Computer Supported Modeling and Reasoning

PD Dr. Jan-Georg Smaus

Spring 2010

Universität Freiburg
Institut für Informatik
Lehrstuhl für Grundlagen der Künstlichen Intelligenz
PD Dr. Jan-Georg Smaus

Computer Supported Modeling and Reasoning II Semestre A.A.2009/2010 Exercise Sheet No. 1 (15th March 2010)

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 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 *ProofGeneral* system is built on top of emacs and provides many functionalities for running Isabelle.

In these exercises, we will opt for (3).

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 ProofGeneral.

Configuring your system for Isabelle You will always run Isabelle by starting xemacs. If you are not familar with xemacs you should look at

```
http://www.digilife.be/quickreferences/QRC/XEmacs%20Reference%20Card.pdf
```

Create a working directory, move to it, and type `xemacs`. You can customize xemacs using the rich menus provided. Open (create) the file `ex1.ML`, which will be your first *proof script*, that is, a file containing Isabelle proofs.

Now customize xemacs:

- Multiple Windows: Isabelle will use several xemacs subwindows (simply *windows* in emacs jargon). You may choose if you want these displayed all in one window (called *frame* in emacs jargon) split into several parts or as several windows. This is done by (un)ticking the box "Proof-General → Options → Display → Multiple Windows".
- X-Symbol: For a nice rendering of mathematical symbols, xemacs uses the X-Symbol package. To enable it, tick the box "Proof-General → Options → X-Symbol".
- Electric Terminator: This will save you from having to type the return key to fire proof commands. Proceed in analogy to X-Symbol.
- Fly Past Comments: This will ignore comments when processing a proof step by step. Proceed in analogy to X-Symbol.

  You should now select "Proof-General → Options → Save Options"

- Choosing the logic: We will use the logic FOL. Go to "Proof-General → Advanced → Customize → Isabelle → Chosen Logic". This will open a new buffer. Set the logic to "FOL" (following the instructions given in that buffer) and click "Save".

You are encouraged to experiment with the configuration menu to configure ProofGeneral according to your personal preferences! You might also want to consult the documentation at `http://proofgeneral.inf.ed.ac.uk/userman`.

Before actually starting Isabelle, we will explain some more points.

Files and Directories On the computers you are using, only the Isabelle binaries are installed, so that you cannot see the sources. We put some files from the sources (for Isabelle2005, which is the version we are using) on the course's webpage. For the newer version Isabelle2008, the sources can be found on the web (most of it actually stayed the same): `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`. You will find several standard Isabelle theories on `http://isabelle.in.tum.de/library/`.

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 consider the syntax of formulae in Isabelle. The logical connectives are typed as `-->`, `\/` (or `|`), `/\` (or `&`), and `~`. Note that xemacs displays those as the connectives used in paper and pencil proofs, thanks to the X-Symbol package. Just type some formulae in `ex??.ML` to see this (but then erase them again).

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. $\perp$-*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., `conjI;` in the file `ex1.ML`. The so-called *Response buffer* will pop up and display "[|?P; ?Q|] $\Longrightarrow$?P$\wedge$?Q" : `thm`, which gives the value and the type of the expression `conjI`. The type `thm` is one of the most important datatypes of Isabelle.

Note that typing `conjI;` has started Isabelle. During the Isabelle session, the Response buffer will display messages from the Isabelle system, but depending on how you configured your system, the buffer will not be visible.

Goals The most common way of using Isabelle is by means of *goals* and *tactics*, and a first intuitive understanding of this is that you build a proof (tree), as you would in a paper and pencil proof, but from the bottom to the top. At any point in an Isabelle session, the Isabelle system will be in a 'state', which consists of one or more *goals*. A goal is a statement you are currently trying to prove. A *tactic* is a function you apply to manipulate the state.

We explain these ideas in detail using the formula $A \rightarrow (B \rightarrow A)$. Type

```
goal thy "A --> (B --> A)";
```

in `ex1.ML`. The function `goal` takes two arguments: our current theory `FOL`, to which the identifier `thy` is bound, and a formula, encoded as a string. The effect is to put the Isabelle system in a state which says: without making any assumptions, prove $A \to (B \to A)$ in the theory `FOL`. The so-called *Goals buffer* will pop up displaying this state.

In our paper and pencil proof of $A \to (B \to A)$, the only rule we used was $\to$-*I*, and so in the Isabelle proof, we will use `impI`. If we replace (i.e., unify) `P` with A and `Q` with $B \to A$, the rule `impI` says: in order to prove $A \to (B \to A)$ without making any assumptions, prove $B \to A$ under the assumption A. Typing

```
by (rtac impI 1);
```

has the effect of manipulating the state in this way. Look in the Goals buffer and you will see it. What happens technically is that the current goal $A \Longrightarrow B \to A$ is obtained by *resolving* the rule `impI` and the previous goal $A \to (B \to A)$; `rtac` stands for 'resolution tactic'. Intuitively, our current state says: if we can prove $B \to A$ under the assumption A, we are done.

You may find it difficult to understand the difference between $\Longrightarrow$ and $\longrightarrow$, since both somehow seem to stand for implication. However, $\longrightarrow$ is a symbol of propositional logic, which is our *object logic*, i.e., the language we are talking about. In contrast, $\Longrightarrow$ is a symbol of the *meta-logic*, i.e., the language in which we talk. A little analogy: if I say 'in German, all nouns are capitalized', I am making a statement *in* the meta language English *about* the object language German. The difference between object and meta-logic will be explained over and over again.

Now type

```
by (rtac impI 1);
```

again. This time, we should read `impI` as follows: in order to prove $B \to A$ without making any assumptions, prove A under the assumption B. Look in the Goals buffer. Our current state says: if we can prove A under the assumptions A and B, we are done. Trivially one can prove A under the assumption A. In Isabelle, this is made explicit by the so-called *assumption* tactic. Type

```
by (atac 1);
```

The effect of this tactic is to remove the first (and in this case only) subgoal provided the conclusion to be proven (in this case A) is one of the assumptions. This completes our proof of $A \to (B \to A)$. Try to see that we built the proof tree starting from the bottom. We can save the theorem by typing

```
qed "aba";
```

This makes `aba` a theorem (an expression of type `thm`) which can be used from now on in the same way as any rule in `IFOL.thy`, say `impI`.

Shortcuts `by (rtac` *rule n*`)` can be abbreviated as `br` *rule n*. `by (atac` *n*`)` can be abbreviated as `ba` *n*.

Metavariables Sometimes Isabelle introduces variables preceded by '?'. These are called *metavariables*, and we now explain what they are. To understand these explanations, you must try it out in Isabelle.

Let us prove the formula $A \wedge B \to A$. In Isabelle you would start as follows

```
goal thy "A /\ B --> A";
br impI 1;
```

The current goal will be

$$A \wedge B \Longrightarrow A$$

This reads as: If we are able to derive `A` assuming `A` $\wedge$ `B`, we are done. In turn, the rule `conjunct1` reads: If we can prove ?`P`$\wedge$?`Q`, we can also prove ?`P`. Now, ?`P` and ?`Q` could be any formula. In particular, ?`P` could be `A`. This is what happens when you type

```
br conjunct1 1;
```

You are currently trying to prove `A`, and if you want use `conjunct1`, then clearly `?P` must be *instantiated* to `A`. We say that the resolution tactic *unifies* the conclusion of our current goal (`A`) with the conclusion of the rule `conjunct1` (`?P`), which yields a *substitution* that replaces `?P` with `A`. As new goal, we have to prove the premise of `conjunct1`, using the old assumptions. But the unification did not involve the `?Q` that also occurs in `conjunct1`, and therefore Isabelle (using this tactic) cannot know how `?Q` should be instantiated. Therefore she leaves this instantiation open and introduces a metavariable (when I tried it was `?Q1`). You may also say that the instantiation of `?Q` is *delayed*: we do not know yet what `?Q` will be. As next step you may type

```
ba 1;
```

You can now better understand proving by assumption: the conclusion does not need to be identical to one of the premises, but rather some *instance* of the conclusion must be identical to one of the premises. Here, `?Q1` is replaced with `B` and the subgoal is solved. We are done.

