## Contents

1 Getting Started 2  
1.1 Overview ....................................................... 2  
1.2 Starting KIV ................................................... 2  
1.3 Logical Foundations .......................................... 2  

2 Predicate Logic Proofs in KIV 7  
2.1 Editing and Loading Theorems ................................. 7  
2.2 Syntax and Special Character Input ........................... 9  
2.3 Proofs in KIV .................................................. 11  
2.4 Rule set for predicate logic .................................. 14  
2.5 Basic rules ..................................................... 16  
2.6 Heuristics for basic rules ..................................... 17  
2.7 Exercise 1 ....................................................... 19
Chapter 1

Getting Started

1.1 Overview

This course deals with the development of provably correct programs and software systems. The KIV system is used as a tool and development environment throughout the course. The first exercise deals with propositional and predicate logic, proofs and their representation in KIV. The topic of the second exercise are proofs over data structures: theorems over specifications of lists and heaps have to be proved. In the third exercise you will create a structured specification yourself. The fourth and fifth exercises deal with program verification “in the small” and “in the large”.

The exercises are solved with the KIV system. As a preparation, documentation with introductory material, theoretical background, and the exercises is issued one week in advance. It is indispensable to read the documentation in advance and to prepare a solution. Otherwise it will not be possible to solve the exercises with the system during the scheduled time. However, the computers are also available at other times, if the normal work is not hindered.

1.2 Starting KIV

Installation

Directories for every group have been set up. Instructions how to do the initial installation will be given in the first course.

Starting KIV

Just type

    ./startkiv.sh

in a shell in your home directory.

1.3 Logical Foundations

The logic underlying the KIV system combines Higher-Order Logic and Dynamic Logic. Higher-Order Logic extends first-order logic with functions that have functions as arguments and results. Also function variables and lambda expressions \( \lambda x.e \) that denote anonymous functions are allowed. Dynamic-Logic is an extension of first-order logic by modal program formulae. Dynamic Logic (DL) allows the expression of properties of programs like partial and total correctness, program equivalence etc. By a suitable axiomatisation, proofs that show these properties become possible. The following chapter defines the syntax and the semantics of the first-order part of DL formally.
Finally a sequent calculus for DL is defined. For a survey on DL see [Har84], [SA91] resp. [And86] are good introduction to first-order logic and to sequent calculus resp. to higher-order logic. In this chapter we introduce the first-order fragment of the logic.

Syntax
Signatures
A signature $\Sigma = (S, OP, X)$ consists of a finite set $S$ of sorts, a finite family $OP = \bigcup_{s,s' \in S} OP_{s,s'}$ of operations (with a list of argument sorts $s$ and target sort $s'$) and a family $X = \bigcup_{s \in S} X_s$ of countably infinite sets of variables.

We always assume that the sorts $S$ contain at least bool and nat, and the operations contain the usual operations on bool ($true, false, \land, \lor, \rightarrow, \leftrightarrow, \neg$) and nat ($0, succ, pred, +$).

First-order expressions
For a given signature $\Sigma$, the set of expressions $EXPR := \bigcup_{s \in S} EXPR_s$, where $EXPR_s$ are defined to be the smallest sets with

- $X_s \subseteq EXPR_s$ for every $s \in S$
- If $f \in OP_{s,s'}$ and $t \in T_s$ then $f(t) \in EXPR_s$
- If $\varphi \in FMA$ and $\overline{x} \in \hat{X}_s$ then $\forall \overline{x}. \varphi \in FMA$.
  $\hat{X}_s$ is the set of duplicate free lists of variables of sorts $s$.
- If $\varphi \in FMA$ and $\overline{x} \in \hat{X}_s$ then $\exists \overline{x}. \varphi \in FMA$
- If $t, t' \in T_s$, then $t = t' \in FMA$
- If $\varphi \in FMA$ and $t, t' \in EXPR_s$, then $(\varphi \sqsupset t; t') \in EXPR_s$

In the definition, FMA (formulas) abbreviates $EXPR_{bool}$, the set $T_s$ (terms of sort $s$) is the subset of $EXPR_s$, that contains neither quantifiers nor programs. BXP (boolean expressions) is $T_{bool}$. We also write $EXPR(\Sigma)$ and $FMA(\Sigma)$ for $EXPR$ and $FMA$, when we want to emphasize that the the sets depend on $\Sigma$.

