⇒set" (infixl "\<union>" 65)
  Compl :: "set⇒set"
  Pow :: "set⇒set"
  UNIV :: set
  Ball  :: "[ set , i⇒o]⇒o"
  Bex  :: "[set , i⇒o]⇒o"

notation
  member ("op :") and
  member ("(_/ : _)" [51,51] 50)

notation (xsymbols)
  member     ("op \<in>") and
  member     ("(_/ \<in> _)" [51, 51] 50)

notation (HTML output)
  member     ("op \<in>") and
  member     ("(_/ \<in> _)" [51, 51] 50)

syntax
  "_Coll " :: " pttrn => o => set"   ("(1{_./ _})")
translations
  "{x. P}" == "CONST Collect (%x. P)"

hide_const (open) member

syntax
  "_Ball"         :: " pttrn => set => o => o"    ("(3ALL _:_./ _) " [0, 0, 10] 10)
  "_Bex"          :: " pttrn => set => o => o"    ("(3EX _:_./ _) " [0, 0, 10] 10)

syntax (HOL)
  "_Ball"         :: " pttrn => set => o => o"    ("(3! _:_./ _) " [0, 0, 10] 10)
  "_Bex"          :: " pttrn => set => o => o"    ("(3? _:_./ _) " [0, 0, 10] 10)

syntax (xsymbols)
  "_Ball"         :: " pttrn => set => o => o"    ("(3∀_\<in>_./ _) " [0, 0, 10] 10)
  "_Bex"          :: " pttrn => set => o => o"    ("(3∃_\<in>_./ _) " [0, 0, 10] 10)

syntax (HTML output)
  "_Ball"         :: " pttrn => set => o => o"    ("(3∀_\<in>_./ _) " [0, 0, 10] 10)
  "_Bex"          :: " pttrn => set => o => o"    ("(3∃_\<in>_./ _) " [0, 0, 10] 10)

translations
  "ALL x:A. P" == "CONST Ball(A (%x. P))"
  "EX x:A. P" == "CONST Bex(A (%x. P))"

syntax
  "op  ~:" :: "[ i,set ]⇒o" ("(_/ ~: _) " [50, 51] 50)
(*  "@Collect " :: "[pttrn , o]⇒set" ("(1{_./ _})") *)
  "@Finset" :: "args ⇒set" ("{(_)}")
(*  "op Int " :: "[ set , set ]⇒set" (infixl "\<inter>" 70)
  "op Un" :: "[set , set ]⇒set" (infixl "\<union>" 65)*)
  "@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
  "NSet.UNIV" == "NSet.Compl({})"
  "x~: y" == "(x : y)"
  "{x, xs}" == "NSet.insert(x,{xs})"
  "{x}" == "NSet.insert(x,{})"
  "{x. P}" == "NSet.Collect(\<lambda>x. P)"
  "INT x:A. B" == "NSet.INTER(A, (\<lambda>x. B))"
  "UN x:A. B" == "NSet.UNION(A, (\<lambda>x. B))"
  "ALL x:A. P" == "CONST NSet.Ball(A, (\<lambda>x. P))"
  "EX x:A. P" == "CONST NSet.Bex(A, (\<lambda>x. P))"

```

```
93 axiomatization where
94   ext : " A = B\<longleftrightarrow>(∀x. ((x:A)\<longleftrightarrow>(x:B)))" and
95   Collect : "( t :{ x . P(x)})\<longleftrightarrow>(P(t))"
96
97 syntax ("" output)
98   "  _setle   " :: "[ set , set ]⇒o" ("op <=")
99   "  _setle   " :: "[ set , set ]⇒o" ("(_/ <= _)" [50, 51] 50)
100  "  _setless   " :: "[ set , set ]⇒o" ("op <")
101  "  _setless   " :: "[ set , set ]⇒o" ("(_/ < _)" [50, 51] 50)
102
103 syntax (xsymbols)
104  "  _setle   " :: "[ set , set ]⇒o" ("op \<subseteq>")
105  "  _setle   " :: "[ set , set ]⇒o" ("(_/ \<subseteq> _)" [50, 51] 50)
106  "  _setless   " :: "[ set , set ]⇒o" ("op \<subset>")
107  "  _setless   " :: "[ set , set ]⇒o" ("(_/ \<subset> _)" [50, 51] 50)
108  "op  ~:" :: "[' a , set ]⇒o" ("op \<notin>")
109  "op  ~:" :: "[' a , set ]⇒o" ("(_/ \<notin> _)" [50, 51] 50)
110
111 translations
112   "op \<subseteq>" == "op <= :: [set,set] ⇒o"
113
114 defs
115   subset_def : " A <= B≡∀x. x\<in>A ⟶ x\<in>B"
116   empty_def: "{} ≡{x. False}"
117   Minus_def: "A − B≡{x. x\<in>A ∧ x\<notin>B}"
118   Un_def: "A \<union> B≡{x. x\<in>A ∨ x\<in>B}"
119   Int_def  : " A \<inter> B≡{x. x\<in>A ∧ x\<in>B}"
120   Ball_def : " Ball(A, P)≡(∀x. x\<in>A ⟶ P(x))"
121   Bex_def: "Bex(A, P)≡(∃x. x\<in>A ∧ P(x))"
122   Compl_def: "Compl(A)≡{x.x\<in>A}"
123   INTER_def: "INTER(A, B)≡{y. ALL x:A. y\<in>B(x)}"
124   UNION_def: "UNION(A, B)≡ {y. EX x:A. y\<in>B(x)}"
125   insert_def  : " insert (a , B)≡{x. x=a} \<union> B"
126   Pow_def: "Pow(A)≡{B. B<= A}"
127
128 lemma inI: "(P(t)) ⟹ (t\<in>{x. P(x)})"
129 proof (rule iffD2)
130   show "t \<in> {x. P(x)} \<longleftrightarrow> P(t)" by (rule Collect)
131 qed
132
133 lemma "inE2": "(t\<in>{x. P(x)}) ⟹ P(t)"
134 proof (rule iffD1)
135   show "t \<in> {x. P(x)} \<longleftrightarrow> P(t)" by (rule Collect)
136 qed
137
138 lemma inE:
139 assumes p1: "(t\<in>{x. P(x)})"
140 assumes p2: "P(t) ⟹ R"
141 shows "R"
142 proof (rule p2)
143   from p1 show "P(t)" by (rule inE2)
144 qed
145
146 lemma equalsI: "(∀ x. x\<in>A\<longleftrightarrow>x\<in>B) ⟹ A = B"
147 proof (rule iffD2)
148   show "A=B \<longleftrightarrow> (∀x. x \<in> A \<longleftrightarrow> x \<in> B)" by (rule ext)
149 qed
150
151 lemma equalsE2: "A = B ⟹ (∀ x. x\<in>A\<longleftrightarrow>x\<in>B)"
152 proof (rule iffD1)
153   show "A=B \<longleftrightarrow> (∀x. x \<in> A \<longleftrightarrow> x \<in> B)" by (rule ext)
154 qed
155
156 lemma equalsE:
157 assumes p1: "A = B"
158 assumes p2: "(∀ x. x\<in>A\<longleftrightarrow>x\<in>B) ⟹ R"
159 shows "R"
160 proof (rule p2)
161   from p1 show "∀x. x \<in> A \<longleftrightarrow> x \<in> B" by (rule equalsE2)
162 qed
163
164 end
```

B Encoding of untyped lambda calculus

```
theory lambda
imports Pure
begin

setup Pure_Thy.old_appl_syntax_setup

typedecl "term"

lemma [trans]: assumes st: "s == t"
  shows "PROP P(t) ==> PROP P(s)"
  by (unfold st)

lemma [trans]:
  assumes ps: "PROP P(s)"
  assumes st: "s == t"
  shows "PROP P(t)"
  using ps by (unfold st)

locale RED =
fixes abs :: "[ term ⇒term] ⇒term" (binder "lam " 10)
fixes "apply" :: "[term, term] ⇒term" (infixl "^" 20)
fixes K   :: " term"
fixes I   :: " term"
fixes S   :: " term"
fixes B   :: " term"
fixes YC  :: " term"
fixes YT  :: " term"
fixes Red :: "[term, term] ⇒prop" ("(_ >--> _)")
assumes K_def : "K ≡lam x. (lam y. x)"
assumes I_def : " I ≡lam x. x"
assumes S_def : "S ≡lam x. (lam y. (lam z. x^z^(y^z)))"
assumes B_def : "B ≡S^(K^S)^K"
assumes YC_def: "YC ≡lam f. ((lam x. f^(x^x))^(lam x. f^(x^x)))"
assumes YT_def: "YT ≡(lam z. lam x. x^(z^z^x))^(lam z. lam x. x^(z^z^x))"
assumes beta : "(lam x. f (x))^a >--> f(a)"
assumes refl : " M >--> M"
assumes trans[trans]: "⟦ M >--> N; N >--> L ⟧ ⟹ M >--> L"
assumes appr : "M >--> N  ⟹ M^Z >--> N^Z"
assumes appl : "M >--> N  ⟹ Z^M >--> Z^N"
assumes epsi : "⟦ !! x. P(x) >--> Q(x) ⟧ ⟹ (lam x. P(x)) >--> (lam x. Q(x))"
begin
(*
notation (output)
  "abs" (binder "\<lambda> " 10)
*)
end