Exercise 1
Prove the following theorems using paper and pencil and using Isabelle.

For all theorems you prove (also in later exercises), use `qed` to save the theorem. As name, use the number of the exercise, e.g. `ex1_1` for $A \wedge B \to B \wedge A$.

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. $(A \vee B) \wedge (B \vee C) \to B \vee (A \wedge C)$

■

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, then chances are that the chosen tactic applied to that goal fails altogether. But the point is: each subgoal must be solved eventually anyway.

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.

Derived Rules We will now see that it possible to *derive* rules in Isabelle.

To do so, we have to understand some Isabelle and ML technicalities. Suppose we want to derive the rule

$$\frac{A \wedge B}{B \wedge A} \wedge - comm$$

It is possible to type

```
goal thy "P&Q ==> Q&P";
```

but the problem is that Isabelle does then not display the premise `P&Q`, but only the desired conclusion `Q&P`. In fact, we have no way of referring to the premise. The solution is to type

```
val [prem] = goal thy "P&Q ==> Q&P";
```

instead. Try to understand what happens here in terms of ML. `goal` is a function which takes as arguments a theory and a string (this is the reason you always have to put "" around a formula), and it returns a list of `thm`'s. Just type `goal;` to see the type of `goal`. The list of `thm`'s it returns are the premises of the goal you are trying to prove. In our example this is the list containing only one element, namely the premise `P&Q`. `val [prem] =`... has the effect of binding `[prem]` to the list of premises, i.e., binding `prem` to the one premise we have. Type

```
val [prem1,prem2] = goal thy "P&Q ==> Q&P";
```

Why do you think things go wrong this time?

You can use the premise as a rule whose name is `prem`. For example, the proof of the derived rule above would be

```
val [prem] = goal thy "P&Q ==> Q&P";
br conjI 1;
br conjunct2 1;
br prem 1;
br conjunct1 1;
br prem 1;
qed "conjComm";
```

Instead of `goal`, you can also use `Goal`. This has two advantages: the argument `thy` is not required any longer (since the current theory is assumed), and it is usually not necessary to bind the assumptions of the goal you want to prove to identifiers. So instead of `val` *prems* = `goal thy`... you simply type `Goal`.... The assumptions will be visible in your subgoals (see [1, Section 2.1.1]). However, this does not work if the assumptions contain premises again, i.e., if they are of the form $\ldots \Longrightarrow \ldots$.

Alternatively, if you have used `val` *prems* = `goal thy`... and now you want to insert *prems* into the premises of your current subgoal n, you can use `by` (`cut_facts_tac` *prems* n) (see [1, Section 3.2.2]).

You should also look in the manual [1, Section 2.1.1] what `premises` does.

Some common derived rules of propositional logic

$$\dfrac{P \wedge Q \qquad \begin{array}{c}[P,Q]\\ \vdots \\ R\end{array}}{R}\ \wedge\text{-}E \qquad \dfrac{P \rightarrow Q \quad P \qquad \begin{array}{c}[Q]\\ \vdots \\ R\end{array}}{R}\ \rightarrow\text{-}E' \qquad \dfrac{\begin{array}{c}[P]\\ \vdots \\ \bot\end{array}}{\neg P}\ \neg\text{-}I \qquad \dfrac{\neg P \quad P}{R}\ \neg\text{-}E$$

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

Elimination tactic Elimination rules such as `conjE`, `impE`, `disjE` and `FalseE` are designed to be used in combination with `etac` (elimination tactic). In the Isabelle proofs we have constructed so far, we have 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. The elimination tactic allows to use *eliminate a connective in the premises.* We will now explain this.

Consider the proof of $A \wedge B \rightarrow A$ on Sheet 1 . We could have simply written

```
goal thy "A /\ B --> A";
br impI 1;
by (etac conjunct1 1);
```

Here, `etac` first does the same thing as `rtac`, but then it immediately proves the resulting subgoal by assumption, so that you do not even get to see it. You should go through the two proofs of $A \wedge B \rightarrow A$ again to understand that `etac` simply does the last two steps of the first proof in one go.

Note that there is the abbreviation `be`, in analogy to `br` and `ba`.

The pragmatics of elimination rules and the elimination tactic 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. This is what `be conjE` is good for.

More generally, `etac` may be useful whenever you know that the first subgoal you will get by applying a rule will be provable by assumption.

We now explain what `etac` does in the general case. We have to take into account that a rule may have several premises, not just one. When you use `rtac`, each of those premises gives rise to a new subgoal (explained above in the paragraph on the resolution tactic). Now `etac` solves the first of those premises by assumption, but the other ones still result in new subgoals. Moreover, in each of those subgoals, that assumption will be removed from the premises.

So we have a current subgoal $[\![\psi_1; \ldots; \psi_n]\!] \Longrightarrow \phi$ and a rule $[\![\phi_1; \ldots; \phi_m]\!] \Longrightarrow \phi$. The result of applying `etac` will be

$$\begin{array}{l} [\![\psi_1; \ldots; \psi_{i-1}; \psi_{i+1}; \ldots; \psi_n]\!] \Longrightarrow \phi_2 \\ \ldots \\ [\![\psi_1; \ldots; \psi_{i-1}; \psi_{i+1}; \ldots; \psi_n]\!] \Longrightarrow \phi_m \end{array} \tag{1}$$

The first subgoal is missing, and the premise ψ_i is removed in each subgoal.

Exercise 2
Prove the following theorems using `etac` 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 `etac`. ■

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 3
Mark each variable occurrence in

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

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 4
Apply the substitution $[z \leftarrow g(x)]$ to $(\forall z.\, p(y, z, g(x))) \wedge \exists x.\, r(g(z))$ following the definition of a substitution (in the chapter on "First-Order Logic" in the slides) step by step. ■

First Order Logic We now have some 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` and `EX` in `FOL`. For example, $\forall y. \exists x. p(x, y)$ is written `ALL y. EX x. p(x,y)`. Note that $(\forall x. p(x)) \rightarrow \exists x. p(x)$ has to be written `(ALL x. p(x)) --> (EX x. p(x))`, including the parentheses.