Semantics
Algebra
Given a signature $\Sigma$, an algebra $A$ consists of a nonempty set $A_s$ for every sort $s$ and an operation $f_A : A_s \rightarrow A_{s'}$ for every $f \in OP_{s,s'}$. We assume $A_{bool} = \{tt, ff\}$, $A_{nat} = \mathbb{N}$ and that operations on booleans and naturals have their usual semantics.

States
Given a signature and an algebra for that signature, a state $z \in ST_A$ is a mapping $z$, which maps variables of sort $s$ to values of $A_s$. The state $z[\overline{x} \leftarrow a]$ is the same as $z$, except that the variables in $\overline{x}$ are mapped to the values $a$. States are sometimes also called valuations.

Semantics of Expressions
Given an Algebra $A$ and a state $z$ the semantics $[e]_z \in A_s$ of a DL expression $e \in EXPR_s$ is defined as:

- $[x]_z = z(x)$
- $[f(t)]_z = f_A([t]_z)$ for $f \in OP_s$ and $t \in T_s$
• \([\forall x.\varphi]_z = \text{tt} \text{ with } x \in X \text{ iff } [\varphi]_{z[x \leftarrow a]} = \text{tt} \text{ for all values } a \in A\)

• \([\exists x.\varphi]_z = \text{tt} \text{ with } x \in X \text{ iff } [\varphi]_{z[x \leftarrow a]} = \text{tt} \text{ for some value } a \in A\)

• \([\{\varphi \supset e; e'\}]_z = [e]_z \text{, if } [\varphi]_z = \text{tt}, \text{ and } [e']_z \text{ otherwise.}\)

Models and Validity

A formula \(\varphi\) holds over a \(\Sigma\)-Algebra and a valuation \(z\), short \(A, z \models \varphi\), if and only if \([\varphi]_z = \text{tt}\). A \(\Sigma\)-Algebra is a model of a formula \(\varphi\), short \(A \models \varphi\), iff for all states \(z\): \(A, z \models \varphi\). A formula \(\varphi\) is valid, if all \(\Sigma\)-algebras are a model of \(\varphi\). A formula \(\varphi\) follows from a set of formulas \(\Phi\), short: \(\Phi \models \varphi\), if any model of all formulas from \(\Phi\) is also a model of \(\varphi\).

Sequent calculus

To define the basic axioms for KIV supports a sequent calculus, since formal reasoning with its rules resembles informal proofs by hand quite closely. For an introduction to 'natural deduction' with sequent calculi see [Ric78] and [SA91].

In the following we will define informally the important concepts of sequent calculus. This should suffice to work with the rules.

Sequents

Let \(\varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m \in \text{FMA}\) be two lists of formulas with \(n, m \geq 0\). Then the scheme

\[
\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m
\]

is called a sequent. \(\varphi_1, \ldots, \varphi_n\) is called the antecedent, \(\psi_1, \ldots, \psi_m\) the succedent of the sequent. A sequent is simply a way to present the formula

\[
\varphi_1 \land \ldots \land \varphi_n \rightarrow \psi_1 \lor \ldots \lor \psi_m
\]

The meaning of sequent therefore is: The conjunction of the antecedent formulas implies the disjunction of the succedent formulas. Note that to determine, if a sequent has a model, it’s free variables have to be treated, as if the whole sequent were universally quantified.

The empty conjunction is defined to be true, the empty disjunction is defined to be false. A sequent with empty succedent therefore is true, if and only if the antecedent contains a contradiction. In the following rules, instead of using concrete formulas we will use formula schemes (e. g. \(\varphi, \psi\)). Such placeholders are called meta variables, in contrast to elements from \(X\), which are called object variables. We will use meta variables \(\varphi, \psi\) for formulas. We also use the capital greek letters \(\Gamma, \Delta\) as meta variables for lists of formulas. As an example \(\Gamma, \varphi \vdash \Delta\) is a scheme for a sequent. Any sequent with nonempty antecedent would be an instance of the scheme.