locale CONV = RED +
fixes Conv :: "[term, term] ⇒prop" ("(_ >=< _)")
assumes beta : "(lam x. f (x))^a >=< f(a)"
assumes refl : " M >=< M"
assumes symm[sym] : "M >=< N  ⟹ N >=< M"
assumes trans[trans]: "⟦ M >=< N; N >=< L ⟧ ⟹ M >=< L"
assumes appr : "M >=< N  ⟹ M^Z >=< N^Z"
assumes appl : "M >=< N  ⟹ Z^M >=< Z^N"
assumes epsi : "⟦ !! x. P(x) >=< Q(x) ⟧ ⟹ lam x. P(x) >=< lam x. Q(x)"
begin
lemmas beta_red = RED.beta[OF RED_axioms] and
        appr_red = RED.appr[OF RED_axioms] and
        appl_red = RED.appl[OF RED_axioms] and
        epsi_red = RED.epsi[OF RED_axioms] and
        trans_red = RED.trans[OF RED_axioms] and
        refl_red  = RED.refl[OF RED_axioms]
end

end
```

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 6 (26th November 2013)

This week we will be using several Isabelle theories. You should have two proof scripts named `sheet06_CNUM.thy` and `sheet06_FOL.thy`

Turing-Completeness In lecture "The λ-Calculus", it was said that the untyped λ-calculus is Turing-complete. This is usually shown not by mimicking a Turing machine in the λ-calculus, but rather by exploiting the fact that the Turing computable functions are the same class as the *μ-recursive* functions [1, chapter 4]. In a lecture on theory of computation, you have probably learned that the μ-recursive functions are obtained from the *primitive recursive* functions by so-called *unbounded minimalization*, while the primitive recursive functions are built from the 0-place zero function, projection functions and the successor function using composition and primitive recursion [2].

The proof that the untyped λ-calculus can compute all μ-recursive functions is thus based on showing that each of the abovementioned ingredients can be encoded in the untyped λ-calculus. While we are not going to study this, one crucial point is that it should be possible to encode the natural numbers and the arithmetic operations in the untyped λ-calculus.

Such an encoding has been proposed by Alonzo Church. Here the number n is encoded as the term $\lambda fx.\ \underbrace{f(f\ldots(f}_{n \text{ times}}x)\cdots)$, which we abbreviate by writing $\lambda fx.\ f^n x$. The successor function and addition are given by the λ-terms:

$$\begin{aligned} succ &\equiv \lambda ufx.\ f(ufx) \\ add &\equiv \lambda uvfx.\ uf(vfx) \end{aligned}$$

Exercise 1 (2 points)
Prove (on paper, not in Isabelle) that *succ* and *add* are indeed the successor and addition function, by evaluating them symbolically (i.e, on "terms" $\lambda fx.\ f^n x$ and $\lambda fx.\ f^m x$). ∎

The encoding of the natural numbers proposed by Alonzo Church is implemented in the theory `church.thy` in the context `CNUM`. It requires the theory `lambda.thy` from the previous exercise sheet. In `church.thy`, you will also find abbreviations $C_0, \ldots$ for some small numbers.

Exercise 2 (2 points)
Reduce the following terms in `CNUM`:

1. *succ* C_1
2. *add* C_2 C_3

∎

Exercise 3 (4 points)
The existence of the fixpoint combinator Y_T (Exer. 9 on Sheet 5) seems to suggest that every function has a fixpoint [1, chapter 3]. This exercise tries to shed some light on this strange impression.

1. From what you remember from any math course that you took: does every function have a fixpoint?
2. Is a Church numeral a λ-term in β-normal form?

3. Try to reduce (on paper) the term Y_T *succ* to normal form. To avoid getting terms that are too complicated, follow two strategies:

   - replace a combinator by its definition only if that allows you to do a β-reduction. After each step, simplify the term by replacing $(\lambda zx.\, x(zzx))(\lambda zx.\, x(zzx))$ with Y_T and $\lambda ufx.\, f(ufx)$ with *succ*.
   - whenever several reduction steps are possible, avoid steps for which it is obvious that they will lead to divergence.

   What do you observe? Compare this to point 1.

4. The function that always returns 0 is encoded as $Z \equiv \lambda u.(\lambda fx.\, x)$. Try to reduce (on paper) the term $Y_T\ Z$ to normal form. What do you observe this time?

■

The simply-typed λ-calculus We now do some paper-and-pencil exercises on the simply-typed λ-calculus.

Exercise 4 (1 point)
Why is it a useful convention that function applications associate to the left whereas types associate to the right? ■

Exercise 5 (1 point)
Derive (on paper) a type judgement for the term $\lambda fghx.\, f(g(h\,x)\,x)\,x$ (including inserting the appropriate type superscripts for the variables f, g, h, x you bind with λ), as this has been done for $\lambda fx.\, f\,x\,x$ in lecture "The λ-Calculus". ■

Polymorphism We now have an exercise where a type judgement in the λ-calculus with polymorphism must be derived.

Exercise 6 (2 points)
Let $\mathcal{B} = \{bool/0, \mathbb{N}/0, pair/2\}$ (e.g., *pair* has arity 2) and $\Sigma = \langle \top : bool,\ 3 : \mathbb{N},\ 4 : \mathbb{N},\ E : \alpha \to \alpha \to bool,\ P : \alpha \to \beta \to (\alpha, \beta)\ pair \rangle$.
Derive (using paper and pencil) an appropriate type judgement for

1. $(E\ \top\ (E\ 3\ 4))$.
2. $P\ 4\ (P\ 3\ \top)$.

■

Assumptions in lemmas Isabelle/Isar has two different ways to state lemmas with assumption. Until now we have always written the assumption in the lemma and assumed them in the proof, e. g.,

```
lemma "x=y==>y=x"
proof -
  assume xy: "x=y"
```

Alternatively, you can explicitly name the assumption when stating the lemma and use the name just like you use assumptions, e. g.

```
lemma assumes xy: "x=y" shows "y=x"
```

Higher-Order Unification We have seen many times by now that applying tactics in Isabelle usually involves unification. In the following, assume that any object term or formula is (represented by) a λ-term, so the variables of the λ-calculus will be called *metavariables*, as explained in lecture "Encoding Syntax".

Often Isabelle will not immediately find the appropriate unifier. An example of this was the derived rule