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

```
allI:        "(!!x. P(x)) ==> (ALL x. P(x))"
spec:        "(ALL x. P(x)) ==> P(x)"

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

The `!!` is the metalevel universal quantification (ProofGeneral 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 from x, i.e., without instantiating x. Whenever you apply a tactic and 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. Note that the rules with `!!` in Isabelle are the rules whose paper and pencil versions have side conditions.

Now, when you are doing proofs in first-order logic, it is crucial that you introduce metalevel universal quantification into your proofs as early as possible. In other words, rules with a side condition should be applied first. Intuitively, the idea is: if you need a proof that goes through for every x, then you must tell Isabelle as early as possible that you want it that way, so that she can properly make the later variables depend on x.

This implies that 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, `br 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 `etac`, you may guess that you should apply `etac` with this rule.

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 5
Try to prove the following theorems of first-order logic in Isabelle. State whenever this is impossible.

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

■

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:

$$\cfrac{\cfrac{A \wedge B}{B}\ \wedge\text{-}ER \qquad \cfrac{A \wedge B}{A}\ \wedge\text{-}EL}{B \wedge A}\ \wedge\text{-}I$$

We may want to replace this by

$$\frac{A \wedge B}{B \wedge A}\ \wedge - comm$$

We have seen on Sheet 1 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

```
[conjunct2,conjunct1] MRS conjI
```

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 be applied. This process is called *(forward) resolution.*

Note that `MRS` is an infix function whose arguments are a list of `thm`s and a `thm`.

Exercise 6
Prove the theorem $A \wedge B \to B \wedge A$ in Isabelle using the rule

```
([conjunct2,conjunct1] MRS conjI)
```

■

Generally, when we write

rule_list `MRS` *rule*,

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

```
[conjunct2] MRS conjI;
```

to see what happens in this case.

There is also an infix function `RS` which is a special case of `MRS` for the first argument being a one-element list. So instead of writing [*rule*$_1$] `MRS` *rule*$_2$, you can write *rule*$_1$ `RS` *rule*$_2$.

Since `([conjunct2,conjunct1] MRS conjI)` is a rule, we can use `RS` to combine it with another rule, say `impI`.

Exercise 7
Prove the theorem $A \wedge B \rightarrow B \wedge A$ in Isabelle using a combination of

```
([conjunct2,conjunct1] MRS conjI)
```

and `impI` with `RS`. ■

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"
subst:        "[| a=b;  P(a) |] ==> 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 8
Derive rules that state that = is *symmetric* and *transitive*:

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

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 might contain `b`; in particular, `a = b` is such a formula. Both proofs are extremely short (less than 5 lines). ■

Instantiation Tactic We have mentioned that when you apply `rtac`, 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 `[|ALL x. p(s(x)) |] ==> p(s(s(z)))`. 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 apply `br spec`, Isabelle does not know that you need `x` to be `s(z)`. The way of telling her is by using

```
by (res_inst_tac [("x","s(z)")] spec 1);
```

The first argument of `res_inst_tac` is a list of pairs of strings. For each pair, the first component should be the name of some variable occurring in the rule (in this case `spec`), and the second component should be a term or formula to which you want to instantiate this variable.

Exercise 9
Prove the following theorem of first-order logic in Isabelle:

$$s(s(s(s(s(zero))))) = \mathit{five} \wedge p(zero) \wedge (\forall x.p(x) \rightarrow p(s(x))) \rightarrow p(\mathit{five})$$

As an alternative, try doing it without instantiation tactic. ■

There is also `eres_inst_tac` that is an analogous generalization of `etac`. There is also `instantiate_tac`, which can be used to instantiate any metavariable in the entire current proof state [1, Section 3.1.4].

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 10
Write out the following λ-terms with full bracketing and without iterated λ's:

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

■

Exercise 11
Rewrite the following λ-term using left-associativity and iterated λ's:

1. $((\lambda x.\, (\lambda y.\, (\lambda z.\, (((zy)z)(w(xz))))))(\lambda x.\, (xx)))$
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 12
Reduce the terms $(\lambda xy.\, (\lambda z.\, zw)(\lambda w.\, w)x)y$, $(\lambda x.\, xy)(\lambda y.\, y)$, and $(\lambda xyz.\, zyx)(\lambda x.\, x)((\lambda w.\, x)c)$ to β-normal form (on paper), underlining the redex in each step. ■

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

Exercise 13
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 14
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. $(E\ (E\ 3\ 4)\ \top)$.
2. $P\ 4\ (P\ \top\ 3)$.

■

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

```
Goal "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

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

has solutions

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

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:

- Choosing the right subgoal.
- `res_inst_tac`: In this context, note that in Isabelle, λ is written `%`.
- `RS` and `MRS`: 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. Note that `back` has type `unit -> unit`, meaning that you have to call it with `()` as argument.

Tacticals Tacticals are operations on tactics. They play an important role in automating proofs in Isabelle. But they also fit in well here because they are helpful for finding the right unifier (see above).

The most basic tacticals are `THEN` and `ORELSE`. Both of those tacticals are of type `tactic` $*$ `tactic` $\rightarrow$ `tactic` and are written infix: tac_1 `THEN` tac_2 applies tac_1 and then tac_2, while tac_1 `ORELSE` tac_2 applies tac_1 if possible and otherwise applies tac_2 (see [1, Chapter 4]). Note that if you invoke

`by` tac_1 `THEN` tac_2;

and tac_1 happens to create a branch-point, and afterwards tac_2 fails, Isabelle will backtrack to this branch-point and try another possibility. Thus `THEN` is another way of tackling the problem that unifications are not unique.

Exercise 15
Derive (in `FOL`)

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

using

1. `res_inst_tac`, where you instantiate the metavariable ?`P` occurring in rule `subst`;
2. `MRS`;
3. `THEN`.

■

References

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

Computer Supported Modeling and Reasoning II Semestre
A.A.2009/2010
Solutions to Exercise Sheet No. 1

Exercise 1

1.

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^1}{B}\;\wedge\text{-}\mathit{ER} \qquad \dfrac{[A \wedge B]^1}{A}\;\wedge\text{-}\mathit{EL}}{B \wedge A}\;\wedge\text{-}\mathit{I}}{A \wedge B \to B \wedge A}\;\to\text{-}\mathit{I}^1$$

2.

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^2}{B}\;\wedge\text{-}\mathit{ER}}{B \vee A}\;\vee\text{-}\mathit{IL}}{A \wedge B \to B \vee A}\;\to\text{-}\mathit{I}^2$$

3.

$$\dfrac{\dfrac{[A \vee B]^3 \qquad \dfrac{[A]^4}{B \vee A}\;\vee\text{-}\mathit{IR} \qquad \dfrac{[B]^4}{B \vee A}\;\vee\text{-}\mathit{IL}}{B \vee A}\;\vee\text{-}\mathit{E}^4}{A \vee B \to B \vee A}\;\to\text{-}\mathit{I}^3$$

4.

$$\dfrac{\dfrac{\dfrac{\dfrac{[(A \wedge B) \wedge C]^5}{A \wedge B}\;\wedge\text{-}\mathit{EL}}{A}\;\wedge\text{-}\mathit{EL} \qquad \dfrac{\dfrac{\dfrac{[(A \wedge B) \wedge C]^5}{A \wedge B}\;\wedge\text{-}\mathit{EL}}{B}\;\wedge\text{-}\mathit{ER} \qquad \dfrac{[(A \wedge B) \wedge C]^5}{C}\;\wedge\text{-}\mathit{ER}}{B \wedge C}\;\wedge\text{-}\mathit{I}}{A \wedge (B \wedge C)}\;\wedge\text{-}\mathit{I}}{(A \wedge B) \wedge C \to A \wedge (B \wedge C)}\;\to\text{-}\mathit{I}^5$$