Rules of sequent calculus

If \(S_1, \ldots, S_n, S (n \geq 0)\) are sequents (possibly containing meta variables), then the scheme

\[
\begin{array}{c}
S_1 & S_2 & \ldots & S_n \\
\hline
S \\quad C
\end{array}
\]

is called a sequent rule. \(S\) is called the conclusion, \(S_1, \ldots, S_n\) die premises of the rule. sequent rules, which have an empty set of premises \((n = 0)\) are called axioms. \(C\) is a side condition to form instances of the meta variables occurring of the rule (usually it restricts the formulas that instantiate the meta variables not to contain certain object variables). Often this condition is true, and we drop it in this case.

The semantics of a rule is: the conclusion follows from the premises. It views all the sequents as they were formulas. Sequent calculus consists of a finite set of such rules.
Derivations

Rules of sequent calculus can be combined to derivations. A derivation is a tree, whose nodes are sequents. All elementary subtrees of such a derivation must be instances of sequent rules. Exactly as for rules the root of the tree is called ‘conclusion’, the leaves are called ‘premises’. A more formal definition of derivation is:

- A sequent is a derivation which has itself as root and as only premise.
- A tree with conclusion \( K \) and subtrees \( T_1 \ldots T_n \) (\( n \geq 0 \)) is a derivation, if all subtrees \( T_1 \ldots T_n \) are derivations, and there is a sequent rule \( R \) with conclusion \( K' \), condition \( C \) and premises \( P_1 \ldots P_n \), such that there is a substitution \( s \) (for the meta variables of \( R \)) such that \( s(K') = K \), the condition \( s(B) \) is true and \( s(P_1) \ldots s(P_n) \) are the conclusions of \( T_1 \ldots T_n \).

The semantics of a derivation in sequent calculus is again: The conclusion follows from the premises. A derivation with an empty set of premises is a proof, that its conclusion is valid. The basic rules of sequent calculus for the first-order part of DL are summarized in Sect. 2.5.

In practical work with sequent calculus one usually starts with a goal \( S \) (written as a sequent) which should be proved. Then rules are applied backwards, reducing the goal to simpler subgoals. Note that for most of the rules defined in Sect. 2.5, the premises are simpler than the conclusion in the sense that the maximal number of symbols in the formulas of the sequent decreases. This process stops when the axiom axiom of sequent calculus is reached.

Theories

Finding a proof for a sequent \( K \) over some signature \( \Sigma \) using the basic rules establishes, that the sequent is valid in all models of \( \Sigma \). This is called logical reasoning.

But usually we want to have some meaning to be associated with the symbols. To do this we will use a (finite) sets of axioms \( \text{Ax} \subseteq L(\Sigma, X) \) as preconditions. Theory reasoning allows proofs with open premises that are axioms. A pair \( (\Sigma, \text{Ax}) \) is called theory or a (basic) specification. The existence of a theory is the usual case in program verification. Usually the theory describes data types used in a program like naturals, lists or records. In a later chapter theories will also be used to describe program systems.

Term generatedness and induction

To describe datastructures adequately, apart from first-order axioms we also need axioms for (structural) induction. Unfortunately an induction scheme is an infinite set of axioms, that cannot be replaced by a finite set of formulas. As a replacement we use generation clauses as axioms. Such a clause has the form

\[ s \text{ generated by } f_1, f_2, \ldots, f_n \]

(‘generated by’-clauses are also possible with several sorts, but since they are rare we will not discuss them.) In the clause \( s \) is a sort, and \( f_1, f_2, \ldots, f_n \) are operation symbols with target sort \( s \). This axiom holds in an algebra \( A \) of the signature, if all elements \( a \in A_s \) can be represented by a term \( t \), which consists of operation symbols from \( f_1, f_2, \ldots, f_n \) only and contains no variables of sort \( s \). A term ‘represents’ an element \( a \), if for a suitable state \( z \): \( z(t) = a \). In other words we can say that the carrier set \( \in A_s \) can be generated by looking at the values of terms as described above.

If the argument sort of the function symbols \( f_1, f_2, \ldots, f_n \) contain no other sort than \( s \), then \( s \) is often called a sort generated by (the constructors) \( f_1, f_2, \ldots, f_n \). In this case every element of \( A_s \) can be represented as the value of a variable free term \( t \).

