Computer Supported Modeling and Reasoning

Dr. Jan-Georg Smaus

WS06/07

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 1 (27th October 2006)

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 do not know (S)ML, consult [1, Appendix 2] for a short overview. See also the explanations at the end of this exercise sheet.

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. However these sources can be found on the web: `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 many 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 `ex1.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 $B \wedge A \rightarrow A \wedge B$.

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

■

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

```
by (fold_goals_tac [not_def]);
```

and

```
by (rewrite_goals_tac [not_def]);
```

replaces each $\neg\psi$ by $\psi \longrightarrow \bot$ or vice versa. Note that `not_def` is an expression of type `thm`, just like `impI` or `aba` (see above).

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

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

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

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

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

Exercise 3
We show that *classical* is equivalent to the principle of the excluded middle. As above, do the proofs both using paper and pencil and in Isabelle.

1. Prove $P \vee \neg P$ (hint: make the assumptions $\neg(P \vee \neg P)$ and P).
2. Prove $P \vee \neg P \to ((\neg P \to P) \to P)$ intuitionistically.

■

Exercise 4
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 3.1) ■

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 Grundlagen der Künstlichen Intelligenz
Dr. Jan-Georg Smaus

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 2 (3rd November 2006)

From now on, all exercises will be Isabelle exercises unless it is clear from the context that they must be paper & pencil exercises or this is explicitly said.

Resolution tactic revisited Let us recall the idea of the resolution tactic. We simplify matters by ignoring the issue of *instantiation* of rules, which was treated on Sheet 1. Suppose one of our current subgoals is $[\![\psi_1; \ldots ; \psi_n]\!] \Longrightarrow \phi$, which reads: if we are able to prove ϕ assuming $\psi_1, \ldots , \psi_n$, 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 `rtac` replaces the subgoal with new subgoals

$$\begin{array}{l} [\![\psi_1; \ldots ; \psi_n]\!] \Longrightarrow \phi_1 \\ \ldots \\ [\![\psi_1; \ldots ; \psi_n]\!] \Longrightarrow \phi_m \end{array} \tag{1}$$

Think about why this is correct: If we are able to prove all $\phi_1, \ldots , \phi_m$ assuming $\psi_1, \ldots , \psi_n$, by the rule we have shown ϕ. Thus it is correct to say: If we have shown (1), we are done.

There is one further point to note. Remember that for some rules (e.g.,`impI`), the left hand side contains premises with `==>` in them. Suppose that for some $j \in \{1, \ldots , m\}$, ϕ_j has the form

$$[\![\mu_1; \ldots ; \mu_k]\!] \Longrightarrow \tilde{\phi}_j$$

In this case, the j'th subgoal of (1), which is

$$[\![\psi_1; \ldots ; \psi_n]\!] \Longrightarrow ([\![\mu_1; \ldots ; \mu_k]\!] \Longrightarrow \tilde{\phi}_j),$$

will be displayed as

$$[\![\psi_1; \ldots ; \psi_n; \mu_1; \ldots ; \mu_k]\!] \Longrightarrow \tilde{\phi}_j$$

In fact, Isabelle makes no distinction between the two forms. Semantically, this corresponds to the equivalence of $A_1 \wedge \ldots \wedge A_n \rightarrow B$ and $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$.

As an example, consider our proof of $A \rightarrow B \rightarrow A$. After one application of `rtac` with rule `impI`, the current goal is

```
A ==> B --> A
```

Now `br impI 1` will convert this into

```
[|A;B|] ==> A
```

You should think of `A` as an old assumption (it was there before) and `B` as a new assumption (it was introduced by this tactic application).

Derived Rules In lecture "Propositional Logic", we have learned that (1) the inference rules like `conjI`, `conjunct1` etc., (2) proven formulae like $A \rightarrow A$, (3) definitions of syntactic conversions like `not_def`, are expressions of type `thm`. We said that the inference rules were postulated to be true, and one can also say that the proven formulae are true. However, the proven formulae are not rules in the proper sense, since there is no $\Longrightarrow$ (which corresponds to the horizontal bar in the paper and pencil notation) in them. We will now see that it also 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`. `[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 ... $\Longrightarrow$ ....

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]).

Some common derived rules of propositional logic

$$\frac{P \wedge Q \qquad \begin{matrix}[P, Q]\\ \vdots \\ R\end{matrix}}{R}\;\wedge\text{-}E \qquad \frac{P \to Q \quad P \quad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R}\;\to\text{-}E' \qquad \frac{\begin{matrix}[P]\\ \vdots \\ \bot\end{matrix}}{\neg P}\;\neg\text{-}I \qquad \frac{\neg P \quad P}{R}\;\neg\text{-}E$$

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

Note that there is a terminological confusion about rule names as presented in the lecture and the ones in the theory files. If you want to see what a rule looks like, type its name followed by `;` in the proof-script window and the rule will be displayed.

Exercise 1
The Isabelle encoding of $\to$-E' is

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

Derive this rule in Isabelle. ∎

Actually, it already existed before. It is contained in `IFOL_lemmas.ML` (see `http://isabelle.in.tum.de/library/` and called `impE`, and it is part of the theory context of our current logic `FOL`. In `IFOL_lemmas.ML`, you will also find `conjE` ($\wedge$-E), `notI` ($\neg$-I), and `notE` ($\neg$-E).

Exercise 2

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

2. Derive the rule
$$\frac{\begin{matrix}[A]\\ \vdots\\ \neg A\end{matrix}}{\neg A}\ \mathit{classical\_dual}$$
using paper and pencil, as well as in Isabelle. Again, do it intuitionistically!

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

As above, we formalize this idea in a simplified way, by ignoring the issue of *instantiation* of rules. We have a current subgoal $[\![\psi_1; \ldots ; \psi_n]\!] \Longrightarrow \phi$ and a rule $[\![\phi_1; \ldots ; \phi_m]\!] \Longrightarrow \phi$. The `rtac` would convert this into (1), but `etac` will also solve the first of these goals immediately by assumption. But for this to be possible, some ψ_i must be identical to ϕ_1. The result of applying `etac` will be

$$\begin{array}{l} [\![\psi_1; \ldots ; \psi_{i-1}; \psi_{i+1}; \ldots ; \psi_n]\!] \Longrightarrow \phi_2 \\ \ldots \\ [\![\psi_1; \ldots ; \psi_{i-1}; \psi_{i+1}; \ldots ; \psi_n]\!] \Longrightarrow \phi_m \end{array} \qquad (2)$$

You should compare this to (1). The first subgoal is missing, and the premise ψ_i is removed in each subgoal.

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

Exercise 3
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. $((A \wedge B) \wedge C) \wedge D \rightarrow (B \wedge C) \wedge (A \wedge D)$
3. $((A \vee B) \vee C) \vee D \rightarrow (B \vee C) \vee (A \vee D)$
4. $(A \vee B) \wedge (A \vee C) \rightarrow A \vee (B \wedge C)$

You may want to compare this to proofs without using `etac`. ■

Generally, you will find that proofs become more complicated, and you can probably not oversee an entire proof all at once. But you should be able to get a local view of the subgoal you are proving. There are certain heuristics, some of which are suggested in our explanations this week. Using those heuristics, most exercises of this week can be solved quite easily.

A general remark about which subgoal should be selected first: 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.

References

[1] L. C. Paulson. *The Isabelle Reference Manual.* Computer Laboratory, University of Cambridge, October 2005.

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

Computer Supported Modeling and Reasoning WS06/07 Exercise Sheet No. 3 (10th November 2006)

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. These notions are defined by induction on the structure of terms/formulae. This is why the following definition is along the lines of our definition of terms and formulae.

1. The (only) occurrence of x in the term x is a free occurrence of x in x;
2. the free occurrences of x in $f(t_1, \dots, t_n)$ are the free occurrences of x in $t_1, \dots, t_n$;
3. there are no free occurrences of x in $\bot$;
4. the free occurrences of x in $p(t_1, \dots, t_n)$ are the free occurrences of x in $t_1, \dots, t_n$;
5. the free occurrences of x in $\neg\phi$ are the free occurrences of x in ϕ;
6. the free occurrences of x in $\psi \circ \phi$ are the free occurrences of x in ψ and the free occurrences of x in ϕ ($\circ \in \{\wedge, \vee, \rightarrow\}$);
7. the free occurrences of x in $\forall y.\, \psi$, where $y \neq x$, are the free occurrences of x in ψ; likewise for $\exists$;
8. x has no free occurrences in $\forall x.\, \psi$; in $\forall x.\, \psi$, the (outermost) $\forall$ binds all free occurrences of x in ψ; the occurrence of x next to $\forall$ is a *binding* occurrence of x; likewise for $\exists$.

A variable occurrence is *bound* if it is not free and not binding. We define

$$FV(\phi) := \{x \mid x \text{ has a free occurrence in } \phi\}.$$

Exercise 1
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. ■

Substitution The notation $s[x \leftarrow t]$ denotes the expression (term, formula) obtained by substituting t for x in s. However, a substitution $[x \leftarrow t]$ replaces only the free occurrences of x in the expression that it is applied to. A substitution is defined as follows:

1. $x[x \leftarrow t] = t$;
2. $y[x \leftarrow t] = y$ if y is a variable other than x;
3. $f(t_1, \dots, t_n)[x \leftarrow t] = f(t_1[x \leftarrow t], \dots, t_n[x \leftarrow t])$ (where f is a function symbol, $n \geq 0$);
4. $p(t_1, \dots, t_n)[x \leftarrow t] = p(t_1[x \leftarrow t], \dots, t_n[x \leftarrow t])$ (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 $z \notin FV(\psi)$.