5.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A \to B \to C]^6 \quad [A]^8}{B \to C}\;\to\text{-}\mathit{E} \qquad \dfrac{[A \to B]^7 \quad [A]^8}{B}\;\to\text{-}\mathit{E}}{C}\;\to\text{-}\mathit{E}}{A \to C}\;\to\text{-}\mathit{I}^8}{(A \to B) \to (A \to C)}\;\to\text{-}\mathit{I}^7}{(A \to B \to C) \to (A \to B) \to (A \to C)}\;\to\text{-}\mathit{I}^6$$

6. Let $\phi \equiv (A \vee B) \wedge (B \vee C)$:

$$\dfrac{\dfrac{\dfrac{[\phi]^9}{A \vee B}\;\wedge\text{-}\mathit{EL} \qquad \dfrac{\dfrac{[\phi]^9}{B \vee C}\;\wedge\text{-}\mathit{ER} \qquad \dfrac{[B]^{11}}{B \vee (A \wedge C)}\;\vee\text{-}\mathit{IL} \qquad \dfrac{\dfrac{[A]^{10} \quad [C]^{11}}{A \wedge C}\;\wedge\text{-}\mathit{I}}{B \vee (A \wedge C)}\;\vee\text{-}\mathit{IR}}{B \vee (A \wedge C)}\;\vee\text{-}\mathit{E}^{11} \qquad \dfrac{[B]^{10}}{B \vee (A \wedge C)}\;\vee\text{-}\mathit{IL}}{B \vee (A \wedge C)}\;\vee\text{-}\mathit{E}^{10}}{\phi \to B \vee (A \wedge C)}\;\to\text{-}\mathit{I}^9$$

■

Exercise 3
In the following, free variables are underlined, and bound variables have an index indicating which quantifier binds them:

$$\forall_1 z.\, q(z_1) \vee p(b)) \vee p(c) \rightarrow p(\underline{x}) \wedge \forall_2 x.\, \exists_3 y.\, r(x_2, f(\underline{z}, b), g(\underline{z}, x_2), g(b, x_2))$$

■

Exercise 4
Here is the definition of a substitution from the lecture slides:

1. $x[x \leftarrow t] = t$;
2. $y[x \leftarrow t] = y$ if y is a variable other than x;
3. $f(t_1, \ldots, t_n)[x \leftarrow t] = f(t_1[x \leftarrow t], \ldots, t_n[x \leftarrow t])$ (where f is a function symbol, $n \geq 0$);
4. $p(t_1, \ldots, t_n)[x \leftarrow t] = p(t_1[x \leftarrow t], \ldots, t_n[x \leftarrow t])$ (where p is a predicate symbol, possibly $\bot$);
5. $(\neg\psi)[x \leftarrow t] = \neg(\psi[x \leftarrow t])$
6. $(\psi \circ \phi)[x \leftarrow t] = (\psi[x \leftarrow t] \circ \phi[x \leftarrow t])$ (where $\circ \in \{\wedge, \vee, \rightarrow\}$);
7. $(\mathsf{Q}x.\psi)[x \leftarrow t] = \mathsf{Q}x.\psi$ (where $\mathsf{Q} \in \{\forall, \exists\}$);
8. $(\mathsf{Q}y.\psi)[x \leftarrow t] = \mathsf{Q}y.(\psi[x \leftarrow t])$ (where $\mathsf{Q} \in \{\forall, \exists\}$) if $y \neq x$ and $y \notin FV(t)$;
9. $(\mathsf{Q}y.\psi)[x \leftarrow t] = \mathsf{Q}z.(\psi[y \leftarrow z][x \leftarrow t])$ (where $\mathsf{Q} \in \{\forall, \exists\}$) if $y \neq x$ and $y \in FV(t)$ where z is a variable such that $z \notin FV(t)$ and $z \notin FV(\psi)$.

Now the exercise:

$$\begin{array}{ll}
(\forall z.p(y,z,g(x)) \wedge \exists x.\, r(g(z)))[z \leftarrow g(x)] = & (6)\\
(\forall z.p(y,z,g(x)))[z \leftarrow g(x)] \wedge (\exists x.\, r(g(z)))[z \leftarrow g(x)] = & (7)\\
(\forall z.p(y,z,g(x))) \wedge (\exists x.\, r(g(z)))[z \leftarrow g(x)] = & (9)\\
(\forall z.p(y,z,g(x))) \wedge (\exists u.\, r(g(z))[x \leftarrow u][z \leftarrow g(x)]) = & (4)\\
(\forall z.p(y,z,g(x))) \wedge (\exists u.\, r(g(z)[x \leftarrow u])[z \leftarrow g(x)]) = & (3)\\
(\forall z.p(y,z,g(x))) \wedge (\exists u.\, r(g(z[x \leftarrow u]))[z \leftarrow g(x)]) = & (2)\\
(\forall z.p(y,z,g(x))) \wedge (\exists u.\, r(g(z))[z \leftarrow g(x)]) = & (4)\\
(\forall z.p(y,z,g(x))) \wedge (\exists u.\, r(g(z)[z \leftarrow g(x)])) = & (3)\\
(\forall z.p(y,z,g(x))) \wedge (\exists u.\, r(g(z[z \leftarrow g(x)]))) = & (1)\\
(\forall z.p(y,z,g(x))) \wedge (\exists u.\, r(g(g(x)))) &
\end{array}$$

We have to apply rule 9 in line 3 because $x \neq z$ and $x \in FV(g(x))$. We may choose u as a new variable name for the bound variable x because $u \notin FV(g(x))$ and $u \notin FV(r(g(z)))$. ■

Exercise 5
It is not possible to prove number 3, $(\forall x.(p(x) \vee q(x))) \rightarrow ((\forall x.q(x)) \vee (\forall x.p(x)))$ and number 5, $(\forall x.\exists y.p(x,y)) \rightarrow (\exists y.\forall x.p(x,y))$. Both are not valid. ■

Exercise 8
We use here the names of the rules as they were introuduced in lecture "First-Order Logic with Equality".
Symmetry:

$$\dfrac{x = y \qquad \dfrac{}{x = x}\,\mathit{refl}}{y = x}\,\mathit{cong}_2$$

Transitivity:

$$\cfrac{\cfrac{x = y}{y = x}\,\mathit{sym} \qquad \cfrac{y = z}{z = y}\,\mathit{sym}}{\cfrac{z = x}{x = z}\,\mathit{sym}}\,\mathit{cong}_2$$

The last solution was suggested by Maria Vassileva. ■

Exercise 10

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

■

Exercise 11

1. $(\lambda xyz.\, zyz(w(xz)))(\lambda x.\, xx)$ or even $(\lambda xyz.\, zyz(w(xz)))\lambda x.\, xx$.
2. $(\lambda x.\, (\lambda w.\, v)(\lambda y.\, yz))((\lambda z.\, x)(\lambda xzu.\, xyz(uz)))$

■

Exercise 12

$$\begin{array}{l}
\underline{(\lambda xy.\, (\lambda z.\, zw)(\lambda w.\, w)x)y} \rightarrow_\beta \\
\quad \lambda y'.\, \underline{(\lambda z.\, zw)(\lambda w.\, w)}y \rightarrow_\beta \\
\quad \lambda y'.\, \underline{(\lambda w.\, w)w}y \rightarrow_\beta \\
\quad \lambda y'.\, wy.
\end{array}$$

$$\begin{array}{l}
\underline{(\lambda x.\, xy)(\lambda y.\, y)} \rightarrow_\beta \\
\quad \underline{(\lambda y.\, y)y} \rightarrow_\beta \\
\quad y.
\end{array}$$

$$\begin{array}{l}
(\lambda xyz.\, zyx)(\lambda x.\, x)(\underline{(\lambda w.\, x)c}) \rightarrow_\beta \\
\quad \underline{(\lambda xyz.\, zyx)(\lambda x.\, x)}(x) \rightarrow_\beta \\
\quad \underline{(\lambda yz.\, zy(\lambda x.\, x))(x)} \rightarrow_\beta \\
\quad \lambda z.\, z(x)(\lambda x.\, x)
\end{array}$$