The simplest example for a generated sort are natural numbers, which are generated by \( 0 : \to \text{nat} \) and the successor function \( +1 : \text{nat} \to \text{nat} \). If we write \( +1 \) postfix (i.e. \( x +1 \) instead of \( +1(x) \)) this means that every natural number is of the form

\[ 0 +1 +1 +1 \ldots +1 \]
CHAPTER 1. GETTING STARTED

That is, we have the axiom

\[ \text{nat generated by } 0, +1 \]

for natural numbers. In this special case two different terms always denote two different numbers (such a datastructure is \textit{freely generated}). This is not always the case: integers can be generated by \( 0 : \text{int} \), the successor function and \( +1 : \text{int} \to \text{int} \) and the predecessor function \( -1 : \text{int} \to \text{int} \).

We have

\[ \text{int generated by } 0, +1, -1 \]

But here the two different terms ‘0’ and ‘0 +1 –1’ represent the same number. Typical examples for data structures in which the function symbols \( f_1, f_2, \ldots f_n \) have other argument sorts than \( s \), are parameterized data structures like lists, arrays or (finite) sets. For the latter

\[ \text{set generated by } \emptyset, \text{insert} \]

holds, if \( \emptyset : \to \text{set} \) is the empty set and function \( \text{insert} : \text{elem} \times \text{set} \to \text{set} \) adds an element to a set. The elements of sort \( \text{elem} \) are the parameter which can be chosen freely.

Generation clauses correspond to induction schemes, which allow induction over the structure of a term. The argument is as follows: To show a goal \( \varphi(x) \) for all elements \( x \) of some sort \( s \), it suffices to show that for every \( f_i \) \( \varphi(f_i(x_1, \ldots x_n)) \) holds, assuming that \( \varphi(x_i) \) holds for all \( x_i \) which have sort \( s \). In those cases, where \( f_i \) has no arguments of sort \( s \) (e.g. if \( f_i \) is a constant), \( \varphi(f_i(x_1, \ldots x_n)) \) must be proven without any preconditions. We will not give a formal definition of the induction rule for arbitrary generated by clauses. Instead we only show the two examples for natural numbers and sets, which should make the principle clear:

\[
\begin{align*}
\vdash \varphi(0) & \quad \varphi(n) \vdash \varphi(n + 1) \\
& \quad \vdash \forall n. \varphi(n)
\end{align*}
\]

\[
\begin{align*}
\vdash \varphi(\emptyset) & \quad \varphi(s) \vdash \varphi(\text{insert}(e, s)) \\
& \quad \vdash \forall s. \varphi(s)
\end{align*}
\]
Chapter 2

Predicate Logic Proofs in KIV

2.1 Editing and Loading Theorems

As far as it is necessary for the first exercises we will shortly describe the structure of the software development environment of the KIV system.

2.1.1 Directory structure

The KIV system is a system for the specification and verification of software systems. Every single software system is handled in a project. After the start of the system (the software development environment) you have the possibility to select an existing project (or install a new one) on the project selection level. In the practical course you have to select for every exercise $k$ the corresponding project Exercise($k$). All data of a project is located in a unix directory with the same name. Therefore the project for the first exercise is located in

(Your project directory)/Exercise1

By selecting a project you reach the project level. If you want back to the project selection level you have to click on Edit – Go Back (Go Back is a sub menu of the menu Edit of the graph visualization tool uDraw). Every project consists of specifications and modules. Their dependency forms a directed acyclic graph. This graph is visualized with the graph visualization system uDraw. Rectangles correspond to specifications, rhombs to modules. Specifications are described in detail in later chapters. For the moment it is enough to know that every specification and every module contains a logical theory (i.e. a signature and axioms, e.g. the theory of natural numbers) where you can prove theorems. Every specification itself is located in its own subdirectory of the project. The name of the subdirectory is specs/<name>. So you find the specification “proplcog” in the directory

(Your project directory)/Exercise1/specs/proplcog

The specification text is in the file specification in this directory. You can click on a rectangle (or rhomb) with the left mouse button to view or edit the specification. To work with a specification you select Work on ....