So you have to be careful about two points when applying a substitution: the substitution must not replace bound occurrences of a variable, and it must be avoided that the substitution introduces a variable that is intended to be free into a context where this variable is bound.

Exercise 2
Apply the substitution $[z \leftarrow f(x)]$ to $(\exists z.\, q(y, z, g(x))) \wedge \forall x.\, r(g(z))$ following the definition of a substitution 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 $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. In `IFOL.thy`, we have

```
iff_def:       "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 `rewrite_goals_tac` and `fold_goals_tac` to convert between the two syntactic forms.

Exercise 3
Derive the rule

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

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

Hint: `etac` may be useful, but it can be done without. ■

First Order Logic Concerning the semantics of first oder logic, you should review the following concepts: *structure*, *satisfiability*, *validity*.

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.

The introduction and elimination rules for the quantifiers are as follows:

$$\dfrac{P(x)}{\forall x.\, P(x)}\ \forall\text{-}I^{*} \qquad \dfrac{\forall x.\, P(x)}{P(t)}\ \forall\text{-}E \qquad \dfrac{A(t)}{\exists x.\, A(x)}\ \exists\text{-}I \qquad \dfrac{\exists x.\, A(x) \qquad \begin{matrix}[A(x)] \\ \vdots \\ B\end{matrix}}{B}\ \exists\text{-}E^{**}$$

Side conditions:

* x is not free in any assumption on which $P(x)$ depends.

** x is not free in B or any assumption of the subderivation of B other than $A(x)$.

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. 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 & 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 4
Prove the following theorems of first-order logic in Isabelle:

1. $(\forall x.p(x)) \to \exists x.p(x)$
2. $((\forall x.p(x)) \lor (\forall x.q(x))) \to (\forall x.(p(x) \lor q(x)))$
3. $(\forall x.(p(x) \land q(x))) \leftrightarrow ((\forall x.p(x)) \land (\forall x.q(x)))$
4. $(\exists x.\forall y.p(x, y)) \to (\forall y.\exists x.p(x, y))$
5. $(\forall x.p(x)) \to (\forall x.p(f(x)))$

What about $(\forall x.(p(x) \lor q(x))) \to ((\forall x.p(x)) \lor (\forall x.q(x)))$? Can you prove it? ■

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

In lecture "First-Order Logic" it was 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 $p(x) \to \forall x.p(x)$, which is not valid. You may want to attempt a proof of $p(x) \to \forall x.p(x)$ 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
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). ■

Miscellaneous Isabelle commands In the following, we will refer to [1] (you will find this on this course's webpages). You are encouraged to look there for more detail.

In order to reuse theorems that you have proven previously in the file, say, `scriptA.ML`, in file `scriptB.ML`, you should invoke the command `use "scriptA.ML";` [1, Section 1.3].

Instead of `goal` or `Goal`, you can also use `goalw` or `Goalw`, which will take a list of rewrite rules as additional argument (just type `goalw;` and `Goalw;` to see the type of those functions) and apply these rewrite rules to the first subgoal and the premises [1, Section 2.1.1].

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

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

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

```
[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 `thms` and a `thm`.

Exercise 7
Prove the theorem $A \wedge B \rightarrow B \wedge A$ 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 8
Prove the theorem $A \wedge B \rightarrow B \wedge A$ in Isabelle using a combination of

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

and `impI` with `RS`. ∎

Actually, using `MRS` (or `RS`), 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 `MRS` expression corresponds to a subtree. You can look at a tree and construct a corresponding `MRS` 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 `MRS` expressions corresponding to those subtrees are $E_1, \ldots, E_n$. Then the `MRS` expression for the subtree at this horizontal bar is simply

$$[E_1, \ldots, E_n] \texttt{ MRS } \mathit{rule}$$

Exercise 9
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 `MRS` (or possibly `RS`). Other than that, only applications of `atac` are allowed. ∎

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.

Exercise 10
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 & pencil. Concerning the paper & pencil proof, I allow you to cheat and derive

$$\frac{y = z \quad x = y}{x = z}\,\mathit{trans}'$$

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). ∎

References

[1] L. C. Paulson. *The Isabelle Reference Manual.* Computer Laboratory, University of Cambridge, October 2005.

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 4 (17th November 2006)

This week we will be using several Isabelle theories. You should have proof scripts named `ex4_FOL.ML`, `ex4_ONE_AXIOM.ML`, `ex4_ONE_RULE.ML`, and `ex4_NSet.ML`.

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 1
Prove the following theorem of first-order logic in Isabelle:

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

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].

Axioms vs. Rules In lecture "First-Order Theories", we have learned that when formalizing a theory, one sometimes has a choice between introducing axioms and introducing (proper) rules. This works when the axioms have a particular form, namely they are *Horn clauses*, i.e., formulae of the form $\forall x_1 \dots x_n.A_1 \wedge \dots \wedge A_m \rightarrow B$ where $A_1, \dots, A_m, B$ are atoms. We illustrate this general principle with an ad-hoc example.

Exercise 2
Consider the axiom $\forall x.R(x) \wedge S(x) \rightarrow T(x)$. How would you write it as a rule?

1. I have written a small theory file `ONE_AXIOM.thy` for you, containing essentially just this axiom. Move it in your working directory and load the file by typing

   ```
   use_thy "ONE_AXIOM";
   ```

   in the proof script `ex4_ONE_AXIOM.ML`. Now derive the rule formulation.

2. I have written a fragment of a theory file `ONE_RULE_fragment.thy`. Complete this fragment by inserting the rule derived above and save the file as `ONE_RULE.thy`. Please send in the file by email.

   Restart Isabelle and type

   ```
   use_thy "ONE_RULE";
   ```

   in the proof script `ex4_ONE_RULE.ML`. Now derive $\forall x.R(x) \wedge S(x) \rightarrow T(x)$.

∎

Naïve Set Theory (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)`

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

$$\frac{P(t)}{t \in \{x \mid P(x)\}}\in\text{-}I \quad \frac{t \in \{x \mid P(x)\}}{P(t)}\in\text{-}E \quad \frac{\forall x. x \in A \leftrightarrow x \in B}{A = B}=\text{-I} \quad \frac{A = B}{\forall x. x \in A \leftrightarrow x \in B}=\text{-E}$$

In `NSet.thy`, instead of those inference rules we have two axioms `ext` and `Collect`. The Isabelle encoding of the *rule* forms can be found in `NSet.ML`. Load `NSet` (with `use_thy`) in `ex4_NSet.ML`. The file `NSet.ML` will be loaded automatically. As an aside, you may want to have a look at `NSet.ML` to see how these encodings and proofs work. We are using lemmas from `IFOL_lemmas.ML`, see `http://isabelle.in.tum.de/library/`.

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

Exercise 4
Prove $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ in Isabelle.

Hint: Have a look at `NSet.ML` and ask yourself why there are two different kinds of elimination rules. ■

Exercise 5
Prove $A \subseteq B \leftrightarrow \mathcal{P}(A) \subseteq \mathcal{P}(B)$ in Isabelle

Hint: same hint as in Exercise 4. ■

Exercise 6
Show that `NSet` is inconsistent, i.e., that $\perp$ can be derived in it.

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

```
goal thy "False";
by (res_inst_tac [("P","{A. A ~: A} :  {A. A ~: A}")] notE 1);
```

Hint: The rule *classical_dual* from Exercise 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 7
Write out the following λ-terms with full bracketing and without iterated λ's:

1. $(\lambda xyz.\, xz(yz))(\lambda xy.\, x)(\lambda xyz.\, xz(yz))$

2. $(\lambda u.\, uu)(\lambda xz.\, (\lambda v.\, x)(\lambda xy.\, xy))(c(\lambda w.\, wc))$

■

Exercise 8
Rewrite the following λ-term using left-associativity and iterated λ's:

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

■

References

[1] L. C. Paulson. *The Isabelle Reference Manual.* Computer Laboratory, University of Cambridge, October 2005.

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 5 (24th November 2006)

You should have proof scripts named `ex5_RED.ML`, `ex5_CONV.ML`, and `ex5_CNUM.ML`.

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

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

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

Exercise 2
Reduce the following terms to β-normal form in Isabelle `RED`.

1. SKK
2. SKS

Hint: You should type `S^K^K --> ?x` (for the first part). In the end, the metavariable `?x` should be instantiated to a term in β-normal form. ■

Exercise 3
Prove the following goals in Isabelle `CONV`:

1. $Y_TF = F(Y_TF)$ (here F will be a free variable, which will be replaced with a metavariable once the goal has been proven).
2. $Y_CF = F(Y_CF)$.