■

Exercise 13
For readability, we abbreviate $\Phi = \rho \to \sigma \to \tau$ and $\Gamma = \{f : \Phi, g : \phi \to \sigma \to \rho, h : \sigma \to \phi, x : \sigma\}$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma \vdash f : \Phi}\,\mathit{hyp} \qquad \cfrac{\cfrac{\cfrac{}{\Gamma \vdash g : \phi \to \sigma \to \rho}\,\mathit{hyp} \qquad \cfrac{\cfrac{}{\Gamma \vdash h : \sigma \to \phi}\,\mathit{hyp} \quad \cfrac{}{\Gamma \vdash x : \sigma}\,\mathit{hyp}}{\Gamma \vdash hx : \phi}\,\mathit{app}}{\Gamma \vdash g(hx) : \sigma \to \rho}\,\mathit{app} \qquad \cfrac{}{\Gamma \vdash x : \sigma}\,\mathit{hyp}}{\Gamma \vdash (g(hx)x) : \rho}\,\mathit{app}}{\Gamma \vdash f(g(hx)x) : \sigma \to \tau}\,\mathit{app} \qquad \cfrac{}{\Gamma \vdash x : \sigma}\,\mathit{hyp}}{\Gamma \vdash f(g(hx)x)x : \tau}\,\mathit{app}}{f : \Phi, g : \phi \to \sigma \to \rho, h : \sigma \to \phi \vdash \lambda x^\sigma.\, f(g(hx)x)x : \sigma \to \tau}\,\mathit{abs}}{f : \Phi, g : \phi \to \sigma \to \rho \vdash \lambda h^{\sigma \to \phi} x^\sigma.\, f(g(hx)x)x : (\sigma \to \phi) \to \sigma \to \tau}\,\mathit{abs}}{\cfrac{f : \Phi \vdash \lambda g^{\phi \to \sigma \to \rho} h^{\sigma \to \phi} x^\sigma.\, f(g(hx)x)x : (\phi \to \sigma \to \rho) \to (\sigma \to \phi) \to \sigma \to \tau}{\lambda f^\Phi g^{\phi \to \sigma \to \rho} h^{\sigma \to \phi} x^\sigma.\, f(g(hx)x)x : (\Phi) \to (\phi \to \sigma \to \rho) \to (\sigma \to \phi) \to \sigma \to \tau}\,\mathit{abs}}\,\mathit{abs}$$

■

Exercise 14

1. Let $\Theta_1 = [\alpha \leftarrow \mathit{bool}]$ and $\Theta_2 = [\alpha \leftarrow \mathbb{N}]$.

$$\dfrac{\dfrac{\dfrac{\Theta_1}{\vdash E : \mathit{bool} \to \mathit{bool} \to \mathit{bool}}\,\mathit{assum} \qquad \dfrac{\dfrac{\dfrac{\Theta_2}{\vdash E : \mathbb{N} \to \mathbb{N} \to \mathit{bool}}\,\mathit{assum} \quad \dfrac{}{3 : \mathbb{N}}\,\mathit{assum}}{\vdash (E\ 3) : \mathbb{N} \to \mathit{bool}}\,\mathit{app} \quad \dfrac{}{4 : \mathbb{N}}\,\mathit{assum}}{(E\ 3\ 4) : \mathit{bool}}\,\mathit{app}}{\vdash E\ (E\ 3\ 4) : \mathit{bool} \to \mathit{bool}}\,\mathit{app} \qquad \dfrac{}{\vdash \top : \mathit{bool}}\,\mathit{assu}}{\vdash (E\ (E\ 3\ 4)\ \top) : \mathit{bool}}\,\mathit{app}$$

2. Let $\Theta_1 = [\alpha \leftarrow \mathbb{N}, \beta \leftarrow (\mathit{bool}, \mathbb{N})\mathit{pair}]$ and $\Theta_2 = [\alpha \leftarrow \mathit{bool}, \beta \leftarrow \mathbb{N}]$.

$$\dfrac{\dfrac{\dfrac{(\Theta_1)}{P : \mathbb{N} \to (\mathit{bool}, \mathbb{N})\mathit{pair} \to (\mathbb{N}, (\mathit{bool}, \mathbb{N})\mathit{pair})\mathit{pair}}\,\mathit{assum} \quad \dfrac{}{4 : \mathbb{N}}\,\mathit{assum}}{\vdash P4 : (\mathit{bool}, \mathbb{N})\mathit{pair} \to (\mathbb{N}, (\mathit{bool}, \mathbb{N})\mathit{pair})\mathit{pair}}\,\mathit{app} \qquad \dfrac{\dfrac{\dfrac{(\Theta_2)}{P : \mathit{bool} \to \mathbb{N} \to (\mathit{bool}, \mathbb{N})\mathit{pair}}\,\mathit{assum} \quad \dfrac{}{\top : \mathit{bool}}\,\mathit{assum}}{P\,\top : \mathbb{N} \to (\mathit{bool}, \mathbb{N})\mathit{pair}}\,\mathit{app} \quad \dfrac{}{3 : \mathbb{N}}\,\mathit{assum}}{\vdash (P\,\top 3) : (\mathit{bool}, \mathbb{N})\mathit{pair}}\,\mathit{app}}{\vdash P4(P\,\top 3) : (\mathbb{N}, (\mathit{bool}, \mathbb{N})\mathit{pair})\mathit{pair}}\,\mathit{app}$$

■

Universität Freiburg
Institut für Informatik
Lehrstuhl für Grundlagen der Künstlichen Intelligenz
PD Dr. Jan-Georg Smaus

Computer Supported Modeling and Reasoning II Semestre A.A.2009/2010 Exercise Sheet No. 2 (22nd March 2010)

Definitions in HOL The basic inference rules of HOL involve only the constants $=$, $\rightarrow$, 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:

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

Consider *True*: The term $\lambda x^{Bool}.x = \lambda x.x$ 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 $\alpha \rightarrow Bool$ (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 $\lambda x.\mathit{True}$.

Exercise 1
Think about similar intuitive explanations for the other definitions. You may also have a look at the chapter "HOL Foundations" in the lecture. ■

Basic HOL In this lecture we derive all well-known inference rules for logical connectives and quantifiers in HOL. 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, I am asking you to pretend that the rules are not there.

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.

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

Exercise 2
Derive the following rules (pretending they do not exist already!):

1. $f = g \Longrightarrow f(x) = g(x)$ (*my_fun_cong*)
2. $x = y \Longrightarrow f(x) = f(y)$ (*my_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 3
Derive the following rules from the lecture (pretending they do not exist already!):

1. $s = t \Longrightarrow t = s$ $\quad$ (*my_sym*)
2. *True* $\quad$ (*my_TrueI*)
3. $(\bigwedge x. P\, x) \Longrightarrow \forall x. P\, x$ $\quad$ (*my_allI*)
4. *False* $\Longrightarrow P$ $\quad$ (*my_FalseE*)
5. $[\![P; Q]\!] \Longrightarrow P \wedge Q$ $\quad$ (*my_conjI*)

■

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. If Isabelle is currently running, you should quit and restart it (also using menus). The `HOL` theory file can be found on the course's webpage. There you will also find some other files contained in the standard HOL library.