After a short time you will get another window that contains a detailed of the specification’s theorem base. The theorem base contains the axioms of the specification, and user defined theorems (properties, lemmas), and their proofs. The specification window starts with the Summary register tab that shows the names of the axioms and theorems sorted by their state (axiom, proved, unproved, partial, invalid, sig invalid). You can select an entry by clicking on it with the left mouse button (this will highlight the entry). Then you can click the right mouse button to get a popup menu where you can View the theorem, begin a New Proof, etc. When you select the register tab Theorem Base you get another view of the axioms and theorems that also displays
their sequents (or at least part of them). Here you don’t have to select a theorem first with the left mouse button; instead you can just click on a theorem with the right mouse button to get the popup menu. Both tabs have the same functionality – they just present different views of the theorem base.

The most important menu entry is File – Save that saves the theorem base back to the hard disk (KIV has no autosave feature). We recommend to save the theorem base regularly. If you leave the specification with File – Close the theorem base will be saved automatically if necessary. Note that the uDraw window the development graph is still active while the specification window is open, i.e. you can click in the graph to view other specifications etc.

2.1.2 Editing Theorems

You can enter new theorems (sequents, not only formulas) either directly with the menu entry Theorems – Enter New, or through a file. Because theorems are often quite large and it is difficult to enter them without a fault you can use your favourite editor to edit a file. For new theorems the file sequents in a specification directory is used. In our example you have to edit

⟨Your project directory⟩/Exercise1/specs/prologic/sequents

The command Edit – Theorems starts the xemacs editor with that file loaded, and is a comfortable shortcut. After editing and saving the file, KIV has to be informed that it should load new or changed theorem. This is done with the commands Theorems – Load New and Theorems – Load Changed. Both of these commands can also be accessed using the corresponding buttons in the Theorem Base tab.

The syntax of the file is as follows:

1. Two or more semicolons (;;) begin a one line comment (as // in Java)

2. (: begins and :) ends a multi line comment (as /* and */ in Java). In contrast to Java these comments can be nested.

3. Parsing stops after a line beginning with ;; END

4. The file may not contain double quotes anywhere.

5. The file begins with an optional keyword variables that allows to declare auxiliary variables for theorems.

6. The keyword lemmas begins the list of theorems.

7. A theorem has the form

⟨theorem name⟩ : ⟨sequent⟩ ;

The trailing semicolon is mandatory. Instead of a sequent you can also just write a formula $\varphi$, which is read as $\vdash \varphi$.

Every theorem can be written in this file. When an already defined theorem is loaded a second time, this definition is ignored. A theorem can be modified with the menu entry Theorems – Load Changed or the Load Changed button in the Theorem Base tab likewise in this file. Other theorems which are not modified are ignored.

To prove a theorem you select either

• Begin to prove an unproved theorem,

• Continue to continue a partial proof,

• Load to load a complete proof, or

• Reprove to start a new proof for a completely or partially proved theorem
in the **Proof** menu. You can also use the popup menu by right-clicking on a theorem and select **New Proof** (for **Begin** or **Reprove**), **Continue Proof**, or **Load Proof**.

The selection of any of these commands opens another window where proofs are done. As before, all other open windows are still active. However, some commands are not possible when you have a current proof. E.g. **Theorems** – **Delete** will issue the message ‘You can’t use this command if you have a current proof.’ You finish a proof by closing the proof window with **File** – **Close**. If you want to keep the proof you should save the theorem base first with **File** – **Save**. (This also saves the current proof.) If you want to discard the proof (because you don’t want to overwrite the old proof for some reason – KIV normally doesn’t keep different versions of proofs) you simply close the window and answer the following question ‘The proof is modified. Update the theorem first?’ with ‘No’. If you answer ‘Yes’ the current proof will be stored in the theorem base and saved the next time the theorem base is saved. (If you change your mind you can also cancel the action.)

### 2.2 Syntax and Special Character Input

The syntax of predicate logic in KIV is essentially the same used e.g. in chapter [1.3](#). The following operator precedences hold:

\[
\{\neg\} \succ \{\land\} \succ \{\lor\} \succ \{\rightarrow\} \succ \{\leftrightarrow\} \succ \{\forall, \exists\}
\]

Quantifiers can contain one or more variables. If more than one variable exists they are separated by commas and after the last variable (and before the quantified formula) a dot follows. Theorems are sequents and not only formulas. The sequent sign is \(\vdash\).