```
"x=y ==> y=x"
```

The paper and pencil proof of it is

$$\dfrac{x = y \quad \dfrac{}{x = x}\ \texttt{refl}}{y = x}\ \texttt{subst}$$

but when you try to do this proof in Isabelle starting from the bottom using rule `subst`, the problem is to make her know that the metavariable ?`b` contained in `subst` should be `y`. The unification problem

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

has solutions

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

If you have ever programmed in Prolog, or have taken a course on first-order logic, you may remember that in that context, every unification problem that has a solution has a unique (up to variable renaming) solution, called the *most general unifier*. So what is different here? The difference is that the variables may assume functions as values, such as λ`z. z = y` above. We also speak of *higher-order* variables. This is why we speak of *higher-order unification*.

There are several ways of helping Isabelle to find the right unifier, including:

- Giving some premises as facts using `from`.
- Using `where` or `of`.
- Using `OF` or `THEN`: Can you explain intuitively why these are useful for finding the right unifier?
- `back`: Whenever Isabelle is at a point where the result of applying a tactic is non-unique (e.g. because the unification is not unique), she creates a *branch point*, indicating that there are several possibilities of action. She will try the first possibility, but you can call the function `back` to go to the most recent branch point and try the next possibility that has not been tried yet.

Exercise 7 (4 points)
Derive (in `FOL`)

$$\dfrac{x = y}{y = x}\ sym$$

using

1. `from`. Note that you should use `-` for the initial proof step.
2. `where`, where you instantiate the metavariable ?`P` occurring in rule `subst`. In this context, note that in Isabelle, λ is written `%`.
3. `OF`;
4. `back` (Note that this is not a good Isar style)
5. `THEN`.

■

Encoding Propositional Logic in $\lambda^{\to}$ We have seen in lecture "Encoding Syntax" that $\lambda^{\to}$ can be used to encode propositional logic. To this end, we introduced constants *not*, *and*, *imp* (this list could easily be extended), i.e. we introduced a signature

$$\Sigma = \langle not : o \to o, and : o \to o \to o, imp : o \to o \to o \rangle.$$

The propositional variables were typed by a context Γ. This Γ will be different for each formula we want to encode.

Exercise 8 (1 point)
Using paper and pencil, encode $a \wedge \neg b$ in $\lambda^{\to}$ and give the proof tree of the judgement $\Gamma \vdash \ulcorner a \wedge \neg b \urcorner : o$, as this has been done for $\neg a \to a$ in lecture "Encoding Syntax". What is an appropriate Γ? ■

References

[1] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1990.

[2] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 7 (3rd December 2013)

This week we will not do any Isabelle exercises, but concentrate on the metalogic. Hence, all exercises should be done on paper.

Encoding First-Order Logic in $\lambda^{\rightarrow}$

Exercise 1 (1 point)
Using paper and pencil, encode $\forall x.\exists y.p(x, f(y))$ in $\lambda^{\rightarrow}$ and give the proof tree of the judgement $\ldots \vdash \ulcorner \forall x.\exists y.p(x, f(y)) \urcorner : o$. ■

I also suggest you have a look at `IFOL.thy` to see how the concepts of lecture "Encoding Syntax" are implemented. The keyword `axiomatization` precedes some declarations of constants such as `False`, and `All`.

An alternative way to declare constants is shown in our encoding of the untyped lambda calculus `lambda.thy`. There, we have a local theory where the constants are defined using the keyword `fixes`.

Exercise 2 (1 point)
Encode `(lam x y. y)^(lam x. x) >--> (lam x. x)` and derive the type of this expression in `RED`. ■

Exercise 3 (1 point)
What is the order of the encoding of the untyped lambda calculus in `lambda.thy`. Justify your answer. ■

Exercise 4 (1 point)
In several exercises we have used the equivalence between `lam x. t x` and `lam y. t y` even though we do not have α-conversion in `RED` or `CONV`. Explain, based on the encoding of the untyped lambda calculus why this equivalence holds. ■

Metalogic

Exercise 5 (1 point)
In the lecture we have seen how existential elimination is expressed in Isabelle's metalogic. Explain how the side condition of `exE` is expressed in the encoding. In particular state what happens if R is instantiated with $Q\,x$. ■

Exercise 6 (2 points)
Derive on paper transitivity of $\equiv$, i. e., derive $\bigwedge s\ t\ u.\ s \equiv t \Rightarrow t \equiv u \Rightarrow s \equiv u$. ■

Exercise 7 (3 points)
In the lecture we have seen how to derive $\exists$-E if we consider $\exists x.\ P(x)$ as abbreviation for $\neg\forall x.\ \neg P(x)$. Derive this proof in Isabelle's metalogic.

Hint: Landscape paper might be helpful. ■

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 8 (10th December 2013)

This week we will connect the Isabelle/Isar language to Isabelle's metalogic. All exercises should be done with pen and paper.

FOL-rules in metalogic The rules we used so far are either axioms or rules derived from these axioms. In metalogic, a rule is a fully quantified statement. Here are some of the rules we used so far together with their metalogic encoding.

Rule	Metalogic encoding
$\wedge$-*I*	$\bigwedge P\,Q.\ P \Rightarrow Q \Rightarrow P \wedge Q$
$\vee$-*IL*	$\bigwedge P\,Q.\ P \Rightarrow P \vee Q$
$\vee$-*IR*	$\bigwedge P\,Q.\ Q \Rightarrow P \vee Q$
$\vee$-*E*	$\bigwedge P\,Q\,R.\ P \vee Q \Rightarrow (P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R$
$\rightarrow$-*I*	$\bigwedge P\,Q.\ (P \Rightarrow Q) \Rightarrow P \rightarrow Q$
$\exists$-*E*	$\bigwedge P\,R.\ (\exists x.Px) \Rightarrow (\bigwedge x.\ Px \Rightarrow R) \Rightarrow R$
conjE	$\bigwedge P\,Q\,R.\ P \wedge Q \Rightarrow (P \Rightarrow Q \Rightarrow R) \Rightarrow R$
RAA	$\bigwedge P\ (\neg P \Rightarrow \bot) \Rightarrow P$

Exercise 1 (1 point)
In the Lecture on the Isar language we have seen that a rule application corresponds to multiple applications of rules at the meta level. Give a proof on the meta-level for the following Isar statements:

```
1 assume "P" "Q"
2 then have "P ∧ Q" by (rule conjI)
```

■

Exercise 2 (1 point)
Translate the following Isar script into metalogic, i. e., derive the proof state after the last line.

```
1 {
2   fix x
3   assume "P(x)"
4   hence "P(x) ∨ Q(x)" by (rule disjI1)
5 }
```

■

Exercise 3 (2 points)
Translate the following proofs into metalogic.

1. Existential elimination using obtain:

```
1 from ‘∃ x. P(x)‘
2 obtain t where "P(t)" proof (rule exE) qed
```

2. Splitting of a conjunction into its parts:

```
1 from ‘P ∧ Q‘ obtain "P" and "Q" proof (rule conjE) qed
```

■

Exercise 4 (2 points)
Consider the following proof script.

```
1 lemma "∀ y. ∃x. p(x,y) ⟹ ∃x. ∀y. p(x,y)"
2 proof (intro exI allI )
3  fix y
4  assume "∀ y. ∃ x. p(x,y)"
5  hence "∃ x. p(x,y)" by (rule allE)
6  then obtain x where "p(x,y)" by (rule exE)
7  then show "p(x,y)" .
8 qed
```

Explain why this proof script fails. ■

Exercise 5 (4 points)
Transform the following Isar proof script into metalogic.

```
1 lemma orcomm: "P ∨ Q ⟶ Q ∨ P"
2 proof (rule impI)
3   assume "P ∨ Q"
4   then show "Q ∨ P"
5   proof (rule disjE)
6     assume "P" thus "Q ∨ P" proof (rule disjI2) qed
7     next
8     assume "Q" thus "Q ∨ P" proof (rule disjI1) qed
9   qed
10 qed
```

■

Folding and unfolding We have seen at a very early stage of the course that several constructs are "syntactic sugar". To deal with these constructs we used the proof prefix unfolding or the proof methods `fold` or `unfold`. These methods take meta-level equalities and transform the proof state.

Exercise 6 (1 point)
Explain how `fold`, `unfold`, and unfolding can be reflected in the metalogic. ■

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 9 (17th December 2013)

Proof automation To study proof automation in Isabelle we consider a simply typed set theory that is similar to naïve set theory studied on sheet 5. But this theory distinguishes between the types *element* (type `e` in the Isabelle theory) and the type *set* (type `s`). These types cannot be the same. Hence, this theory trivially prevents Russell's paradox.

The theory file already has some introduction, destruction, and elimination rules proven. But there are still some operators left. On this sheet, we will derive introduction and elimination rules for most of them. The ultimate goal is to be able to proof all the simple lemmas on this sheet by one call to Isabelle's proof tools `fast`, `blast`, or `clarify`.

Isabelle's automatic proof tools differ in their proof search strategies. In general, you should try this procedure:

1. Try `blast`.
2. If `blast` seems to run endlessly, try `fast`.
3. If `fast` does not terminate or either `blast` or `fast` yields an error, try `clarify`.

Note that `clarify` will quite likely not solve your goal directly, but apply rules until either no rule is applicable anymore, or a rule that generates more than one sub-goal has to be applied, or backtracking is needed. Hence, `clarify` can be used to advance your prove to the next state where you either missed a rule, or need a case split. If you miss a rule, proof that rule, add it to the `claset` of the current context, and continue with your proof. If `clarify` stopped because the next rule would generate more sub-goals, perform the next step "by hand" and continue with `clarify`. Ultimately, your proof should be done and all your sub-goals should be proven by `clarify`. Then, all you need to do is to check that all splitting rules are contained in the `claset` and you can simplify the proof to one invocation of `fast`.

So how can we manage the `claset`? Use the command `print_claset` to see what is contained in the current `claset`. You can add rules to the current `claset` by adding the attributes `intro` or `elim` to the lemma. If you want to redeclare a lemma with different attributes, use `lemmas`. Note that by default all rules are treated as unsafe. Thus, automated proof tools generate a backtracking point when applying these rules. This is needed since the proof might become impossible after a rule application. Then, the automated tool has to revert this application and try something different. If you are sure it is safe to apply a rule you should a `!` to the attribute. This classifies the rule as safe and speeds up proof search. Be careful. If you declare a rule as safe but the rule is actually unsafe, you might end up with a failed proof even though a proof is possible.

Sometimes it is not necessary to add all the rules to the `claset` of the current context. Instead, we can give a modification of the `claset` for every invocation of an automated tool. Isabelle/Isar allows us to simply add new introduction and elimination rules when calling a tool by appending `intro:` and/or `elim:` to the call. Hence, if we want to add a rule `newE` as elimination rule and a rule `newSafeI` as safe introduction rule to an invocation of `fast` we would write