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 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.

Recall `simpset`, `addsimps`, `delsimps`, `simp_tac`, `asm_simp_tac` (and their upper-case counterparts) from lecture "Term Rewriting".

`print_ss (simpset());` will print the printable contents of the current simpset. In particular, you will see all the rewrite rules.

Tracing `set trace_simp;` or `trace_simp := true;` will set the tracer, so that you can look at the single steps of the rewriting process. In order to look at this output, you may have to switch to the *Trace buffer*. This is done by selecting "Proof-General → Buffers → Trace". However, we advise you to turn off the tracer most of the time, since it will slow down Isabelle a lot and may even cause her to get hung up, probably due to some exhaustion of resources.

For more details on the simplifier, you should look at [2, Chapter 10]. In the following exercise, you will find that using a combination of simplification and `blast_tac` (or `fast_tac` if you like) is effective.

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

6. $[\![\neg E \Longrightarrow B \neq B'; \neg E]\!] \Longrightarrow (\textit{if } E \textit{ then } A \textit{ else } B) \neq (\textit{if } E \textit{ then } A \textit{ else } B')$

7. $A \vee B \Longrightarrow (\textit{if } P \textit{ then } A \textit{ else } B)$

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

Sets We will now have some exercises concerning sets. Sets are defined in `Set.thy`. One might get confused about syntax here. There is the ASCII syntax vs. the X-symbol syntax. Then there is the distinction between the higher-order abstract syntax used in operator declarations, e.g. `Ball A S`, and its concrete counterpart `ALL x:A. S x` (declared like this in `Set.thy`), which would then be displayed by X-symbols (and in the lecture slides) as $\forall x \in A.S\ x$.

Exercise 5
Prove the following three lemmas from lecture "Sets":

1. $P\ a \Longrightarrow a \in \textit{Collect } P$ (`CollectI`)
2. $a \in \textit{Collect } P \Longrightarrow P\ a$ (`CollectD`)
3. $(\forall x.(x \in A) = (x \in B)) \Longrightarrow A = B$ (`set_ext`)

Hints and notes: You will have to use the axioms in `Set.thy` which state the isomorphism between characteristic functions and sets (see lecture "Sets"). You may use any lemmas proven in `HOL.*`, but *not* the lemmas in `Set` (since that would undermine the purpose of this exercise). You may not use `fast_tac`, `blast_tac`, `auto` etc.! For the last lemma, another lemma called `box_equals` is helpful. First make a sketch of the proofs by hand (they are not very complicated). Use `res_inst_tac` or `RS` or other means to ensure that Isabelle finds the right unifier, since this will be the crucial problem here. ■

Exercise 6
Solve the following goals involving sets, using as much automation as you like.

1. $[\![\forall x \in A.x \in B; \forall x \in B.x \in C]\!] \Longrightarrow \forall x \in A.x \in C$
2. $A \cap B \subseteq A$
3. $(A = B) = ((\textit{Pow } A) = (\textit{Pow } B))$
4. $(A \subseteq B) = (A \in \textit{Pow } B)$
5. $X \subseteq \textit{insert } x\ X$
6. $[\![X \subseteq Y; X \neq \{\}]\!] \Longrightarrow (Y - X) \subset Y$
7. $x \neq y \Longrightarrow \{x\} \neq \{x, y\}$
8. $[\![S = \{a, b\}; a \neq b]\!] \Longrightarrow$ $\exists A : \textit{Pow } S.(\exists B : \textit{Pow } S.(\exists C : \textit{Pow } S.(\exists D : \textit{Pow } S.$ $A \neq B \wedge A \neq C \wedge A \neq D \wedge B \neq C \wedge B \neq D \wedge C \neq D)))$
9. $\textit{range}(\lambda x.\textit{False}) = \{\textit{False}\}$
10. $f\ `\ \{\} = \{\}$
11. $(A \neq \{\}) = (((\lambda x.\textit{True})\ `\ A) = \{\textit{True}\})$
12. $\bigcup\{\} = S$, where S is a term you should choose appropriately.

■

Finite Sets In lecture "Least Fixpoints", the set of finite subsets of A was defined as $\mathit{lfp}\ F$ where

$$F = \lambda X.\{\{\}\} \cup \bigcup x \in A.((\mathit{insert}\ x)`X)$$

(recall that ` is nice syntax for *image*, whose type is $[\alpha \Rightarrow \beta, \alpha\ \mathit{set}] \Rightarrow (\beta\ \mathit{set})$).

We shall try to understand this better, in particular analyze the types that are involved. All the following statements are consequences of the typing rules of HOL and the definitions of the occurring constants. Note first that if A is a set then this means that A is of type $\tau\ \mathit{set}$ for some τ. Secondly note that the expression $x \in A$ forces x to be of type τ. Next, the type of $(\mathit{insert}\ x)$ is $\tau\ \mathit{set} \Rightarrow \tau\ \mathit{set}$. Next, the expression $(\mathit{insert}\ x)`X$ forces $X$ to be of type $\tau\ \mathit{set}\ \mathit{set}$, and $(\mathit{insert}\ x)`X$ is also of type $\tau\ \mathit{set}\ \mathit{set}$. Next, $\bigcup x \in A.((\mathit{insert}\ x)`X)$ is also of type $\tau\ \mathit{set}\ \mathit{set}$ (see the declaration of `UNION`), and the same holds for $\{\{\}\} \cup \bigcup x \in A.((\mathit{insert}\ x)`X)$. Next, F is of type $\tau\ \mathit{set}\ \mathit{set} \Rightarrow \tau\ \mathit{set}\ \mathit{set}$. Finally, $\mathit{lfp}\ F$ is of type $\tau\ \mathit{set}\ \mathit{set}$, which is what we expect.

Such informal type checking is usually helpful to understand what an expression does and whether it is what one intends.

In the following, we assume that we have a type *nat* (so $\tau = \mathit{nat}$) containing constants $0, 1, 2, \ldots$.

Exercise 7
Let *Ev* be the *set* of even numbers and

$$F = \lambda X.\{\{\}\} \cup \bigcup x \in \mathit{Ev}.((\mathit{insert}\ x)`X)$$

(i.e., we instantiate A by *Ev*). Give the values of the following expressions (where appropriate, use natural language to describe them):

1. $F\,\{\}$
2. $F(F\,\{\})$
3. $F(F(F\,\{\}))$
4. $F^n\,\{\} = \underbrace{F(\cdots(F}_{n}\,\{\}))$
5. $F\,\{\{7\}\}$
6. $F\,\{S \mid S \text{ is finite}\}$
7. $F\,\{S \mid S \subseteq \mathit{Ev},\ S \text{ is finite}\}$
8. $F\,\{S \mid S \subseteq \mathit{Ev}\}$
9. $F\ \mathit{UNIV}$

Can you name a few fixpoints of F? ∎

We now show that this approach can also be taken in Isabelle, although it is not the one taken in the HOL library.