Is it possible to show Y_TF`-->`$F(Y_TF)$ and Y_CF`-->`$F(Y_CF)$ in `RED`? ■

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 $\lambda f x.\, \underbrace{f(f \ldots (f\, x)}_{n \text{ times}} \cdots)$, which we abbreviate by writing $\lambda f x.\, f^n x$. The successor function and addition are given by the λ-terms:

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

Exercise 4
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" $\lambda f x.\, f^n x$ and $\lambda f x.\, f^m x$). ■

The encoding of the natural numbers proposed by Alonzo Church is implemented in the theory `CNUM.thy`. In the theory file, you will also find abbreviations $C_0, \ldots$ for some small numbers.

Exercise 5
Reduce the following terms in `CNUM`:

1. $\mathit{succ}\ C_0$
2. $\mathit{add}\ C_2\ C_3$

■

Exercise 6
The existence of fixpoint combinators Y_T and Y_C (Exer. 3) 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 $Y_T\ \mathit{succ}$ 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 $(\lambda z x.\, x(z z x))(\lambda z x.\, x(z z x))$ with Y_T and $\lambda u f x.\, f(u f x)$ 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 $Z \equiv \lambda u.(\lambda f x.\, x)$. 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 7
Why is it a useful convention that function applications associate to the left whereas types associate to the right? ■

Exercise 8
Derive (on paper) a type judgement for the term $\lambda f g x.\, f\,(g\, x)\, x\, (g\, x)$ (including inserting the appropriate type superscripts for the variables f, g, x you bind with λ), as this has been done for $\lambda f x.\, f\, x\, x$ 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 9
Let $\mathcal{B} = \{\mathit{bool}/0, \mathbb{N}/0, \mathit{pair}/2\}$ (e.g., *pair* has arity 2) and $\Sigma = \langle \top : \mathit{bool}, 3 : \mathbb{N}, 4 : \mathbb{N}, P : \alpha \to \beta \to (\alpha, \beta)\ \mathit{pair} \rangle$.

Derive (using paper & pencil) an appropriate type judgement for $P\ (P\ 3\ \top)\ 4$. ■

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 Grundlagen der Künstlichen Intelligenz
Dr. Jan-Georg Smaus

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 6 (1st December 2006)

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 variable of the λ-calculus will be called *metavariable*, 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

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

has solutions

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

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 1
Derive (in `FOL`)

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

using

1. `res_inst_tac`, where you instantiate the metavariable `?P` occurring in rule `subst`;
2. `RS`;
3. `MRS`;
4. `back`
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 2
Using paper & pencil, encode $a \vee \neg b$ in $\lambda^{\to}$ and give the proof tree of the judgement $\Gamma \vdash \ulcorner a \vee \neg b)\urcorner : o$, as this has been done for $\neg a \to a$ in lecture "Encoding Syntax". What is an appropriate Γ? ■

Encoding First-Order Logic in $\lambda^{\to}$ In order to encode first-order logic, we need to extend Σ by quantifiers:

$$\Sigma = \langle not : o \to o, and : o \to o \to o, imp : o \to o \to o, all : (i \to o) \to o,\ exists : (i \to o) \to o\rangle$$

Exercise 3
Using paper & pencil, encode $\forall x.\exists y.p(f(x), y)$ in $\lambda^{\to}$ and give the proof tree of the judgement $\ldots \vdash \ulcorner \forall x.\exists y.p(f(x), y)\urcorner : o$.

Hint: to guess what the appropriate types for x, y, f and p are, consider the encoding of first-order arithmetic given in lecture "Encoding Syntax". ■

I also suggest you have a look at `IFOL.thy` (see `http://isabelle.in.tum.de/library/`) to see how the concepts of lecture "Encoding Syntax" are implemented. The keyword `consts` precedes a number of declarations of constants such as `Not`, `False`, and `All`.

Looking at the current proof state In order to look at the current proof state in Isabelle, you can type `topthm();`. Try this with any old proof in `FOL` that you have done previously. To find more commands of this kind, have a look at [1, Section 2.6.2]

"No subgoals!" This is a little digression about the status of premises used in Isabelle proofs and bound to ML identifiers. When you start a proof with `val [prem] = goal`, you might expect that once you interrupt the proof and start a new one, the value of `prem` will be forgotten, or at least, it cannot be reused in subsequent proofs. However, this is not quite so. Seemingly, such premises can later be used. It is only when you do `qed` that it will be checked if the premise actually belonged to the current proof.

Exercise 4
Try to "prove" `False` in `FOL`. You should be able to finish a proof starting with

```
Goal "False";
```

in a state saying "No subgoals!". Now try to do `qed`. ■

Automation by Proof Search In lecture "Automation by Proof Search" it has been said that construction of proofs can be automated to a large extent. There are many situations where there is no doubt about which rule should be applied; it can be applied "blindly". Such rules were referred to as *safe* rules.

We want to illustrate here that such a claim is highly non-trivial and depends on the kind of rules we have at our disposal and on the resolution techniques we use. If we limit ourselves to the rules of `FOL` and `rtac` (possibly `res_inst_tac`), as we did in the first week's exercises, such a claim would be completely unjustified.

Exercise 5
We gave the heuristic that when the top-level symbol of the conclusion of the current subgoal is $\rightarrow$, `impI` should be applied.

Prove $(\forall x.\ P(x) \leftrightarrow R(x)) \rightarrow (P(s) \rightarrow R(s))$ using only `rtac`, `atac`, and `rewrite_goals_tac`. You will see that you cannot find a proof following this heuristic. In fact, constructing the proof requires a lot of look-ahead.

Now in contrast, prove the theorem but this time following the heuristic, and using `etac` wherever possible (this implies: using `conjE`, not `conjunct1` or `conjunct2`; using `allE`, not `spec`; using `impE`, not `mp`). ■

Thus one should be aware that the rules and resolution techniques must be carefully tuned for a theory to allow for this "determinism", and years of research have gone into this. See [1, chapter 11] for more details. In the above example, note that the second application of `impI` is not particularly clever. It necessitates a later application of `impE`. But of course, one cannot expect a proof following some principles blindly to be the shortest proof in all cases.

The classical rule set (type `claset`) of `FOL` is denoted by the identifier `FOL_CS`. Type

```
print_cs FOL_cs;
```

to have a look at it. This set is crucial in the implementation of tactics like `fast_tac` (`Fast_tac`) and `blast_tac` (`Blast_tac`). More generally, typing `claset()` returns the classical rule set of the current theory (use `print_cs` to look at it).

In the following exercise, you are asked to derive one of the rules used in `FOL_CS`. For the purpose of the exercise, use only elementary proof steps (`rtac`, `atac`, `etac`). Conceptually, rules must be derived before they can be used by tactics such as `fast_tac` and `blast_tac`, and so you should convince yourself that they can be derived.

Exercise 6
Derive the rule $\vee$-*swap* presented in lecture "Automation by Proof Search": $(\neg Q \Longrightarrow P) \Longrightarrow P \vee Q$.

Hint: Use `classical` at the very beginning. Otherwise, when your current subgoal is a disjunction, find out by trying which introduction rule is the right one. ■

Negation and Swapping In addition to the rules in `FOL_CS`, one has to be aware that for dealing with negated formulae in the premises of the current subgoal, *introduction* rules are combined with `swap` using `etac`. Type `swap;` to have a look at it.

Exercise 7
Prove $(A \rightarrow B) \vee (B \rightarrow A)$ using elementary proof steps, in particular, not use the rule $\vee$-*swap* (`disjCI`) (you may however use the derived rule `excluded_middle`, and you may use `cut_facts_tac`).

Now try to prove $(A \rightarrow B) \vee (B \rightarrow A)$ using `swap` with `etac` whenever there is a negated formula on the left-hand side. You may start your proof by using $\vee$-*swap*. What is the effect of using `swap` with `etac`, and what problem arises?

Now use a different strategy: use `swap` with `etac` whenever there is a negated formula on the left-hand side, but immediately after that, use an introduction rule.

So generally, we have a formula $\neg(A \circ B)$ in the premises. Can you explain why the technique of using `swap` with `etac` followed by an introduction rule for $\circ$ is way of dealing with (i.e., decompose) $\neg(A \circ B)$?

Since this technique is standard, the classical reasoner of Isabelle uses such derived rules. You may want to read [1, Section 11.2] for further explanations.

Now prove $(A \rightarrow B) \vee (B \rightarrow A)$ using `Clarify_step_tac`. Can you explain which rule was used each time?

Give the answers to the questions as comments in your proof script. ■

Try to prove Peirce's law $((A \rightarrow B) \rightarrow A) \rightarrow A$ using `Blast_tac` and `Fast_tac`.

Exercise 8
Prove Peirce's law by applying

```
by (Clarify_step_tac 1);
```

several times. Try to figure out for each step which rule in `FOL_CS` was applied. ■

Duplicating Rules In lecture "Automation by Proof Search", we have mentioned that for dealing with quantifiers in automated proof search, one sometimes needs so-called duplicating rules. We now give a simple exercise to illustrate this point.

Exercise 9
Prove the following formula in `FOL`: $(\forall x.P(x)) \rightarrow (P(a) \wedge P(b))$ using the strategy of applying `etac` with `allE` wherever possible. Does it work? Now try rule `all_dupE` instead. ■

References

[1] L. C. Paulson. *The Isabelle Reference Manual.* Computer Laboratory, University of Cambridge, October 2005.