Many of the logical symbols use a special character, and KIV offers even more like \(\cup\) or \(\leq\), or greek letters. A full list is at the end of this section. To insert these characters in the xemacs one has to press F12 and enter the name of the special character (followed by return). The most important names are

<table>
<thead>
<tr>
<th>(\neg)</th>
<th>(\land)</th>
<th>(\lor)</th>
<th>(\rightarrow)</th>
<th>(\leftrightarrow)</th>
<th>(\forall)</th>
<th>(\exists)</th>
<th>(\vdash)</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>and</td>
<td>or</td>
<td>implies</td>
<td>equiv</td>
<td>all</td>
<td>ex</td>
<td>follows</td>
</tr>
</tbody>
</table>

It is sufficient to type in a unique prefix, e.g. instead of typing `implies` it is enough to enter `im` and hit return. The available completions for a prefix are shown by hitting the tab key.

Similar support for special characters as in xemacs is also available in all editable text fields of dialog windows. Pressing F12 in such a field will open a dialog, where all the symbols supported by KIV are shown. You can insert a symbol by either double clicking it or entering a unique prefix and hitting ENTER. After that the dialog is closed automatically. You can cancel the dialog by pressing ESC or clicking the close button.

Note also, that you can always paste an arbitrary text selection with your mouse, or paste from your clipboard. Finally, since KIV originally had an ascii-syntax, the most important special characters still have the following ascii-syntax (but it is not recommended that you use it).

<table>
<thead>
<tr>
<th>(\neg)</th>
<th>(\land)</th>
<th>(\lor)</th>
<th>(\rightarrow)</th>
<th>(\leftrightarrow)</th>
<th>(\forall)</th>
<th>(\exists)</th>
<th>(\vdash)</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>and</td>
<td>or</td>
<td>-&gt;</td>
<td>&lt;-&gt;</td>
<td>all</td>
<td>ex</td>
<td></td>
</tr>
</tbody>
</table>

When entering terms and predicates and formulas some common pitfalls should be avoided:

1. In KIV nearly arbitrary identifiers are allowed for symbols, e.g. 0, +, ++, <, list<, |, \(\rightarrow\)3, x=y, 3_7\textbackslash etc. The disallowed characters are brackets, space, comma, semicolon, backquote, and double quotation mark. Colon, star, and the dollar sign are treated specially. Therefore:

   **There must be spaces around identifiers unless the previous or next character is a bracket, comma, or semicolon!**
"x=y" is not the equation “x = y”, but a three character identifier!

2. Variables, function and predicate symbols can only be entered when they are already defined in the signature of the specification.

In the specification `barbier` the variables x and y of sort person are defined. Only these two variables can be used to formulate the propositions.

3. Functions and predicates can be written in the usual form \( f(t_1, \ldots, t_n) \), but there are also infix, prefix, and postfix operators. E.g. + is an infix operator so you must write “m + n” instead of “+(m,n)”. +1 is a postfix operator and is placed behind its argument: “(m + n) +1”. A prefix operator precedes its single argument without brackets. An infix, prefix, or postfix operator can’t be written in another manner, i.e. it is not possible to write “+(m, n)” or “+1(n)”.

Postfix operators have a higher precedence than prefix operators. Infix operators have lowest precedence, so the expression “m + n +1” is equivalent to “m + (n +1)” and not to “(m + n) +1”! Furthermore, for infix operators a precedence can be specified, e.g. \( * \) is defined to have higher precedence than +. If you are not sure about precedences you should place brackets around the expression.

Infix operators are specified in the signature definition of the specification with two dots around the symbol. E.g. in the specification `nat`

\[
. + . : nat \times nat \rightarrow nat;
\]

Postfix operations have got a dot before the symbol in the definition (the dots mark the positions of the parameters):

\[
. +1 : nat \rightarrow nat;
\]

4. The dot is allowed at the beginning of a symbol, but nowhere else. It is often used to define postfix operations. The leading space before a symbol starting with a dot can be dropped. As an example in “x.first” the postfix operation “.first” is applied on x, and the expression needs no space in the middle. A single dot is not a symbol, but used only in quantifiers. There, no space is needed before the dot, but one space is needed after it to separate the next symbol.