Exercise 8
Load the theory `FinSet.thy` in your proof script `ex2.ML`. Type

```
open FinSet;
Fin.defs;
```

to see how the keyword `inductive` was translated to a fixpoint definition. Talking (ML-)technically, `Fin` is a *structure* (module), and `defs` is a value (component) of this structure (and `Fin` is a component of `FinSet`). Prove the following goal:

$$\{0, 1\} \in \mathit{Fin}\ \{0, 1, 2\}$$

Hints: `Fin.defs` gives you a rewrite rule to replace *Fin* by its definition. Tarski's fixpoint theorem is proven in `FixedPoint.ML` and is called `lfp_unfold`. To apply the theorem in such a way that it rewrites a goal using the equation exactly once, the tactic `stac` is useful. The tactic combines `rtac` with a use of the `subst` rule.

To show monotonicity, one of the automatically generated `thm`'s of the `FinSet` structure is useful. If you have used `lfp_unfold` the appropriate number of times, the rest of the proof is quite automatic. ■

In the above, we have right from the start distinguished between a type τ and a set A (*Ev* in the exercise) of type τ *set*, which could be thought of as a "subset" of the type τ (although of course, sets and types are formal objects of a different kind). Alternatively, one could define "the set of all finite sets whose elements have type τ". In this case, no fixed A is involved, and it is closer to what actually happens in the Isabelle library. In `Finite_Set.thy` (see the course's webpage) a constant *Finites* is defined. It has polymorphic type α *set set*. We have $A \in$ *Finites* if and only if A is a finite set. However, it would be wrong to think of *Finites* as one single set that contains all finite sets. Instead, for each τ, there is an instance of *Finites* of type τ *set set* containing all finite sets of element type τ. In `Finite_Set.thy` we find the lines

```
inductive "Finites"
  intros
    emptyI [simp, intro!]: "{} : Finites"
    insertI [simp, intro!]: "A : Finites ==> insert a A : Finites"
```

The Isabelle mechanism of interpreting the keyword `inductive` translates this into the following definition: *Finites* = *lfp* G where

$$G \equiv \lambda S.\ \{x \mid x = \{\} \vee (\exists A\ a.\ x = \mathit{insert}\ a\ A \wedge A \in S)\}$$

You can see this by typing in your proof script:

```
thm "Finites.defs";
```

Note that for reasons related to how ML identifiers are (not) bound, you cannot simply type

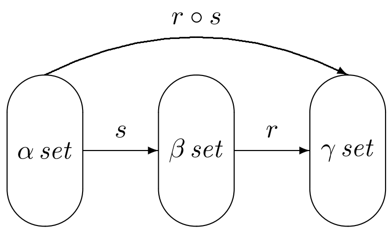
```
Finites.defs;
```

Note that the A in the above expression is existentially quantified and has nothing to do with the even numbers as in Exercise 7.

Note that there is a convenient syntactic translation

```
translations  "finite A"  ==  "A : Finites"
```

Relations In lecture "Well-Founded Recursion", relations and the standard operations about them (converse, composition, identity, etc.) were introduced. A comment on the order when writing $r \circ s$: If r is of type $(\beta \times \gamma)$ *set* and s is of type $(\alpha \times \beta)$ *set* then $r \circ s$ is of type $(\alpha \times \gamma)$ *set*. This may be illustrated as follows:



$$\begin{tikzcd} \alpha\ set \arrow[r, "s"] \arrow[rr, bend left, "r \circ s"] & \beta\ set \arrow[r, "r"] & \gamma\ set \end{tikzcd}$$

One may find this somewhat confusing but it is in analogy to function concatenation, which is also denoted using $\circ$: there, $(f \circ g)(x)$ is defined as $f(g(x))$. And: if f is of type $\beta \Rightarrow \gamma$ and g is of type $\alpha \Rightarrow \beta$ then $f \circ g$ is of type $\alpha \Rightarrow \gamma$. As such, a relation cannot be "applied" to an argument: it is not a function. However, for relations we have that

$$(r \circ s)``A = r``(s``A)$$

where A is of type α *set*. This is in perfect analogy to

$$(f \circ g)`X = f`(g`X)$$

where f, g are functions as specified above. You should compare *image* on functions, abbreviated by ‘, defined in `Set.thy`, with *Image* on relations, abbreviated by “, defined in

`Relation.thy`. The idea is the same in both cases. Note that in Isabelle, relation concatenation is written `O` (see `Relation.thy`) and function concatenation is written `o` (see `Fun.thy`).

Exercise 9
Show in Isabelle that there is an injective function $f :: ind \Rightarrow ind$ that has "two witnesses of not being surjective", i.e., two different values not reached by the function. This is the goal

$$\exists (f :: ind \Rightarrow ind)\, x\, y.\ (\forall w. f w \neq x) \wedge (\forall w. f w \neq y) \wedge x \neq y \wedge inj\, f$$

Hints: In essential places you have to provide the instantiations. Isabelle will not be able to guess them.

To make the axioms for *ind* usable, you will have to include

```
val inj_Suc_Rep = thm "inj_Suc_Rep";
```

in your proof script, and a similar line for the other axiom.

Also, in some places you should reflect about what the essential lemmas/definitions are that you need. In particular, a lemma concerning injective functions in `Fun.thy` will be useful. You should add the appropriate lemmas/definitions to the simplifier and apply `asm_simp_tac`.

There is one useful rule that requires special care: `not_sym`. You should first do `asm_simp_tac` without this rule, and then again with this rule added. This way you can avoid looping. ■

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

Exercise 10
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 `nat.recs;` gives you a scheme for defining primitive recursive functions on the natural numbers. Recall (from your knowledge of the theory of computation [1]) 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:

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

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.ML` to understand why).

Exercise 11
Define the factorial function by primitive recursion, i.e., find appropriate ϕ and γ, and prove the following two goals:

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

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

Now prove the goal

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

What instantiation do you obtain for $?x$?

Advice: have a coffee while Isabelle calculates. ■

The summation operator $\sum_{i=l}^{n} f(i)$ is standard syntax in mathematics. In lecture "Encoding Syntax", it was suggested that it should have type $\mathit{sum} : i \Rightarrow i \Rightarrow (i \Rightarrow i) \Rightarrow i$, where i was the type of individuals. In the context of HOL and for the sake of simplification, I want to define a summation operator *sumup* of type $(\mathit{nat} \Rightarrow \mathit{nat}) \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$, where $\mathit{sumup}\, f\, n$ is to stand for $\sum_{i=1}^{n} f\, i$.

Exercise 12
Define an expresion ς for the summation operator *sumup* using *nat_rec*, and test it by proving the following goals:

1. $\mathit{sumup} = \varsigma \Longrightarrow \mathit{sumup}\,(\lambda i.i)\, 5 = 15$
2. $\mathit{sumup} = \varsigma \Longrightarrow \mathit{sumup}\,(\lambda i.i + 2)\, 6 = 33$
3. $\mathit{sumup} = \varsigma \Longrightarrow \mathit{sumup}\,(\lambda i.2 * i)\, m = m * (m + 1)$

Hint: If you find it difficult to define *sumup*, you should start by defining it for a fixed f; e.g., define an expression S such that that $S\, n = \Sigma_{i=1}^{n} 2 * i$, using *nat_rec*.

Definition by recursion asks for proofs by induction. Whenever there is a statement about arbitrary m of type *nat* to be proven, it has to be proven by induction (i.e., *nat_induct*) on m.