```
fast elim: newE intro!: newSafeI
```

If we are sure that these rules are always useful if we use the current theory, we can either add the attributes to the rule, or redeclare them as

```
lemmas [elim]: newE
lemmas [intro!]: newSafeI
```

Let's turn to simply typed set theory. It is implemented in the theory file `SimpleTypedSet.thy`. Make yourself familiar with all the operators defined in this theory and the corresponding lemmas.

Exercise 1 (6 points)
Classify (on paper) each of the six rules proven in `SimpleTypedSet.thy` as introduction or elimination rule. State whether they are safe or unsafe. ■

Finite sets The theory allows us to write finite sets using typical math notation. The definition recursively adds elements to the base set starting with the empty set. We will now derive some rules to reason about finite sets.

Exercise 2 (3 points)

1. Derive the rule `emptyE2`: `x : {} ==> False` in Isabelle.
2. Derive the safe elimination rule `emptyE`: `x : {} ==> R`.
3. Do we need an introduction rule for the empty set? How would it look like?

■

After proving some rules to deal with empty sets, we now consider insertion into a set.

Exercise 3 (5 points)

1. Proof the insertion introduction rules `insertI1`: `x = y ==> x : insert(y, A)` and `insertI2`: `x : A ==> x : insert(y, A)`.
2. For each of these rules, give an example demonstrating that these rules are not safe.
3. Proof the safe version of these rules.
   Hint: We have seen in the lecture how to use classical reasoning to derive safe rules.

■

Now that we have some rules, we can try to prove simple theorems using full automation.

Exercise 4 (3 points)
Proof the theorems using only `blast`, `fast`, or `clarify`:

1. `a : {a, b, c}`
2. `b : {a, b, c}`
3. `c : {a, b, c}`

■

Set complement Now we will switch to the complement operation. In our theory, the complement of a set `A` is written `Compl(A)`. Ultimately we want to show that `A Int Compl(A) = {}` by full automation. Hence, the first thing to do is to derive introduction and elimination rules for `Compl`.

Exercise 5 (3 points)

1. Show the rule `ComplI`: `x ~: A ==> x : Compl(A)`.
2. Show the destruction rule `ComplD`: `x : Compl(A) ==> x ~: A`
3. Show the elimination rule `ComplE`: `x : Compl(A) ==> (x ~: A ==> R) ==> R`

■

With these rules we can proof the fact we want to prove.

Exercise 6 (1 point)
Proof `A Int Compl(A) = {}` using full automation, i. e., with only one application of `blast` or `fast`. ■

Set union and set intersection Now we want to reconsider Exercise 2 from Sheet 5. There, we had a really long proof. Now we want to see how we can proof this exercise with as much automation as possible.

Exercise 7 (5 points)
Prove $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ in Isabelle using full automation. You should not use `unfolding`, but derive introduction and elimination rules for the operators used in this exercise.
■

We wish you a merry Christmas and a happy new year.

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 10 (7th January 2014)

Equational Theories and Term Rewriting In lecture "First-Order Logic with Equality" (and "Term Rewriting"), we have seen that an equational theory is given by the four rules *refl*, *sym*, *trans*, and *subst* plus additional (possibly conditional) rules of the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$. We will call the rules of an equational theory other than the four standard rules: *particular* rules. We have seen an (in general incomplete) decision procedure for an equality $e = e'$ in an equational theory, see lecture "Term Rewriting". The procedure defines a *term rewriting system* (TRS).

Our notion of *term* is that of the λ-calculus as it is built into Isabelle. It is the universal representation for object logics in Isabelle (see lecture "Encoding Syntax"). As syntactic sugar, we sometimes use constants written in infix notation, and constants applied to a tuple of arguments (as opposed to Currying).

Exercise 1 (7 points)
For each of the four general rules *refl*, *sym*, *trans*, *subst* of an equational theory, explain how (at which place exactly) the procedure reflects that rule, i. e., explain how the procedure can be modified to produce a proof using only these rules.

At what point are the *particular* rules (of the form $\phi_1 = \psi_1, \ldots, \phi_n = \psi_n \Longrightarrow \phi = \psi$) reflected?

Is rule *sym* fully catered for in the procedure? Under which conditions does it matter? Can you think of an extension of the procedure? ∎

Typically, equational theories are associated with a set of function symbols occurring in the rewrite rules. E. g., one may speak of an equational theory for $+$.

Exercise 2 (4 points)
Consider an equational theory

$$E = \{x = f(x, x)\}$$

(here f is a constant). Suppose moreover that 1 is a constant. Solve the goal

$$f(1, f(1, f(1, 1))) = 1$$

using our procedure.

Now consider the equational theory

$$E' = \{f(x, x) = x\}$$

and solve the same goal using our procedure. Any remarks? ∎

Ordered Rewriting The biggest problem for term rewriting is *(non-)termination*. For some crucial rules, this problem is solved by *ordered* term rewriting. A term ordering is a partial order on ground terms. It can be defined using a *norm*, which is some function from terms to natural numbers. In lecture "Term Rewriting" it was said that ordered term rewriting solves the problem of rewriting modulo *ACI*. Ordered rewriting will be useful in the following exercise.

Exercise 3 (8 points)
Consider the equational theories given by

$$\begin{aligned}
E_1 &= \{ \quad f(f(x,y),z) = f(x,f(y,z))\} \\
E_2 &= \{ \quad f(f(x,y),z) = f(x,f(y,z)), \quad f(x,y) = f(y,x)\} \\
E_3 &= \{ \quad f(f(x,y),z) = f(x,f(y,z)), \quad f(x,y) = f(y,x), \quad f(x,x) = x\}
\end{aligned}$$

(Here f is a constant). Suppose moreover that $0, 1, \ldots$ and l are constants.

What are the properties stated for f called? Can you give an example for E_3 from common mathematics?

Give a term ordering suitable for solving equations in E_3 while preserving termination.

For each of the above theories, use our procedure to decide the following equations positively; if equality does not hold simply state "False".

1. $f(f(l(1), l(2)), l(3)) = f(f(l(1), l(1)), l(3))$
2. $f(f(l(3), l(2)), l(1)) = f(f(l(2), l(3)), l(1))$
3. $f(l(2), f(l(1), l(1))) = f(l(2), l(1))$
4. $f(f(f(l(1), l(2)), l(3)), l(4)) = f(l(1), f(l(2), f(l(3), l(4))))$

■

Exercise 4 (2 points)
You may have noticed that the TRS E_2 of the last exercise

$$E_2 = \{f(f(x, y), z) = f(x, f(y, z)), f(x, y) = f(y, x)\}$$

with a suitable term ordering is not confluent. Take the term ordering, where the terms are first ordered by their depth (number of nested function applications) and then lexicographic, i. e., $f(a, b) < f(c, d)$ if $a < c$ or $a = c \wedge b < d$ (provided they have the same depth). Also assume that natural numbers are ordered according to their natural order. When starting with $f(f(2, 1), 3)$, then using the first rule you end with $f(2, f(1, 3))$ while using the second rule you get $f(3, f(2, 1))$ and finally end with $f(3, f(1, 2))$.

1. Give another rewrite rule that allows for swapping two elements in a right parenthesized expression. Since this rule is symmetric, the rule should only be applied if the right hand side is smaller than the left hand side according to this ordering.
2. Show, that using the rule you can rewrite the terms $f(2, f(1, 3))$ and $f(3, f(1, 2))$ to a common normal form.

■

Higher-Order Pattern Rewriting Recall *higher-order abstract syntax*. We will consider first-order logic as an example, and so you should note that the quantifiers occurring below are *constants* on the level of the representation. Assume that P, Q, and R are *metavariables* (as far as term rewriting is concerned, simply think: variables).

In the following exercise, assume that $0, 1, \ldots$ are constants, in addition to constants used for representing the logical symbols of first-order logic.

Exercise 5 (5 points)
Which of the following expressions are higher-order pattern rules (for each expression that is not, state the reason):

1. $(\forall x.P\ x \wedge Q\ x) = (\forall x.P\ x) \wedge (\forall x.R\ x)$
2. $(\forall x.(P\ x) \wedge (\lambda q.q\ x)Q) = (\forall x.P\ x) \wedge (\exists x.Q\ x)$
3. $(\lambda x.P\ x) = (\lambda y.P\ y)$
4. $(\forall x.P\ x \wedge (\exists y.Q\ y)) = (\exists y.Q\ y) \wedge (\forall x.P\ x)$
5. $(\forall x.P\ x \wedge Q\ 0) = (\forall x.P\ x) \wedge (\forall x.Q\ x)$

■

See [1, 2] for more details on term rewriting (not higher-order rewriting, however; we are not aware of a book on that topic).

References

[1] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[2] J. W. Klop. *Handbook of Logic in Computer Science*, chapter "Term Rewriting Systems". Oxford: Clarendon Press, 1993.

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 11 (14th January 2014)

Exercise 1 (1 point)
In lecture "HOL: Foundations", it has been said that a tuple (x, y) can be encoded as $\{\{x\}, \{x, y\}\}$. Show (on paper) that this encoding has all the properties you expect from the Cartesian product. ∎

Definitions in HOL The basic inference rules of HOL involve only the constants $=$, $\to$, and ϵ. All other constants are defined in terms of those three, and the rules are derived, as we saw in lecture "HOL: Deriving Rules". But now, we want to check the definitions semantically:

$$\begin{array}{lcl} \mathit{True} & = & (\lambda x^{\mathit{Bool}}.x = \lambda x.x) \\ \forall & = & \lambda\phi.(\phi = \lambda x.\mathit{True}) \\ \mathit{False} & = & \forall\phi.\phi \\ \vee & = & \lambda\phi\eta.\forall\psi.(\phi \to \psi) \to (\eta \to \psi) \to \psi \\ \wedge & = & \lambda\phi\eta.\forall\psi.(\phi \to \eta \to \psi) \to \psi \\ \neg & = & \lambda\phi.(\phi \to \mathit{False}) \\ \exists & = & (\lambda\phi.\phi(\epsilon x.\phi x)) \end{array}$$

Consider *True*: The term $\lambda x^{\mathit{Bool}}.x = \lambda x.x$ evaluates to T, and so it is a suitable definition for the constant *True*.

Now consider $\forall$: Note the use of HOAS here. $\forall$ should be a function that expects an argument ϕ of type $\alpha \to \mathit{Bool}$ (generalizing the technique we used for encoding first-order $\forall$). So ϕ is such that when you pass it an argument x of type α, it will return a proposition (something of type *Bool*). Now when does ϕx hold *for all* x? This is the case exactly when ϕx evaluates to T for all x, which is the same as saying that ϕ is the function $\lambda x.\mathit{True}$.

Think about similar intuitive explanations for the other definitions. You may also have a look at the chapter "HOL Foundations" in the lecture.

Exercise 2 (3 points)
Consider the rules and definitions of constants $\vee$, $\wedge$ etc. in Isabelle/HOL as they were presented in the lecture (this corresponds to an older version of Isabelle). Now have a look at `HOL.thy` (see the theory files in `http://isabelle.in.tum.de/library/`). Point out where the rules and definitions deviate from the presentation in the lecture. ∎

Basic HOL In this lecture we derive all well-known inference rules for logical connectives and quantifiers in HOL.

However, in a realistic setup of Isabelle/HOL, those rules are available by default since they are derived from the eight basic rules once and for all. For the sake of exercise, this week we will work with a setup where those derived rules are not present already.

We provide a theory file `HOL_BASIC` which contains the definitions of HOL without any pre-proven lemmas. You should create a script `sheet11.thy`, that imports `HOL_BASIC`.

In a previous version of Isabelle, `subst` was actually a derived rules. In the theory `HOL_BASIC`, `subst` is a basic rule. The reason why we preferred to provide `HOL_BASIC` with `subst` as a basic rule is that the derivation is quite tricky and many other exercises of this sheet depend on it. Nevertheless, in the following exercise you are asked to attempt this derivation.

Exercise 3 (3 points)
Derive the rule `subst` using only elementary proof steps (no `auto`, `simp` etc.), and using just a single basic rule from `HOL_BASIC.thy`.

Hint: this proof is short, and all the difficulty is at the level of Isabelle technicalities (generating equalities in the meta-logic and folding resp. unfolding them). ∎

The following exercise illustrates an important point using a very simple example: The unification algorithm of Isabelle is incomplete!

Exercise 4 (3 points)
Derive the following rules in `HOL_BASIC`:

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

Hint: For `fun_cong`, you should first draw a derivation tree. When you translate this tree into an Isabelle backwards proof, Isabelle will not find the unifier. You have to think of a way of helping Isabelle find the unifier. There are at least three techniques for doing this. ∎

Exercise 5 (10 points)
Derive the following rules from the lecture in `HOL_BASIC`.

1. $s = t \Longrightarrow t = s$ (*sym*)
2. $[\![r = s; s = t]\!] \Longrightarrow r = t$ (*trans*)
3. $[\![P \Longrightarrow Q; Q \Longrightarrow P]\!] \Longrightarrow P = Q$ (*iffI*)
4. $[\![P = Q; Q]\!] \Longrightarrow P$ (*iffD2*)
5. $True$ (*TrueI*)
6. $P = True \Longrightarrow P$ (*eqTrueE*)
7. $P \Longrightarrow P = True$ (*eqTrueI*)
8. $(\bigwedge x.P\,x) \Longrightarrow \forall x.P\,x$ (*allI*)
9. $(\forall x.P\,x) \Longrightarrow P\,x$ (*spec*)
10. $False \Longrightarrow P$ (*FalseE*)
11. $False = True \Longrightarrow P$ (*False_neq_True*)
12. $True = False \Longrightarrow P$ (*True_neq_False*)
13. $(P \Longrightarrow False) \Longrightarrow \neg P$ (*notI*)
14. $[\![\neg P; P]\!] \Longrightarrow R$ (*notE*)
15. $\neg(True = False)$ (*True_Not_False*)
16. $P(x) \Longrightarrow \exists x.P\,x$ (*existsI*)
17. $[\![(\exists x.P\,x); \bigwedge x.P\,x \Longrightarrow Q]\!] \Longrightarrow Q$ (*existsE*)
18. $[\![P; Q]\!] \Longrightarrow P \wedge Q$ (*conjI*)
19. $P \wedge Q \Longrightarrow P$ (*conjEL*)
20. $P \wedge Q \Longrightarrow Q$ (*conjER*)
21. $[\![P \wedge Q; [\![P; Q]\!] \Longrightarrow R]\!] \Longrightarrow R$ (*conjE*)
22. $P \Longrightarrow P \vee Q$ (*disjIL*)
23. $Q \Longrightarrow P \vee Q$ (*disjIR*)
24. $[\![P \vee Q; P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R$ (*disjE*)
25. $P \vee \neg P$ (*excl_midd*)

■

In Exercise 2 we have seen that the existential quantifier is not defined using Hilbert's choice operator. In the theory `HOL_BASIC` we have the definition `EXold_def` that defines the existential quantifier `Exold` using the choice operator. The next exercise tries to show equivalence of these two definitions.

Exercise 6 (2 points)
Prove the following rules in the theory `HOL_BASIC`:

1. $\exists x.Px \Longrightarrow \exists_{\text{old}} x.Px$
2. $\exists_{\text{old}} x.Px \Longrightarrow \exists x.Px$

■

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 12 (21st January 2014)

The Semantics of HOL In HOL, we have the three basic constants ϵ, $=$ and $\rightarrow$. We then define the constant *True*, and afterwards the constant *False*. To understand on a semantic level why $\forall P.P$ is an appropriate definition for *False*, it is instructive to see explicitly that it is not the case that "everything is true".

Exercise 1 (3 points)
Expand the definition of *False* to a lambda-term ϕ using just the constants ϵ, $=$, $\rightarrow$. Show that $\mathcal{V}_A^{\mathfrak{M}}(\phi) = F$ for all interpretations $\mathfrak{M}$ and assignments A. ∎

Working with Isabelle HOL From now on, we will work with a full-fledged Isabelle HOL setup. You will have to set the logic used by Isabelle to "HOL" in the same way that you set it to "FOL" at the beginning of this course. Remember to restart `jedit`. Instead of `FOL` your theories should now import `HOL`.

Simplifier Much more than the theories we have looked at previously, `HOL` is centered around proving equalities. The reason for the central role of equalities is that in higher-order logic, formulae are terms of type *bool*, and so rather than saying that two formulae are equivalent, we say that they are equal.

Since equality has such a central role, the *simplifier* is extremely important in `HOL`. Generally in Isabelle, the simplifier is a module that simplifies subgoals by *rewriting* using equalities. Of course, those equalities must be true in the given theory. Even so, one should not simply apply equalities blindly, since this may lead to inefficiencies or even non-termination. For example, if our theory contains an equality $x + y = y + x$, then this equation can be applied infinitely many times.

Note that technically, the rewriting works with *metalevel* equalities ($==$), not object level equalities ($=$). The following simple exercise sheds light on this distinction.

Exercise 2 (2 points)
Derive the lemmas $(x = y) \Longrightarrow (x == y)$ and $(x == y) \Longrightarrow (x = y)$ in `HOL` and add a little explanation as comment in your proof script. Use elementary proof steps (no `auto`, `simp`, `blast` etc.) and only basic rules (i. e., axioms in `HOL.thy`, not lemmas). ∎

The simplifier has to be set separately for each major Isabelle theory such as FOL, ZF, HOL. The data-structure containing the setup of the simplifier is called the *simpset*. For `HOL`, this is built by reading the file `simpdata.ML`. When we work with extensions of `HOL`, we will mainly rely on this setup. However, we will also sometimes modify the simpset.

To manipulate the simpset, you can use the attributes `simp`, `split`, and `cong`. Adding the attribute `simp` to a lemma makes this lemma available as simplification rule. The attribute `split` declares splitting rules (recall the lecture on "Term Rewriting"). Finally, to simplify arguments of a function, Isabelle uses congruence rules that are declared with the `cong` attribute. For example, the lemma

$$P = Q \Longrightarrow (Q \Longrightarrow a = c) \Longrightarrow (\neg Q \Longrightarrow b = d) \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ a\ \mathit{else}\ b) = (\mathit{if}\ Q\ \mathit{then}\ c\ \mathit{else}\ d)$$

can be used to simplify all arguments of the `if` function. A slightly simpler variant of this lemma can be used to only simplify the condition, but neither the `then` nor the `else` part:

$$P = Q \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ a\ \mathit{else}\ b) = (\mathit{if}\ Q\ \mathit{then}\ a\ \mathit{else}\ b).$$

One additional point to note here is the use of the object level equality (=). When Isabelle adds an equality the the simpset, this equality is implicitly turned into a meta-level equality using the rules from the previous exercise. Hence, all object level equalities can be added to the simpset without explicitly turning them into meta-level equalities. If we for example want to find which rules Isabelle already has proven containing the `if` construct, we can use `find_theorems`:

```
find_theorems "if ?P then ?Q else ?R"
```

In the output we see that `If_def` is actually a meta-level equality, but all other equalities are object-level equalities. Nevertheless we can rewrite an `if` into a disjunction using `simp add:if_bool_eq_disj`.

You can inspect the current simpset with the command `print_simpset`.

Tracing If you want to see what the simplifier is doing you can set the attribute `simp_trace` to `true`. You can do this either by declaring this attribute globally or noting it locally, i. e., using `declare[[simp_trace=true]]` at global theory level, or `note[[simp_trace=true]]` in a proof script. Additionally you can set the attribute `simp_trace_depth_limit` to a positive number. This attributes controls how many recursive invocations of the simplifier should be traced. The default value is 1.

For more details on the simplifier, you should look at [1, Chapter 10]. In the following exercise, you will find that using a combination of simplification and `blast` (or `fast` if you like) is effective.

Integration of Simplifier and Automation Isabelle offers some automatic provers that integrate the simplifier. These tools are `auto`, `force`, `fastforce`, `slowsimp`, `bestsimp`, and `clarsimp`. The automated tools `auto` and `clarsimp` - much like `clarify` - have a special role. These tools try to solve a goal, but do not fail if the goal cannot be proven. Instead, they give you a simplified goal.

Besides the usual methods to manipulate the claset used by these tools, there is also a way to manipulate the simpset. You can add or delete simplification, congruence, and splitting rules using `simp add`, `cong add`, `split add`, resp. `simp del`, `cong del`, or `split del`. Furthermore, you can restrict the simplification rules to the rules in a specific set using `simp only`.

Nitpicking a formula Isabelle/HOL has limited support for counterexample generation. The tool `nitpick` for example can be used to search for a counterexample. If you try to find a counterexample to the formula $A \vee B$ using `nitpick`, you should create a proof script like this:

```
lemma "A\/B" nitpick oops
```

Note that you have to end the prove-mode of Isabelle using the `oops` command, or finish the proof in case no counterexample exists. In this example, `nitpick` finds that setting the variables A and B to *False* will falsify this formula.

Exercise 3 (6 points)
The following are some theorems and non-theorems of `HOL`. For each of them, prove it or else state that it is not provable.

1. $(\mathit{if}\ \mathit{False}\ \mathit{then}\ A\ \mathit{else}\ B) = B$
2. $(\mathit{if}\ \mathit{False}\ \mathit{then}\ A\ \mathit{else}\ B) \neq A$
3. $[\![\neg E \Longrightarrow B = B']\!] \Longrightarrow (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B')$
4. $[\![\neg E \Longrightarrow B \neq B']\!] \Longrightarrow (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) \neq (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B')$
5. $[\![A \neq A'; B \neq B']\!] \Longrightarrow (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) \neq (\mathit{if}\ E\ \mathit{then}\ A'\ \mathit{else}\ B')$
6. $(\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ \neg E\ \mathit{then}\ B\ \mathit{else}\ A)$
7. $f\ (\mathit{if}\ E\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ E\ \mathit{then}\ (f\ x)\ \mathit{else}\ (f\ y))$
8. $P \vee Q \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ Q\ \mathit{then}\ B\ \mathit{else}\ A)$

9. $[\![P \vee Q; P = (\neg Q)]\!] \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ Q\ \mathit{then}\ B\ \mathit{else}\ A)$
10. $[\![\neg E \Longrightarrow B \neq B'; \neg E]\!] \Longrightarrow (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) \neq (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B')$
11. $(\mathit{if}\ R\ \mathit{then}\ A\ \mathit{else}\ B) \Longrightarrow A \vee B$
12. $A \vee B \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ A\ \mathit{else}\ B)$

It may be interesting to switch on the tracer as explained above and look at how the simplification has worked. ∎

Closedness Requirement in Constant Definitions Recall from lecture "Conservative Theory Extensions" that the term defining a constant must be closed. We illustrated what goes wrong otherwise. We now repeat this in Isabelle.

Exercise 4 (2 points)
Formalise a flawed constant definition for the `fix` rule from the lecture in `Nonconservative.thy`. You only need a line of the form

```
axiomatization where ...
```

Have a look at `HOL.thy` for some examples.
Hint: You should think of a name for this axiom. ∎

Exercise 5 (3 points)
Prove `False` in the theory `Nonconservative.thy`. ∎

References

[1] L. C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, October 2007.

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 13 (28th January 2014)

The `primrec` syntax In lectures "Well-Founded Recursion" and "Arithmetic", we have seen the `primrec` syntax for defining recursive functions.

Exercise 1 (3 points)
Consider the functional `iterate` defined as follows:

$$\texttt{iterate}\ n\ f\ a = f^n(a)$$

Generate a theory file `Iterate.thy`, where you define this function `iterate` (its type is $nat \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha$) using the `primrec` syntax.

Try it out on the goals

1. `iterate` $(Suc(Suc\,0))\ Suc\ (Suc\ 0) = ?x$;
2. `iterate` $(Suc(Suc(Suc\,0)))\ (Suc \circ Suc)\ 0 = ?x$.

What instantiations do you obtain for $?x$ in each case?
Hint: Look around in any theory files to get the syntax right. ∎

Primitive recursion Typing `thm nat.recs` gives you a scheme for defining primitive recursive functions on the natural numbers. Recall (from your knowledge of the theory of computation [2]) that a primitive recursive function f of type $nat \Rightarrow nat$ is defined by giving a constant c (of type nat) and a function g of type $nat \Rightarrow nat \Rightarrow nat$, as follows:

$$\begin{aligned} f\ 0 &= c \\ f\ n+1 &= g\ n\ (f\ n) \end{aligned}$$

In Isabelle, the expression $nat\_rec\ c\ g$ defines f. You can address the two `thm`'s of the recursion scheme by the names *nat_rec_0* and *nat_rec_Suc* (search in `Nat.thy` to understand why).

Exercise 2 (2 points)
Define the factorial function by primitive recursion, i.e., find appropriate ϕ and γ, and prove the following two goals:

$$\begin{aligned} &[\![c = \phi;\ g = \gamma]\!] \Longrightarrow && nat\_rec\ c\ g\ 0 = Suc\ 0 \\ &[\![c = \phi;\ g = \gamma]\!] \Longrightarrow && nat\_rec\ c\ g\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))) = \\ &&& Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (\\ &&& Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (\\ &&& Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (\\ &&& Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))))))))))))))))))))) \end{aligned}$$

Hint: if you find it silly to type $Suc\ (Suc\ 0)))))))))))))))))))))))$, just type 24.

Now prove the goal

$$[\![c = \phi;\ g = \gamma]\!] \Longrightarrow nat\_rec\ c\ g\ 13 = ?x$$

What instantiation do you obtain for $?x$?
Advice: have a coffee while Isabelle calculates. ∎

AVL trees In the remaining weeks, we will develop a theory of *AVL trees*. These are binary trees where the inner nodes are labeled with terms of a type α. These trees can thus be used to store such a set of labels and thereby implement finite sets, dictionaries etc. AVL trees allow for efficient insert-algorithms of the order $O(\log n)$. The idea is to maintain a certain "balanced-ness"-invariant during inserting by certain "rotation-operations". We will develop the necessary recursive functions in such a way that they can be executed in Haskell (and with some minor changes related to type classes) also in SML.

Before diving into AVL trees we start with binary search trees. Hence, in this week, we will not consider any balancing operations. We will start with the definitions of a datatype and a few primitive recursive functions in a newly created theory `Tree.thy`. As a convention we insert elements that are less than the element in the root of the tree into its left child and elements that are greater than the element in the root of the tree into its right child. Note that (in general) it is easier to only use one operator for comparisons, e. g., $<$.