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 7 (8th December 2006)

Using Tacticals for β-Reduction We now continue looking at tacticals. We will work with the `RED` theory (see Sheet 5).

Exercise 1
Write a tactic to automate β-reduction and use it to reduce the following terms to β-normal form:

1. $SKKISS$
2. $SKIKISS$

Hint: Whenever you write tactics in Isabelle that involve an arbitrary number of applications of simpler tactics, you will find that termination is a big problem.

First you should write a tactic that does one proper β-reduction step. Note that according to the rules of the λ-calculus, the redex may be somewhere inside the term, in which case contraction of redices in contexts must be applied. In terms of Isabelle, this means: A proper β-reduction step consists of zero or more applications of the congruences `appr`, `appl` and `epsi`, followed by exactly one application of `beta`.

This can be implemented in Isabelle by using `fast_tac`, where the set of safe introduction rules should consist of `appr`, `appl`, `epsi`, and `beta`.

This is the beginning of a possible solution:

```
val RED_cs = empty_cs addSIs [appr,appl,epsi,beta];

(*We now write a tactic which does one proper beta-reduction step*)
fun proper_beta_step i =
  (fast_tac RED_cs i);

(*Example*)
Goal "S^K^K --> ?x";
by (rewrite_goals_tac [S_def,K_def]);
br trans 1;
by (proper_beta_step 1);
br trans 1;
. . .
```

This gives you a tactic to perform one proper β-reduction step. However, if the λ-term in your current subgoal does not reach a normal form in one β-reduction step, then the application of `proper_beta_step` must be preceded by an application of rule `trans`. Write a function that does this using `THEN`.

Write a function that applies the previous function (transitivity followed by a proper β-reduction step) as often as possible, using `REPEAT`. Note that the transitivity step will only be performed when a proper β-reduction step is possible afterwards, so if you do it right, it will not happen that you apply a transitivity step that will lead to a dead-end.

With the function defined so far, the final subgoal will have a λ-term in normal form, and the subgoal can be solved using `refl`.

Finally, you can enhance your function so that also the rewriting and folding of the definitions of S, K, etc. is automated.

To experiment, you should try out your tactics on a simple term such as SKK. ■

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).

We should clarify what the notion of *term* is for term rewriting. Do we mean the syntactic category of terms in first-order logic, as opposed to *atoms*? No! One can apply term rewriting to first-order formulae, but the distinction between function symbols, predicate symbols and logical symbols (such as $\wedge$) is irrelevant for term rewriting. 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"). So if the object logic is first-order logic, then the function symbols, predicate symbols and logical symbols would all be *constants* on the level of the representation. As syntactic sugar, we sometimes also use constants written in infix notation, and constants applied to a tuple of arguments (as opposed to Currying).

Exercise 2
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.

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 3
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 4
Consider the equational theories given by

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

(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, decide the following equations using our procedure:

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

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

■

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

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

■

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

Metalogic We now have some first exercises on lecture "Isabelle's Metalogic".

Exercise 6
To encode *Prop* or FOL in $\mathcal{M}$, the signature of $\mathcal{M}$ has to be extended by a constant *true*. How is this done in `IFOL.thy` (see `http://isabelle.in.tum.de/library/`)? Can you quote the relevant line? ■

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 Grundlagen der Künstlichen Intelligenz
Dr. Jan-Georg Smaus

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 8 (15th December 2006)

Instantiating the Proof State In lecture "Isabelle's Metalogic", it was explained that at some points, one either has to guess the right instance of a rule, or one has to generalize the resolution rule to allow for instantiation of the current goal (proof state). In particular, this concerned the rule $\wedge$-*EL* used in the proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$

Exercise 1
Using paper & pencil, do steps (5) and (6) in the proof of $A \wedge B \rightarrow (C \rightarrow A \wedge C)$ without magically guessing the right instance of $\wedge$-*EL*, but instead using a fresh copy of $\wedge$-*EL*, and using a generalized resolution rule where instantiation of goals is allowed. Explain why you need instantiation of goals. ■

Scope of $\bigwedge$ In lecture "Isabelle's Metalogic", we have shown how the proof of $(\forall z.G(z)) \rightarrow (\forall z.G(z) \vee H(z))$ can be done in $\mathcal{M}$, i.e., we have proven $[\![(\forall z.G\,z) \rightarrow (\forall z.G\,z \vee H\,z)]\!]$ in $\mathcal{M}$. Recall that at some point it was said that the meta-axiom $\forall$-*I* could have been lifted in a different way.

Exercise 2

1. Prove $(\forall z.G(z)) \rightarrow (\forall z.G(z) \vee H(z))$ in Isabelle always using `rtac` or `atac`. Compare the proof state after each step to the presentation in the lecture. What differences do you note?
2. Using paper & pencil, work out the proof in $\mathcal{M}$ as it would be if we had lifted $\forall$-*I* differently, as mentioned in the lecture. This includes saying how you lift each rule and what the result of the lifting is. It should correspond very closely to the proof in Isabelle!

■

Metavariables In lecture "Isabelle's Metalogic", it was explained that there are two kinds of free variables: ordinary variables and *metavariables*. The distinction is important for *goals*: meta-variables in goals may be instantiated by resolution, while the other free variables must not become instantiated. Once a meta-theorem is proven, all its free variables are replaced by metavariables. This means in particular that the free variables in meta-axioms are metavariables.

Exercise 3

1. Try to prove $A \rightarrow B$ in Isabelle.
2. Prove $A \rightarrow ?B$ in Isabelle in at least three different ways, i.e., obtaining three different instances for $?B$.
3. Prove $A \rightarrow A \vee B$ and $A \rightarrow A \vee ?B$ (in the most straightforward way) in Isabelle. Save the `thm`'s using `qed`. How do they differ?
4. Prove $((?A \rightarrow ?B) \rightarrow ?A) \rightarrow ?A$ using `rtac` with `impI` and `mp`, as well as `atac`. What does the "formula" $((?A \rightarrow ?B) \rightarrow ?A) \rightarrow ?A$ remind you of?

5. Prove $(?A \wedge ?B) \wedge (?C \wedge ?D) \longrightarrow (?B \wedge ?C) \wedge (?D \wedge ?A)$ using exactly the same proof that you used to prove $(A \wedge B) \wedge (C \wedge D) \to (B \wedge C) \wedge (D \wedge A)$ in Sheet 2 Exercise 3. What have you proven?

Can you comment on why it is important to distinguish metavariables from ordinary free variables? ■

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 9 (20th December 2006)

This sheet is due on 12th January 2007. If you have any questions concerning this sheet, please email them to me. I will be reading email from 4th January onwards.

Exercise 1
In lecture "HOL: Foundations", it has been said that a tuple (x, y) can be encoded as $\{\{x\}, \{x, y\}\}$. Why is that so? ■

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 will see in lecture "HOL: Deriving Rules". But now, we want to check the definitions semantically:

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

Consider *True*: The term $\lambda x^{\mathit{Bool}}.x = \lambda x.x$ 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 $\alpha \rightarrow \mathit{Bool}$ (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
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 `http://isabelle.in.tum.de/library/` and the other theory files in that directory (this is Isabelle 2005). Point out where the rules and definitions deviate from the presentation in the lecture. ■

Basic HOL The following exercises should be done after lecture "HOL: Deriving Rules". 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.

You will have to set the logic used by Isabelle to "Pure" in the same way that you set it to "FOL" at the beginning of this course.

We provide a theory file `HOL_BASIC` which contains the definitions of HOL without any pre-proven lemmas. You should create a script `ex9.ML`, where you call `use_thy "HOL_BASIC"`.

Previously, we have said that all the rules, definitions, etc. (the `thms`), of a theory are bound to ML identifiers. So one could type, e.g., `impI` and see the according rule displayed. However, due to ML technicalities, these bindings have to be performed explicitly. This is done in `HOL_BASIC.ML`. Note that `use_thy` looks for this file automatically. This is not very different from the standard Isabelle/HOL setup (see `HOL.ML` at `http://isabelle.in.tum.de/library/`).

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

Exercise 3
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 4
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. $P = \mathit{True} \Longrightarrow P$ (*eqTrueE*)
7. $P \Longrightarrow P = \mathit{True}$ (*eqTrueI*)
8. $(\bigwedge x. P\, x) \Longrightarrow \forall x. P\, x$ (*allI*)
9. $(\forall x. P\, x) \Longrightarrow P\, x$ (*spec*)
10. $\mathit{False} \Longrightarrow P$ (*FalseE*)
11. $\mathit{False} = \mathit{True} \Longrightarrow P$ (*False_neq_True*)
12. $\mathit{True} = \mathit{False} \Longrightarrow P$ (*True_neq_False*)
13. $(P \Longrightarrow \mathit{False}) \Longrightarrow \neg P$ (*notI*)
14. $[\![\neg P; P]\!] \Longrightarrow R$ (*notE*)
15. $\neg(\mathit{True} = \mathit{False})$ (*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. $P \vee \neg P$ (*excluded middle*)

26. $Q = \mathit{True} \Longrightarrow \mathit{If}\ Q\ x\ y = x$ (*ite_then*)

27. $Q = \mathit{False} \Longrightarrow \mathit{If}\ Q\ x\ y = y$ (*ite_else*)

■

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 10 (12th January 2007)

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 at `http://isabelle.in.tum.de/library/`. In the same directory you find many other files contained in the standard HOL library.