The last exercise is similar to a well-known theorem often used as an example for induction proofs: $\forall n. \sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$. ■

References

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

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

Computer Supported Modeling and Reasoning II Semestre A.A.2009/2010 Solutions to Exercise Sheet No. 2

Exercise 1
Explanations can be found in the slides by clicking on the connectives on the slide containing the HOL definitions. ■

Exercise 7
In this exercise, it is easy to confuse $\{\}$ with $\{\{\}\}$. Recall the definition of $f\,`\,A$: it is $\{y \mid \exists x \in A.\, y = f(x)\}$. From this it follows that $f\,`\,\{\} = \{\}$.

1. $F\,\{\} = (\lambda X.\{\{\}\} \cup \bigcup x \in Ev.((insert\ x)\,`\,X))\,\{\} = \{\{\}\} \cup \bigcup x \in Ev.((insert\ x)\,`\,\{\}) = \{\{\}\} \cup \bigcup x \in Ev.(\{\}) = \{\{\}\} \cup \{\} = \{\{\}\}$
2. $F(F\,\{\}) = F(\{\{\}\}) = \{\{\}, \{0\}, \{2\} \ldots\}$
3. $F(F(F\,\{\})) = F(\{\{\}, \{0\}, \{2\} \ldots\}) = \{\{\}, \{0\}, \{2\}, \ldots, \{0, 2\}, \{0, 4\}, \ldots, \{2, 4\}, \{2, 6\}, \ldots\}$
4. $F^n\,\{\} = \bigcup_{k=0}^{n-1}\{S \mid S \subseteq Ev,\ S \text{ contains } k \text{ elements}\} = \{S \mid S \subseteq Ev,\ |S| < n\}$
5. $F\,\{\{7\}\} = \{\{\}, \{7, 0\}, \{7, 2\}, \{7, 4\} \ldots\}$
6. $F\,\{S \mid S \text{ is finite}\} = \{\{\}\} \cup \{S \mid S \text{ is finite and contains at least one even number}\}$
7. $F\,\{S \mid S \subseteq Ev,\ S \text{ is finite}\} = \{S \mid S \subseteq Ev,\ S \text{ is finite}\}$
8. $F\,\{S \mid S \subseteq Ev\} = \{S \mid S \subseteq Ev\}$
9. $F\ UNIV = \{\{\}\} \cup \{S \mid S \text{ contains at least one even number}\}$

Observe that for an arbitrary B of type *nat set set*, it is not generally true that $F\ B \not\supseteq B$, as can be seen in points 5, 6, and 9. Note moreover that if S is an element of B such that $S \neq \{\}$ and S does not contain an even number, then $S \notin F\ B$, hence $F\ B \neq B$, i.e., B cannot be a fixpoint. The reason is that for any $x \in Ev$, we have that $insert\ x\ B$ does contain an even number and so

$$F\ B = \{\{\}\} \cup \{S' \cup \{\{x\}\} \mid x \in Ev,\ S' \in B\}.$$

However, based on any B of type *nat set set*, the following is a fixpoint:

$$\{E \mid E \subseteq Ev,\ E \text{ is finite}\} \cup \{S \cup E \mid S \in B, E \subseteq Ev,\ E \neq \{\},\ E \text{ is finite}\}. \tag{2}$$

Consider $B \equiv \{S \mid S \text{ is finite}\}$; it turns out that (2) becomes the r.h.s. of point 6, so here just one application of F leads to the fixpoint. Consider $B \equiv \{\}$; it turns out that (2) becomes $\{S \mid S \subseteq Ev,\ S \text{ is finite}\}$ (this is the least fixpoint of F, see point 7). Consider $B \equiv \{S \mid S \subseteq Ev\}$; it turns out that (2) becomes $\{S \mid S \subseteq Ev\}$ (see point 8). Consider $B \equiv \{\{7\}\}$; it turns out that (2) becomes

$$\{E \mid E \subseteq Ev,\ E \text{ is finite}\} \cup \{\{7\} \cup E \mid E \subseteq Ev,\ E \neq \{\},\ E \text{ is finite}\}.$$

Consider $B \equiv Ev$; it turns out that (2) becomes

$$\{S \mid S \subseteq Ev,\ S \text{ is finite}\} \cup \{Ev\},$$

so again we have a fixpoint. ■

Universität Freiburg
Institut für Informatik
Lehrstuhl für Grundlagen der Künstlichen Intelligenz
PD Dr. Jan-Georg Smaus

Computer Supported Modeling and Reasoning II Semestre A.A.2009/2010 Exercise Sheet No. 3 (29th March 2010)

AVL trees 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, a functional programming language.

The first part is concerned with the basic definitions of the theory file. We have provided a fragment of the theory file `AVL.thy`. The fragment is called `AVL_fragment.thy`. Your task will be to fill in more constants and definitions, and then save the file under the name `AVL.thy`.

The line

```
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”. Such a definition will cause Isabelle to prove a number of `thm`'s 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.

The fragment defines a constant *height* and a definition stating that the height of a tree is the maximal number of nodes in a path from the root to a leaf. Use this as a guideline for the following exercises.

Exercise 1 (2 points)
Give a definition of a function *isin* (of result type *bool*) such that $isin\, a\, t$ holds iff a is a node label contained in tree t (note that you must also add the appropriate type declaration of *isin* in the fragment).

Make sure Isabelle accepts your theory file and test the function using simple examples. ■

Exercise 2 (3 points)
Assume that the elements of the tree belong to a type for which $\leq$ is defined (for example the type *nat*). We call a tree *ordered* if for each node labeled n, all node labels in the left subtree are smaller, and all labels in the right subtree are greater. Define a function *isord* (of result type *bool*) such that $isord\, t$ holds iff t is ordered (as above, you must also add the appropriate type declaration of *isord* in the fragment).

Again, test your function with some small examples. ■

From an algorithmical viewpoint, the definition of *isin* in Exercise 1 is inefficient. Can you suggest under which condition and how it could be made more efficient?

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 can do

```
use_thy "AVL";
open AVL;

tree.cases;
tree.distinct;
tree.induct;
tree.exhaust;
```

to see some of them.

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 now aim at implementing AVL trees in more efficient ways. For the efficient implementation to behave as the specification, we have to show that each operation behaves in the same way.

We noted that *isin* traverses the entire tree, which is unnecessary in case the tree is ordered.

Exercise 3 (1 point)
Write an efficient version of *isin* called *isin_eff* which works correctly on ordered trees (no correct behaviour is required on non-ordered trees). Insert this definition into your `AVL.thy` file and save it. ■

Exercise 4 (2 points)
Prove the following goals:

1. $\mathit{isord}\ (\mathit{Node}\ n\ l\ r) \longrightarrow x < n \longrightarrow \mathit{isin}\ x\ (\mathit{Node}\ n\ l\ r) \longrightarrow \mathit{isin}\ x\ l$ (`qed` it as `isin_ord_l`)
2. $\mathit{isord}\ (\mathit{Node}\ n\ l\ r) \longrightarrow n < x \longrightarrow \mathit{isin}\ x\ (\mathit{Node}\ n\ l\ r) \longrightarrow \mathit{isin}\ x\ r$ (`qed` it as `isin_ord_r`)
3. $\mathit{isord}\ t \longrightarrow (\mathit{isin}\ k\ t = \mathit{isin\_eff}\ k\ t)$ (`qed` it as `isin_eff_correct`)

■

The last statement shows that under certain conditions, an efficient implementation of a function is equivalent to an inefficient, "declarative" implementation.