```
datatype 'a tree = Leaf | Node 'a "('a tree)" "('a tree)"
```

is interpreted by Isabelle as a datatype definition, as we have seen in lecture "Datatypes". Similar to a type definition using `typedef`, such a definition will cause Isabelle to prove a number of theorems about the new type, e. g., stating that $Leaf \neq \texttt{Node}\, a\, t_1\, t_2$ or that there is an induction schema for this type.

So each node has a node label of type `'a`, and a left and right subtree. But this definition only defines a binary tree. To model a binary search tree we need to add an additional constraint on the type.

Exercise 3 (1 point)
Create the theory file `Tree.thy` and import the theory `Main.thy` which is a collection of some Isabelle/HOL theories. Define a datatype for binary trees. ■

Next we define a containment test function `isin` and a function `toset` to transform a tree into a set.

Exercise 4 (2 points)

1. Define the primitive recursive function `isin` that tests whether a specific element is contained in the tree. You should not assume any ordering of the tree nodes.
2. Define the primitive recursive function `toset` that returns a set containing all elements in the tree.

■

Proofs by Induction and Case Splitting To prove properties related to datatypes one often uses structural induction. In fact, when a new datatype is defined in Isabelle, she proves an induction theorem. Assuming your datatype is called `tree` like in the example above, typing `thm tree.induct` shows you the induction theorem. This theorem can then be used in conjunction with the proof method `induct`.

There are several ways to tell Isabelle what kind of induction you want to use. We will simply tell Isabelle the variable over which we want to induct and let Isabelle find the correct induction rule. Hence, if our variable is `t`, we use the proof method `induct t`. In our case, this generates two cases. The base case asks us to show the property provided the tree is a leaf. The induction step then asks us to show the property for a non-leaf node provided we know that the property holds for the left and the right child. If you want to generalize parameters for the induction you can use `arbitrary:` to specify them. In general, if you have a parameter in your assumptions or goals that depend upon the variable you use for induction, you have to make this parameter general as it will have different values in the different cases. We will see examples of general parameters later.

A similar proof method is `cases`. This method simply splits the goal according to the different type constructors. In the case of our tree, `cases t` would ask us to show the property provided the tree is a leaf, and then to show the property provided the tree is a non-leaf node. However, we don't get any hypothesis about the children of the non-leaf node. Hence, in general,

if you can prove a fact using `cases` you can also prove it using `induct`, but not vice versa. Case splitting over a Boolean predicate can also be realized with `cases`. Additionally, if you start a proof with `cases` you can use the `case` command in your proof script to select the individual cases. Consider the following proof.

```
lemma "(~A ==> B) ==> A \/ B"
proof (cases A)
  case True then show "A \/ B" ..
  next
  case False
    assume ab: "~A ==> B"
    from False have "B" by (rule ab)
    thus "A \/ B" ..
qed
```

After the proof statement `case True` the current fact is set to A since we split cases on A. Essentially this statement corresponds to `assume True: "A"`. Hence, you can recall that fact whenever you need the assumption A.

When dealing with datatypes the situation is a little bit different. Assuming t is a tree and we write `cases t` we get two cases: One for the empty tree and one for an internal node. The empty tree case is simply named `Leaf` and corresponds to `assume Leaf: "t=Leaf"`, but the other case is a bit more tricky. The case is called `Node`. But since the `Node` constructor takes arguments, the case is specified as `case (Node a l r)`. This corresponds to

```
fix a l r
assume Node: "t = (Node a l r)"
```

Cases can also be used with `induct`. In this case, the assumption for the current case also contains the whole induction premise.

Exercise 5 (2 points)
Proof that every x that is contained in the tree is also in the set generated by `toset`, i. e., proof `(isin x t) = (x : toset t)`

Hint: You need induction. ∎

Search Tree Functions To simplify typing we put the remaining lemmas and definitions in the context `linorder`. This is done by placing them between `context linorder begin` and the corresponding `end`.

The function `isin` from Exercise 4 is rather naïve. Especially when operating on binary search trees we can provide a much more efficient version. But to ensure correctness of this function, we need to restrict the input to sorted trees.

Most of the proofs can be done automatically. Sometimes Isabelle finds an `if` in the assumptions. Then, the simplifier usually gives up since it is not configured to split `if`s in the assumptions. You can fix this by adding `split: split_if_asm` as a modification to the simplifier setup.

Exercise 6 (2 points)
Consider the following function:

```
primrec isordP :: "'a tree => ('a => bool) => bool" where
  isord_leaf: "isordP Leaf P = True" |
  isord_step: "isordP (Node n l r) P =
    (P n & isordP l (%x. P x & x < n) & isordP r (%x. P x & n < x))"
```

This function is a helper function to define a function `isord` that returns *True* if and only if the input tree is sorted.

1. Explain the second parameter of `isordP`. What is it used for?
2. Using `isordP` define a function `isord :: "'a tree => bool"` that return *True* if and only if the input tree is sorted.

■

This function `isin` is correct even if the tree is not sorted. If the tree is sorted, we can write a more efficient version, called `contains`.

Exercise 7 (1 point)

1. Write the function `contains` that takes a tree and returns *True* if and only if the element occurs in the tree. This function should assume the input tree to be sorted. Note that you cannot state this directly when defining the function.

■

Now an obvious goal is to show that `contains` and `isin` return the same value under suitable assumptions. But before we can do this, we need a couple of helper lemmas.

Exercise 8 (5 points)
Show the following helper lemmas:

1. `isordP t P ==> isin x t ==> P x`

   Note that P in the induction hypothesis differs from the P in the conclusion.

2. `contains x t ==> isin x t`

3. `P x ==> isordP t P ==> isin x t ==> contains x t`

■

Exercise 9 (3 points)
Using these helper lemmas show `(isin x t) = (contains x t)` under a suitable hypothesis on `t`. ■

So far we did not modify a tree. Now, we will define insertion into a binary search tree. By convention, we place elements less than the label of the current node into the left subtree, elements greater than the label in the right subtree, and elements equal to the current element are already contained and not added a second time.

Exercise 10 (3 points)
Define a function `insert::"'a => 'a tree => 'a tree"` that takes an element and a tree and inserts this element into the tree according to our convention. You might want to use the `if` construct of Isabelle. ■

Finally, we can prove some simple lemmas about insertion.

Exercise 11 (6 points)
Prove the following lemmas.

1. `isin x (insert x t)`

2. `isin y t ==> isin y (insert x t)`

3. `P x --> isordP t P --> isordP (insert x t) P`

4. `isord t --> isord (insert x t)`

5. `contains x (insert x t)`

6. `contains y t --> contains y (insert x t)`

■

Universität Freiburg
Institut für Informatik
Lehrstuhl für Softwaretechnik
Dr. Jochen Hoenicke Jürgen Christ

Computer Supported Modeling and Reasoning WS13/14
Exercise Sheet No. 14 (4th February 2014)

AVL trees This week we continue to look at AVL trees. In last week's exercises you should have obtained a syntactically correct Isabelle theory, and a number of `thm`'s of this theory have already been proven by Isabelle. You should import the theory `Tree.thy` in your new theory `AVL.thy`. Again we will work in the context `linorder`.

Names for recursion rules It can be convenient to have names for the two equations (base case and recursive case) of a recursive definition. The `primrec` syntax allows for this. Instead of

```
primrec height :: "'a tree => nat" where
"height Leaf = 0" |
"height (Node n l r) = Suc(max (height l) (height r))"
```

you can have

```
primrec height :: "'a tree => nat" where
height_empty: "height Leaf = 0" |
height_branch: "height (Node n l r) = Suc(max (height l) (height r))"
```

defining two names `height_empty` and `height_branch` for the according `thm`'s. You should copy this fragment into `AVL.thy`.

Balancing AVL Trees The efficiency of the datastructure 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 deeper on the right, or deeper on the left. To this end, you should insert the following lines into the `AVL.thy` file:

```
datatype bal = Just | Left | Right

consts bal :: "'a tree => bal"

defs bal_def:
  "bal t == case t of
             Leaf => Just |
             (Node n l r) => if height l = height r then Just
                             else if height l < height r then Right
                                  else Left"
```

Note that datatype definitions and the declaration and definition of constants have to be done outside of any contexts. Hence, you either have to put the definition before you begin the context `linorder`, or temporarily end that context. If you want to define something inside a context, use the `definition` syntax.

Exercise 1 (1 point)
What is the complexity of *bal* in n, where n is the number of nodes in the tree? If inserting

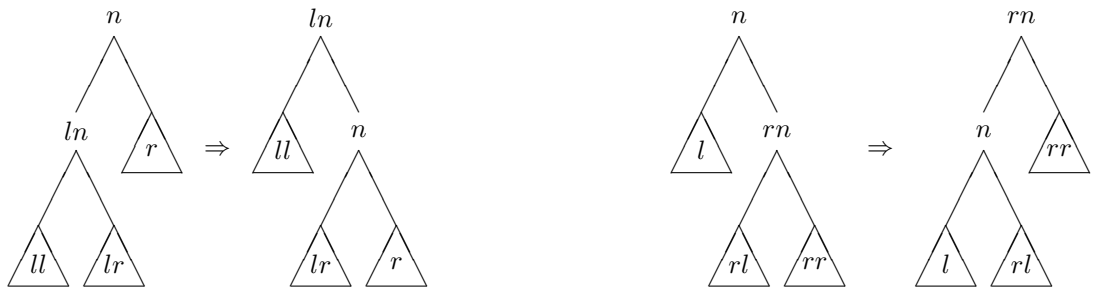

Figure 1: *r_rot* and *l_rot*

a node into a tree involves calls to *bal*, do you think this complexity is satisfactory? Can you suggest a way of improvement? ■

We now consider the rotation operations. There are four different kinds of rotations, depicted in Figures 1 and 2.

You should insert the following lines into the `AVL.thy` file:

```
fun r_rot :: "'a * 'a tree * 'a tree => 'a tree" where
"r_rot (n, Node ln ll lr, r) = Node ln ll (Node n lr r)"

fun rl_rot :: "'a * 'a tree * 'a tree => 'a tree" where
"rl_rot(n, l, Node rn (Node rln rll rlr) rr) =
   Node rln (Node n l rll) (Node rn rlr rr)"
```

The `fun` syntax is provided in Isabelle for defining recursive functions. It is more general than the `primrec` syntax, since a the function definition does not have to be primitive-recursive. Isabelle tries to find a well-founded ordering to ensure termination of the recursive definition. If she fails, the definition is rejected. But, as you might know, termination is undecidable. Hence, Isabelle might reject a definition even though the defined function would actually terminate. In these cases, the `function` syntax can be used. This syntax is slightly more powerful than the `fun` syntax since a well-founded ordering can be provided by the user. We use the `fun` syntax although the rotation functions are not recursive, since this syntax also allows for *pattern matching*, which is convenient. After inserting these lines, Isabelle will show you the ordering found to prove termination which is `{}` in this case. Essentially this means that every call to this function terminates since it is non-recursive. You might want to try different (especially recursive) definitions.

Note that each rotation function does not actually take a tree as input, but rather a triple consisting of a node and two trees. However, this is a a technical detail: conceptually, the rotation functions have a tree as input and output. Note moreover that the rotation functions are partial: e.g., $r\_rot\ (n, Leaf, Leaf)$ is undefined. This is no problem.

Exercise 2 (2 points)
Define the two missing rotation functions in your `AVL.thy` file. ■

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 (why?). If $x < n$, we must insert x into l (to maintain the orderedness property). Let l' be the tree obtained by inserting x into l, and assume (by "inductive hypothesis") that it is balanced and ordered. As an 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

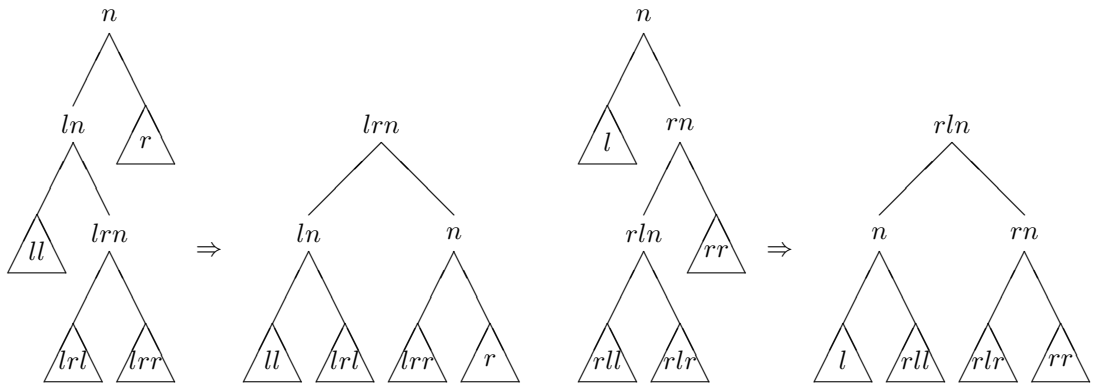

Figure 2: *lr_rot* and *rl_rot*

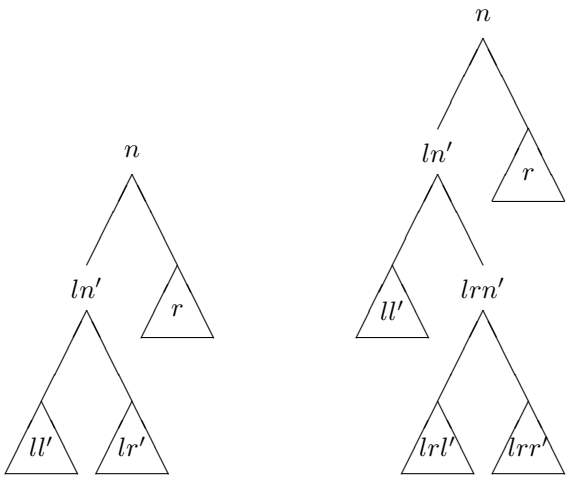

Figure 3: Too deep on left

of Figure 3, where either $height\ ll' = height\ r + 1$ or $height\ lr' = height\ r + 1$.[1] The tree is too deep 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 3, 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 2).

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 1).

Exercise 3 (2 points)
Define a function *l_bal* which performs an appropriate rotation for a tree that is too deep on the left. Here is a fragment of the code you should insert into `AVL.thy`:

```
definition
  l_bal :: "                                  " where
  l_bal_def: "l_bal n l r == if
                   then
                   else                  "
```

In complete symmetry, define a function *r_bal*. ∎

The insertion function is defined as follows:

```
primrec insert_avl :: "'a::order => 'a tree => 'a tree" where
"insert_avl x Leaf = Node x Leaf Leaf" |
"insert_avl x (Node n l r) =
   (if x=n
    then Node n l r
    else if x<n
         then let l' = insert_avl x l
              in if height l' = Suc(Suc(height r))
                 then l_bal n l' r
                 else Node n l' r
         else let r' = insert_avl x r
              in if height r' = Suc(Suc(height l))
                 then r_bal n l r'
                 else Node n l r')"
```

You should also insert this fragment into `AVL.thy`.

"Declarative" AVL trees So far, we have reconstructed the theory of AVL trees as defined by Cornelia Pusch and Tobias Nipkow. We have already mentioned that there are some inherent inefficiencies in the way that the datastructure itself is defined and the procedures are implemented. This is what computer-supported modeling and reasoning is about: one should think of the theory file so far as a *specification* of AVL trees rather than an implementation. We use this specification to prove any property about AVL trees that we consider essential for "correct" behaviour of AVL trees. We will then aim at implementing AVL trees in more efficient ways.

But first, we work on the theory file we have so far. In the file `AVL_lemmas.thy`, you find a number of essential lemmas about AVL trees. Several symmetric cases are left out (i. e., they are proven by `sorry`). A point to note is that the goals are usually of the form $\phi_1 \longrightarrow \ldots \longrightarrow \phi_n \longrightarrow \psi$, where $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \psi$ would be more natural. However, it turns out that highly automated proofs can deal better with the first form. The attribute `rule_format` saves the theorem as $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \psi$ rather than $\phi_1 \longrightarrow \ldots \longrightarrow \phi_n \longrightarrow \psi$ when the command `qed` is issued.

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

You may encounter problems with the provided theory file since you might have defined the `AVL` theory file in a different way than expected by `AVL_lemmas.thy`. If this is the case, see if you can repair `AVL.thy` appropriately. Comment out all lemmas you cannot repair.

Exercise 4 (2 points)
Proof the symmetric cases in `AVL_lemmas.thy`. ■

Exercise 5 (2 points)
For each of the lemmas already proven in `AVL_lemmas.thy`, state in simple informal language what it says (you may say "symmetric to ...", wherever appropriate).

Which of the proofs needed induction? Can you give a high-level explanation for this? Your explanation should make plausible that, e.g., `isbal_r_rot` does not need induction. ■