We give an overview of some of the files that are read when this theory is invoked.

- `HOL_lemmas.ML`: this file is called explicitly from `HOL.thy`. It starts with commands to bind all the rules and definitions in `HOL.thy` (e.g., `subst` or `False_def`) to ML identifiers. This is followed by the derivations of many rules. Some of those rules will look familiar from the theory `FOL`, but it also contains many other useful rules. You should have a look at this file. The rules are organized in sections, but this is only relevant for documentation purposes.
- `cladata.ML`: this sets up the classical reasoner.
- `blastdata.ML`: this file contains some setup for `blast_tac`.
- `simpdata.ML`: this sets up the generic simplifier for `HOL`.

Not all of this is different from, say `FOL.thy`, but it will become more relevant now.

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 formula 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 1
Derive the `thm`'s $(x = y) \Longrightarrow (x == y)$ and $(x == y) \Longrightarrow (x = y)$ in `HOL` and add a little explanation as comment in the theory file. Use elementary proof steps (no `auto()`, `simp_tac` etc.). ■

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 the 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 [1, 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 2
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. $[\![E \Longrightarrow A = A']\!] \Longrightarrow (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ E\ \mathit{then}\ A'\ \mathit{else}\ B)$
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. $[\![\neg E \Longrightarrow B \neq B'; \neg E]\!] \Longrightarrow (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) \neq (\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B')$
7. $(\mathit{if}\ E\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ \neg E\ \mathit{then}\ B\ \mathit{else}\ A)$
8. $f\ (\mathit{if}\ E\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ E\ \mathit{then}\ (f\ x)\ \mathit{else}\ (f\ y))$
9. $[\![(P \vee Q)]\!] \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ Q\ \mathit{then}\ B\ \mathit{else}\ A)$
10. $[\![(P \vee Q) \wedge \neg(P \wedge Q)]\!] \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ A\ \mathit{else}\ B) = (\mathit{if}\ Q\ \mathit{then}\ B\ \mathit{else}\ A)$

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

References

[1] L. C. Paulson. *The Isabelle Reference Manual.* Computer Laboratory, University of Cambridge, October 2005.

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 11 (19th January 2007)

Orders In the lecture, various type classes involving orders were introduced: `ord`, `order`, and `linorder`. Each of them is a subclass of the previous class (it is defined by adding some additional requirements).

In Isabelle, it is possible to specify the type of a term contained in a goal we want to prove. It is also possible to specify only the fact that the type belongs to a certain class. It is necessary to do this whenever the goal only holds for terms of certain types. To specify that x should have type τ, you write $x :: \tau$, and to specify that x has some type in the class $\mathcal{C}$, you write $x :: \alpha :: \mathcal{C}$.

Exercise 1
The following are some lemmas involving orders. Some of the lemmas hold for `ord`, some for `order` but not `ord`, and some only hold for `linorder`. Try to prove each of the following goals for the biggest type class possible, and if applicable, state (as a comment in your script) for which type classes the goal cannot be proven. As mentioned above, you will have to insert appropriate type annotations.

1. $x < y \Longrightarrow x \leq y$
2. $[\![x \leq y; y \leq z; z \leq x]\!] \Longrightarrow x = z$
3. $(x < y \wedge y < z \wedge z < x) = False$
4. $min\ x\ y \neq x \Longrightarrow y \leq x$
5. $min\ x\ y \neq y \Longrightarrow x \leq y$
   (a useful auxiliary lemma to show this is $((if\ P\ then\ A\ else\ B) \neq B) \Longrightarrow P)$
6. $min\ x\ y = min\ y\ x$
7. $min\ x\ (min\ y\ z) = min\ (min\ x\ y)\ z$
8. $(\neg x \leq z \vee x \leq y \vee y \leq z) \Longrightarrow min\ x\ (min\ y\ z) = min\ (min\ x\ y)\ z$
9. $mono\ (\lambda x.x)$
10. $mono\ (\lambda x.y)$

Hint: you are encouraged to look at the `HOL` sources and use anything that seems useful there. Sometimes `simp_tac` is useful, but note that you have to use it so that it will also exploit assumptions, and moreover, often the simpset will not contain the required `thm`'s to prove the goal. The reason for this is that many rules are critical for termination. Therefore, sometimes you have to insert rules into the simpset locally. For example, I once found it useful to do

```
by (asm_simp_tac (HOL_ss addsimps [order_le_less]) 1);
```

Note that `addsimps` returns a simplifier set, whereas `Addsimps` modifies the *current* simplifier set.

Inserting the right rules into the `simpset` is the key to achieving a high degree of automation in these exercises. It is helpful to use the function `thms_containing`, as in

```
thms_containing ["op <=","op |","op <"];
```

to find `thm`'s containing a particular constant.

To appreciate the termination problem: for some goals, it was clear to me that I had to use `order_antisym` and `order_trans` to prove them, but

```
by (asm_simp_tac (HOL_ss addsimps [order_antisym,order_trans]) 1);
```

caused a loop.

To tune `simptac` further, you can also add splitting rules, e.g. I once used

```
by (asm_simp_tac (HOL_ss addsimps [order_antisym,linorder_not_le]
                         addsplits [split_if]) 1);
```

You should study the reference manual and lecture "Term Rewriting" to understand what this does.

I also found the rule `swap` useful in some places. ∎

Sets We will now have some exercises concerning sets. 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 $\forall x \in A.S\ x$.

Sets are defined in `Set.thy`. Note that sets are not defined using the `typedef` syntax. Instead the isomorphism axioms, which are simpler than in the general case, have been added directly. This is explained in lecture "Conservative Theory Extensions".

Exercise 2
Prove the following three lemmas from lecture "Sets":

1. $P\ a \Longrightarrow a \in \mathit{Collect}\ P$ (`CollectI`)
2. $a \in \mathit{Collect}\ P \Longrightarrow P\ a$ (`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 3
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. $A \subseteq A \cup B$
3. $A \cap B \subseteq A$
4. $(A = B) = ((\mathit{Pow}\ A) = (\mathit{Pow}\ B))$
5. $(A \subseteq B) = (A \in \mathit{Pow}\ B)$
6. $X \subseteq \mathit{insert}\ x\ X$
7. $x \notin X \Longrightarrow X \subset \mathit{insert}\ x\ X$
8. $[\![X \subseteq Y; X \neq \{\}]\!] \Longrightarrow (Y - X) \subset Y$
9. $x \neq y \Longrightarrow \{x\} \neq \{x, y\}$
10. $$\begin{array}{l}[\![S = \{a,b\}; a \neq b]\!] \Longrightarrow \\ \quad \exists A : \mathit{Pow}\ S.(\exists B : \mathit{Pow}\ S.(\exists C : \mathit{Pow}\ S.(\exists D : \mathit{Pow}\ S. \\ \qquad A \neq B \wedge A \neq C \wedge A \neq D \wedge B \neq C \wedge B \neq D \wedge C \neq D)))\end{array}$$

11. $range(\lambda x.\, True) = \{True\}$

12. $range(\lambda x. if\ x = True\ then\ False\ else\ True) = \{True, False\}$

13. $f \text{ ` } \{\} = \{\}$

14. $(A \neq \{\}) = (((\lambda x.\, True) \text{ ` } A) = \{True\})$

■

Exercise 4
Attempt to redo Exercise 6 on Sheet 4. Why does it not work in `Set`? (Do this in your proof script, and describe in a comment what goes wrong). ■

Fixpoints A fixpoint of a function f is a value s such that $f(s) = s$. This is a common concept in mathematics, which you may have seen in calculus courses. We use arithmetic symbols and **if-then-else** with their usual meanings.

Exercise 5
Which are the fixpoints of the following two functions:

1. $f = \lambda x.\, x^2$

2. $f = \lambda x.\, x + 1$

■

We are not concerned with finding effective procedures for computing the fixpoints of arbitrary functions. We are interested in fixpoints because they are related to induction and recursion, and in this context, we are looking at fixpoints of particular functions. Essentially, a fixpoint combinator Y allows us to have one single recursive equation as foundation of recursion, as opposed to defining each recursive function with a recursive equation (recall the explanation in lecture "Background: Recursion, Induction, and Fixpoints").

The following exercise shows that the definition of a general fixpoint combinator, $Y \equiv \lambda F. F(YF)$, can be used for computing the value of a function defined as $Y\ F$, for some "appropriate" F. In lecture "Well-Founded Recursion", we will see what "appropriate" means. For non-appropriate F, there is a problem, as shall be seen in the next exercise.

Exercise 6
Let

$$\begin{array}{rcl} Y & \equiv & \lambda F. F(YF) \\ Fac & \equiv & (\lambda f.\, \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * f(n-1)) \\ Bar & \equiv & (\lambda f.\, \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * f(n+1)) \\ Foo & \equiv & \lambda f.\, \lambda x.\, f x \end{array}$$

Reduce the following terms (on paper, of course):

1. $(Y\ Fac)\ 2$

2. $(Y\ Bar)\ 2$

3. $(Y\ Foo)\ 4882273$

In the first part, how many times did you expand the definition of Y? What would have happened if you had expanded it more often?

Do Bar and Foo have fixpoints?

Hint: Expand the definitions as late as possible, i.e., only when it serves to do an application. Otherwise your expressions will become completely unreadable. ■

Finite Sets In lecture ,"Least Fixpoints", the set of finite subsets of A was defined as *lfp* F where

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

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

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 $\tau\ set$ for some τ. Secondly note that the expression $x \in A$ forces x to be of type τ. Next, the type of $(insert\ x)$ is $\tau\ set \Rightarrow \tau\ set$. Next, the expression $(insert\ x)\text{`}\,X$ forces $X$ to be of type $\tau\ set\ set$, and $(insert\ x)\text{`}\,X$ is also of type $\tau\ set\ set$ (see the declaration of `UNION`). Next, $\bigcup x \in A.((insert\ x)\text{`}\,X)$ is also of type $\tau\ set\ set$, and the same holds for $\{\{\}\} \cup \bigcup x \in A.((insert\ x)\text{`}\,X)$. Next, F is of type $\tau\ set\ set \Rightarrow \tau\ set\ set$. Finally, *lfp* F is of type $\tau\ set\ set$, 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 $\tau = nat$) containing constants $0, 1, 2, \ldots$.

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

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

(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. $F\,\{\{1\}\}$
5. $F\,\{S \mid S \text{ is finite}\}$ (please describe in natural language)
6. $F\,\{S \mid S \subseteq Ev,\ S \text{ is finite}\}$ (please describe in natural language)
7. $F\,\{S \mid S \subseteq Ev\}$ (please describe in natural language)
8. $F\ UNIV$ (please describe in natural language)

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 `ex11.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:

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

Hints: `Fin.defs` gives you a rewrite rule to replace *Fin* by its definition. Tarski's fixpoint theorem is proven in `Lfp.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 $\tau\, set$, 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 `http://isabelle.in.tum.de/library/` a constant *Finites* is defined. It has polymorphic type $\alpha\, set\, set$. We have $A \in \mathit{Finites}$ 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 $\tau\, set\, set$ 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: $\mathit{Finites} = \mathit{lfp}\ G$ 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.

Again, let us have a look at the type of this expression. The expression $\mathit{insert}\, a\, A$ forces A to be of type $\tau\, set$ for some τ and a to be of type τ. Next, $\mathit{insert}\, a\, A$ is of type $\tau\, set$, and hence x is also of type $\tau\, set$. Moreover, the expression $A \in S$ forces S to be of type $\tau\, set\, set$. The expression $\{x \mid x = \{\} \vee (\exists A\, a.\ x = \mathit{insert}\, a\, A \wedge A \in S)\}$ is of type $\tau\, set\, set$. Next, G is of type $\tau\, set\, set \Rightarrow \tau\, set\, set$, and so finally, *Finites* is of type $\tau\, set\, set$. But actually, since τ is arbitrary, we can replace it by a type variable α.

Note that there is a convenient syntactic translation

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

Exercise 9
Repeat Exercise 7 for G instead of F. Assume that we use the type instance $nat\, set\, set \Rightarrow nat\, set\, set$ of the type of G.

If Ev is the set of even numbers, give an expression for the set of all finite subsets of Ev, using *Finites*. ■

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 12 (26th January 2007)

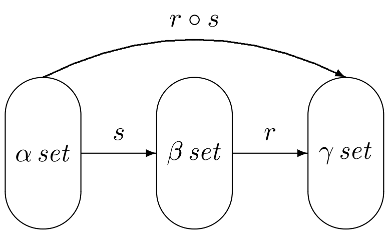
Relations In lecture "Well-Founded Recursion", relations and the standard operations about them (converse, composition, identity, etc.) were introduced. These are defined in the theory `Relation.thy`. We now discuss whether $((\alpha \times \alpha)\, set, \circ, Id, \_^{-1})$ is a group[1] i.e., whether the group axioms

$$\begin{aligned}(r \circ s) \circ t &= r \circ (s \circ t) && (1)\\ r \circ Id &= r && (2)\\ r \circ r^{-1} &= Id && (3)\end{aligned}$$

are true.

Exercise 1
Which of the group axioms hold for $((\alpha \times \alpha)\, set, \circ, Id, \_^{-1})$? ■

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 $\alpha\, set$. 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 2
Give counterexamples for the following two purported theorems:

1. $r^{-1} \circ r \subseteq Id$

[1] The first argument $(\alpha \times \alpha)\, set$ is a type, and each term of that type is a relation. Implicitly, the type $(\alpha \times \alpha)\, set$ is interpreted as the set of all terms of the type.

2. $\mathit{Id} \subseteq r^{-1} \circ r$

■

Exercise 3
Find reasonable (short!) conditions ϕ and ψ such that you can prove the following in Isabelle:

1. $\phi \Longrightarrow \mathit{Id} \subseteq r^{-1} \circ r$
2. $\neg\phi \Longrightarrow \exists x.\ (x, x) \notin r^{-1} \circ r$
3. $\psi \Longrightarrow r^{-1} \circ r \subseteq \mathit{Id}$
4. $\neg\psi \Longrightarrow \exists x\, y.\ (x, y) \in r^{-1} \circ r \wedge x \neq y$

Hint: before you try to phrase the conditions in Isabelle, they should be clear to you intuitively. You will find the constants to express the conditions in `Set.thy` and `Relation.thy` (and they are short!). ■

Well-founded Orders In the following exercises, the crucial step will be to use `wf_induct` with a suitable instantiation. Much of the rest can be automated (e.g. using `Blast_tac`).

Exercise 4
Show `wf_not_sym2`: $\mathit{wf}\ r \Longrightarrow \forall a x.(a, x) \in r \longrightarrow (x, a) \notin r$ in Isabelle. ■

Exercise 5
Show `wf_minimal`: $\mathit{wf}\ r \Longrightarrow \exists x.\forall y.(y, x) \notin r^{+}$ in Isabelle. ■

Exercise 6
Show `wf_subrel`: $\mathit{wf}\ p \Longrightarrow \forall r.r \subseteq p \longrightarrow (\exists x.\forall y.(y, x) \notin r^{+})$ in Isabelle. ■

Exercise 7
Show `wf_eq_minimal2`: $\mathit{wf}\ p = (\forall r.r \neq \{\} \wedge r \subseteq p \longrightarrow (\exists x \in \mathit{Domain}\ r.\forall y.(y, x) \notin r))$

Hint: You should use `wf_eq_minimal` and `Domain_def`. The theorem `wf_eq_minimal` provides a characterization of well-foundedness in terms of, intuitively speaking, sets of points. The theorem `wf_eq_minimal2` reasons in terms of sets of arrows, that is, relations. The proof of `wf_eq_minimal2` can be automated until one arrives at two subgoals, one having a statement about all point sets Q in the premises, and one having a statement about all relations r in the premises.

At this point, you must use `eres_inst_tac` with `allE` and instantiate the point set so that it is somehow related to a particular relation, and conversely, instantiate the relation so that it is somehow related to a particular point set. Afterwards, the rest can be done automatically. ■

Exercise 8
Using paper & pencil, show that $(4, 24) \in \mathit{wfrec\_rel}\ \text{`}{<}\text{'}\ \mathit{Fac}$, using the inductive definition

$$\forall z.(z, x) \in R \longrightarrow (z, g\, z) \in \mathit{wfrec\_rel}\, R\, F \Longrightarrow (x, F\, g\, x) \in \mathit{wfrec\_rel}\, R\, F,$$

given in lecture "Well-Founded Recursion". Use mathematical facts such as "there exists no z smaller than 0" freely.

Argue that one cannot show that $(4, 23) \in \mathit{wfrec\_rel}\ \text{`}{<}\text{'}\ \mathit{Fac}$. ■

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

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

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 to prove your goal. 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. ■

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 13 (2nd February 2007)

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

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

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:

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

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.ML` if you want to understand why).

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

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

Hint: if you find it rather silly to type $Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,(Suc\,0))))))))))))))))))))))))$, you may also type 24.
Now prove the goal

$$[\![c = \phi;\ g = \gamma]\!] \Longrightarrow nat\_rec\,c\,g\,11 = ?x$$

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 $sum : i \Rightarrow i \Rightarrow (i \Rightarrow i) \Rightarrow i$, 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 $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$, where $sumup\, f\, n$ is to stand for $\sum_{i=1}^{n} f\, i$.

Exercise 3
Define an expresion ς for the summation operator *sumup* using *nat_rec*, and test it by proving the following goals:

1. $sumup = \varsigma \Longrightarrow sumup\,(\lambda i.i)\,4 = 10$
2. $sumup = \varsigma \Longrightarrow sumup\,(\lambda i.i + 2)\,3 = 12$
3. $sumup = \varsigma \Longrightarrow sumup\,(\lambda i.c)\,m = c * m$
4. $sumup = \varsigma \Longrightarrow sumup\,(\lambda i.2 * i)\,m = m * (m + 1)$

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: $\forall n. \sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$. ■

The approximation theorem of lecture "Least Fixpoints" In the lecture we have seen the theorem

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

We now prove this theorem. You will have to use *iterate* from Exercise 1.

Exercise 4
Prove the following goals in Isabelle

1. $mono\ f \Longrightarrow (\bigcup_{n \in UNIV}(iterate\ n\ f\ \{\})) \subseteq \mathit{lfp}\ f$ (`union_below_lfp`)

   Hint: In the beginning, `auto`(); will do a good job. You then have to massage your current goal to reach a subgoal where you have to prove $\bigwedge n.iterate\ n\ f\ \{\} \subseteq \mathit{lfp}\ f$. This proof goes by induction (`induct_tac`). Afterwards you will have to use `lfp_unfold` and an elementary lemma about monotonicity.

2. $(\forall S.\ f(\bigcup S) = \bigcup(f \text{ ` } S)) \Longrightarrow mono\ f$ (`distr_implies_mono`)

   Hint: The following lemmas are crucial: `Un_eq_Union`, `Un_upper1`, `Un_absorb1`, `image_Collect`. You can use those by adding them to the simplifier set.

3. Write (on paper) a function $f :: nat\ set \Rightarrow nat\ set$ that is monotone but for which $(\forall S.\ f(\bigcup S) = \bigcup(f \text{ ` } S))$ does not hold.

4. $[\![mono\ f;\ \exists m.f\ (iterate\ m\ f\ \{\}) = (iterate\ m\ f\ \{\})]\!] \Longrightarrow \mathit{lfp}\ f \subseteq \bigcup_{n \in UNIV}(iterate\ n\ f\ \{\}))$ (`lfp_below_union_if_finite`)

   Hint: In the beginning, `auto`(); will do a good job. This will reach a goal where you have to prove an existential quantification. Make a smart guess how you should instantiate the existentially quantified variable!

   Note: this is interesting but will not be needed below.

5. $(\forall S.\ f(\bigcup S) = \bigcup(f \text{ ` } S)) \Longrightarrow f\ (\bigcup_{n \in UNIV}(iterate\ n\ f\ \{\})) = (\bigcup_{n \in UNIV}(iterate\ n\ f\ \{\}))$ (`union_is_fixpoint`).

   Hint: I based the proof of this lemma on several auxiliary lemmas, but there may be other ways of doing it (if you find a simpler way, great!):

(a) $\bigcup_{n \in UNIV} iterate\ (Suc\ n)\ f\ \{\} = \bigcup_{n \in \{m \mid 0<m\}} (iterate\ n\ f\ \{\})$ (`shift_successor`)
Hint: If you start with `auto();`, you will obtain a goal where you have to prove an existential formula. Use your intuition about natural numbers to guess the appropriate instantiation for the existentially quantified variable. Moreover, `iterate_Suc` will be useful.

(b) $\bigcup_{n \in UNIV}\ g\ (n :: nat)) = (\bigcup_{n \in \{m \mid 0<m\}} (g\ n)) \cup (g\ 0)$ (`split_universal_union`)
Hint: `case_tac` is useful here.

(c) $(\forall S.\ f(\bigcup S) = \bigcup(f\ `\ S)) \Longrightarrow$
$f\ (\bigcup_{n \in UNIV} (iterate\ n\ f\ \{\})) = (\bigcup_{n \in UNIV} f\ (iterate\ n\ f\ \{\}))$ (`f_special_distr`)
Hint: It is a good idea to start with `box_equals`, instantiating $?a$ and $?b$ to set unions using an expression $\exists n \ldots = iterate\ n\ f\ \{\}$. The `thm image_Collect` is useful again, but don't use it blindly! Work out exactly how you want to use it.

`union_is_fixpoint` can be shown using lemmas `f_special_distr`, `iterate_suc`, `shift_successor`, and `split_universal_union`.

6. $(\forall S.\ f(\bigcup S) = \bigcup(f\ `\ S)) \Longrightarrow lfp\ f \subseteq \bigcup_{n \in UNIV} (iterate\ n\ f\ \{\})$ (`lfp_below_union`)
Hint: This uses an important lemma in `Lfp.ML` and `union_is_fixpoint`.

7. $(\forall S.\ f(\bigcup S) = \bigcup(f\ `\ S)) \Longrightarrow \bigcup_{n \in UNIV} (iterate\ n\ f\ \{\})) = lfp\ f$ (`approx`)
Hint: Using some of the lemmas above, this should now be a very neat proof.

■

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.

First some organizational matters: The exercises on AVL trees are spread over three weeks. You will be devoloping and working with a theory file `AVL.thy` and a corresponding lemma file `AVL.ML`. So these files will *change* while you work on them.

For communicating the *versions* of those files in the coming weeks, we will proceed as follows: This week you will receive `AVL_fragment1.thy`. The version after completing this sheet should be named `AVL_completion1.thy`. In week 14 you will receive `AVL_fragment.ML`. You will work with this file as well as continue working with `AVL_completion1.thy`. The versions after completing sheet 14 should be named `AVL_completion2.thy` and `AVL_completion1.ML`. The versions after completing sheet 15 should be named `AVL_completion3.thy` and `AVL_completion2.ML`. This is illustrated in Figures 1 and 2. So again: work with `AVL.thy` and `AVL.ML` but only turn in files named as stated above.

The first part of this verification project is concerned with the basic definitions of the theory file. We have provided a fragment of the theory file `AVL.thy`. Your task will be to fill in more constants and definitions.

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". Similar to a type definition using `typedef`, 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. We will look into this in more detail next week.

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 guideline for the following exercises.

Exercise 5
Give a definition of a function *isin* (of result type *bool*) such that *isin* $a\, t$ holds iff a is a node

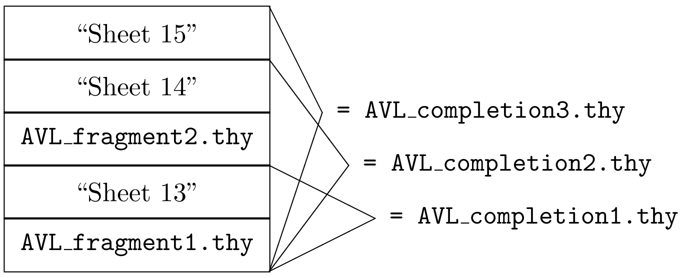


Figure 1: Theory files for AVL trees

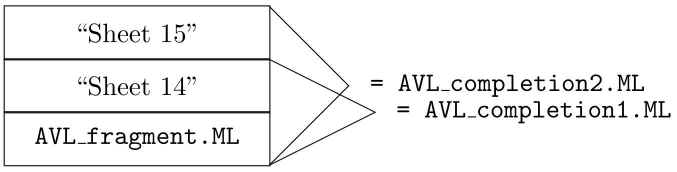


Figure 2: Lemma files for AVL trees

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 a few simple examples. ■

Exercise 6
Assume that the elements of the tree belong to a type in the class *order*. 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. ■

Exercise 7
From an algorithmical viewpoint, the definition of *isin* in Exercise 5 is inefficient. Can you suggest under which condition and how it could be made more efficient? ■

References

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

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

Computer Supported Modeling and Reasoning WS06/07
Exercise Sheet No. 14 (9th February 2007)

AVL trees This week we continue to look at AVL trees.

Exercise 1
We call a tree *balanced* if it is either a leaf or a node with balanced subtrees with a height that differs at most by one. Define a function *isbal* (of result type *bool*) such that *isbal t* holds iff t is balanced (as previously, you must also add the appropriate type declaration of *isbal* in the fragment). ∎

You should now 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.

As mentioned previously, 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 heavier on the right, or heavier on the left. To this end, you should insert the following lines into the `AVL.thy` file (for your convenience, all code fragments of this exercise sheet are contained in the file `AVL_fragment2.thy`):

```
datatype bal = Just | Left | Right

constdefs
 bal :: "'a tree => bal"
"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"
```

Exercise 2
What is the complexity of *bal* in n, where n is the number of nodes in the tree? If inserting 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, 2, 3, 4.

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

```
consts
 r_rot,l_rot,lr_rot,rl_rot :: "'a * 'a tree * 'a tree => 'a tree"
```

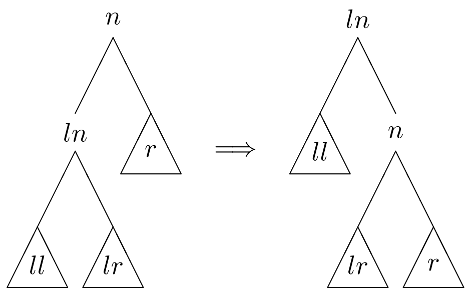


Figure 1: *r_rot*

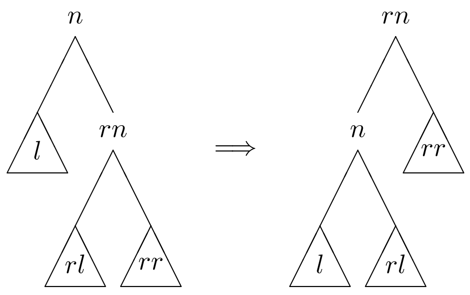


Figure 2: *l_rot*

```
recdef r_rot "{}"
"r_rot (n, Node ln ll lr, r) = Node ln ll (Node n lr r)"

recdef l_rot "{}"
"l_rot(n, l, Node rn rl rr) = Node rn (Node n l rl) rr"
```

The `recdef` syntax is provided in Isabelle for defining recursive functions. It is more general than the `primrec` syntax, since a well-founded ordering can be provided by the user. We use this syntax although the rotation functions are not recursive, since this syntax also allows for *pattern matching*, which is convenient. The `"{}"` indicates that we are not actually providing an ordering.

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., $r\_rot\ (n, \mathit{Leaf}, \mathit{Leaf})$ is undefined. This is no problem.

Exercise 3
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 $\mathit{Node}\ n\ l\ r$. 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 $\mathit{Node}\ n\ l'\ r$. It is ordered, but it might not be balanced.

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

So suppose that $\mathit{height}\, l' = \mathit{height}\, r + 2$. Then $\mathit{Node}\, n\, l'\, r$ looks as shown in the first picture

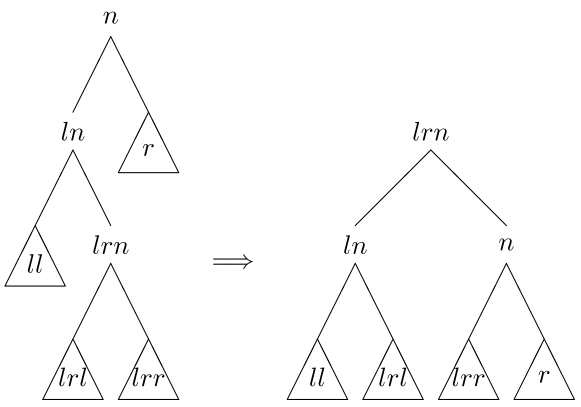


Figure 3: *lr_rot*

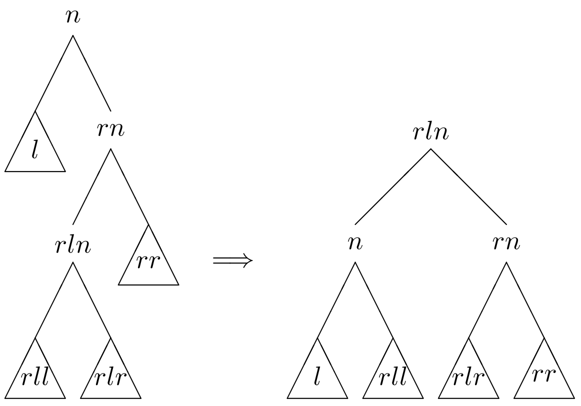


Figure 4: *rl_rot*

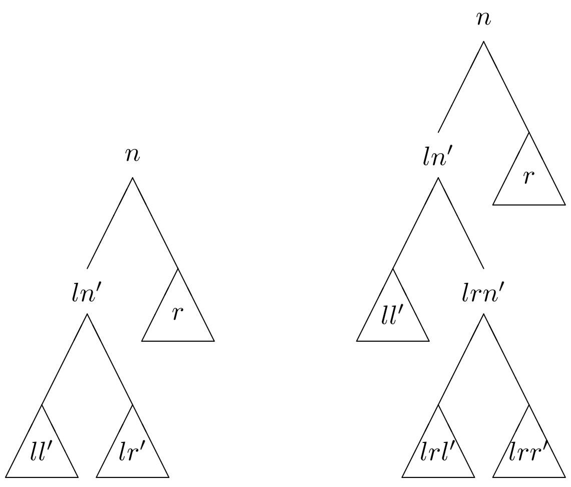


Figure 5: Too heavy on left

of Figure 5, where either *height* $ll' =$ *height* $r + 1$ or *height* $lr' =$ *height* $r + 1$.[2] The tree is too heavy on the left, and rotation must rectify this. We distinguish two cases:

bal $l' =$ *Right*. Since l' is balanced, this means that *height* $lr' =$ *height* $r + 1$ and *height* $ll' =$ *height* r. So, since lr' has height > 0, it follows that *Node* $n\, l'\, r$ actually looks as shown in the second picture of Figure 5, where both lrl' and lrr' have height *height* r or *height* $r - 1$. Since all three trees ll', lrl', lrr' have the same height as r or one less, it follows that *lr_rot* produces a balanced tree (see Figure 3).

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

Exercise 4
Define a function *l_bal* which performs an appropriate rotation for a tree that is too heavy on the left. Here is a fragment of the code you should insert into `AVL.thy`:

```
constdefs
 l_bal :: "'a => 'a tree => 'a tree => 'a tree"
"l_bal n l r == if
                then
                else                "
```

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

The insertion function is defined as follows:

```
consts
 insert :: "'a::order => 'a tree => 'a tree"
primrec
"insert x Leaf = Node x Leaf Leaf"
"insert x (Node n l r) =
   (if x=n
    then Node n l r
    else if x<n
         then let l' = insert x l
              in if height l' = Suc(Suc(height r))
                 then l_bal n l' r
                 else Node n l' r
         else let r' = insert 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`.

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 Leaf = 0"
"height (Node n l r) = Suc(max (height l) (height r))"
```

you can have

```
primrec
height_empty "height Leaf = 0"
height_branch "height (Node n l r) = Suc(max (height l) (height r))"
```

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

defining two names and for the according `thm`'s.

Exercise 5
Modify the `AVL.thy` file by introducing names for the equations as shown for `height` (choose the obvious names!). ■

"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_fragment.ML`, you find a number of essential lemmas about AVL trees. Several symmetric cases are left out. 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 command `qed_spec_mp` saves the theorem as $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \psi$ rather than $\phi_1 \longrightarrow \ldots \longrightarrow \phi_n \longrightarrow \psi$.

You may encounter problems with the provided ML-file since you might have defined the theory file in a different way from the model solution. If this is the case, see if you can repair `AVL.ML` appropriately. For the theorems you cannot repair, just comment them out for the time being, and use the theory file of the model solution next week.

Exercise 6
Copy `AVL_fragment.ML` into some scratch proof script. Complete it by inserting the symmetric cases, and save it as `AVL.ML`. ■

Exercise 7
For each of the lemmas already proven in `AVL_fragment.ML`, 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. ■

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

Computer Supported Modeling and Reasoning WS06/07 Exercise Sheet No. 15 (16th February 2007)

Efficient 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.

The first inefficiency we noted in Exercise 7 on Sheet 13 is that *isin* traverses the entire tree, which is unnecessary in case the tree is ordered.

Exercise 1
Write an efficient version of *isin* called *isin_eff* which works correctly on ordered trees (no correct behaviour is required on non-ordered trees). ■

Exercise 2
In some scratch proof script, prove the following goals:

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

After you have completed these proofs, paste them into `AVL.ML` (since they are fundamental lemmas about AVL trees). ■

Note that the inefficiency of *isin* is not a problem for insertion of nodes, but still *isin* is an important operation and so we should make it efficient. However for verification purposes we may also use the inefficient one.

Another inefficiency we noted was related to *bal*, which calls *height* and hence has running time $O(n)$ where n is the number of nodes in the tree. We suggested that this could be made more efficient by recording for each node what its balance is: *Just*, *Left*, or *Right*.

Exercise 3
Define a new datatype *etree* with term constructors (constants) *ELeaf* and *ENode* in analogy to *tree*, but with an additional label giving the balancing information for a node. Do this still in the same theory file `AVL.thy`. ■

In the following, we will call members of the new type *E-trees*.

Exercise 4
Define the function *prune* which, given an E-tree *et*, returns the tree obtained from *et* by throwing away all the balancing labels.

Open a proof script `ex15.ML` and test *prune* on four different E-trees. E.g., prove the goal

$$prune(ENode\ Just\ 2\ ELeaf\ ELeaf) = ?t\text{''}$$

Isabelle should instantiate $?t$ to $Node\ 2\ Leaf\ Leaf$. ■

The balancing labels are supposed to record, for each node in a tree, the information that would otherwise have to be computed by *bal*. Obviously, the balancing labels might be incorrect, i.e., deviate from what *bal* computes. E.g., *ENode Just* 2 *ELeaf ELeaf* is labeled correctly whereas *ENode Just* 2 *ELeaf* (*ENode Just* 3 *ELeaf ELeaf*) is not.

Exercise 5
Define a function *correct* which says if an E-tree is correct, and test it in `ex15.ML` on four trees, two correct ones and two incorrect ones. ■

Exercise 6
Prove the following goal in a scratch proof script:

$$\mathit{correct}\ (\mathit{ENode}\ b\ n\ l\ r) \longrightarrow (\mathit{bal}\ (\mathit{prune}\ (\mathit{ENode}\ b\ n\ l\ r)) = b).$$

Now paste the proof into `AVL.ML`. ■

Exercise 7
Define the function *label* which, given a tree t, returns the E-tree obtained from t by inserting correct balancing labels.

Test *label* in analogy to Exercise 4. In these tests, the final result should be completely evaluated. It is not acceptable if the instantiation of the metavariables still contains "unevaluated function calls". ■

Exercise 8
Prove the following two goals in a scratch proof script:

1. $\mathit{prune}(\mathit{label}\ t) = t$;
2. $\psi \Longrightarrow \mathit{label}(\mathit{prune}\ t) = t$,

where ψ is a natural condition that you have to guess. Now paste the proof into `AVL.ML`. ■

This completes our little verification project and this course. Of course, there are many more properties of AVL trees to be shown.