5. Special characters in Xemacs normally have the same name as the corresponding \LaTeX symbol, so \( \odot \) is odot, \( \oplus \) is called oplus and so on. In ascii syntax these symbols can be entered as the xemacs name with a leading backslash, e.g. \texttt{\textbackslash odot} for \( \odot \), \texttt{\textbackslash oplus} for \( \oplus \), etc.

A parser error occurs if the input is not correct. In the specification `nat` e.g. the variable x is not defined. So the input “n + x” results in the parser error

Parser: parse error in "n + ?? x".

The parser displays two question marks before the next token that can’t be parsed. The text up to this point was parsed successfully (though perhaps not in the manner you expected). The input “n + = n” results in

Parser: parse error in "n + ?? = n".

If the beginning of the input is already faulty (e.g. the input “x” in the specification `nat`) you get

Parser: parse error at the beginning of "x".

If something is missing at the end (e.g. the semicolon after a sequent in the sequents file, see \texttt{2.1.2}) you get

Parser: parse error at the end of "lemmas lem-01 : \vdash 0 + m = m".
2.3 Proofs in KIV

The proof window (the window has the title "KIV - Specification Strategy") is used for proofs in the sequent calculus. In the large frame on the right you always find the currently selected premise of the proof tree, the actual “goal” you are just now working on. On the left side all applicable rules are shown. Another window with the title “current proof” shows the current proof tree.

Selecting basic rules

In the first exercise the basic rule set of the sequent calculus is used instead of the optimized rule set that is normally (and in later exercises) used. For your convenience this switching of rule sets has already been done (using a configuration file). Switching rule sets can also be done manually (and we will need this feature later on) by selecting the menu entry Control – Options. After that a window will appear – with a rather longish list of options. For the first exercise only the option “Use basic rules” is of interest, which is already selected. Clicking on this option will deselect it or select it again. Clicking the “Ok” button will activate your selection of options. For the first exercise the “Use basic rules” option should be activated.

Short description of the menu commands

In the following section all menu commands of the proof level which are relevant for first order proofs are described. Some important commands are also present in the icon bar.
• Menu File
  – **Save**: save the theorem base and the current proof (Icon: diskette)
  – **Close**: end the current proof (Icon: closing door)

• Menu View
  – **Specification**: view the used specification (the specification can also be viewed under the specification tab)

• Menu Proof
  – **Begin/Continue**: choose a new theorem to prove

• Menu Goal
  – **Open Goals**: display informations about open premises
  – **Switch Goal**: continue working at an other goal (Icon: Green Arrow left/right to switch to the previous/next goal).
  – **Show Tree**: display the current proof tree. This command always is very useful to get informations about the current proof. (Icon: Green Three-Node Tree).
  – **Prune Tree**: prune a branch in the current proof tree. This makes sense if there is an error in the proof. The position where the tree should be pruned is chosen by clicking on the node and pressing p (or choosing **Operations** – **Prune Tree** in the tree window). If the current proof tree is displayed it’s also possible to click on the node and use p or **Operations** – **Prune Tree** (without using a menu command). Note that this command doesn’t change the chosen heuristics. It is therefore possible that the new goal is treated by the heuristics at once. In this case all heuristics should be deactivated before pruning the tree by clicking “Use Heuristics” in the bottom left corner of the proof window.

• Menu Control
  – **Heuristics**: choose heuristics (see below; Icon: Terminal with On/Off).
  – **Backtrack**: got back to the last backtrack point. The system generates always a backtrack point if rules are applied or the systems behavior changes because of an interaction. Nevertheless only a certain number of backtrack points are stored. (Icon: yellow, curved arrow)
  – **Options**: To select the option **Use basic rules**. A window appears with a rather longish list of options. For the first exercise only the option “Use basic rules” is of interest. Clicking on this option will mark it as selected. By clicking the “Ok” button the selection of options is ended.
    Options are valid as long as you are working on a specification or doing proofs in this specification, i.e. the option remains in charge until you go back to the project level.

**Viewing proof trees**

Proof trees can be displayed graphically in the tree windows. A brief description how such a tree can be interpreted follows.

**Colors**

A proof tree appears in different colors which serves for an easier orientation.

• Color of branches: closed branches are displayed in green (i.e. there exist no more open premises resp. all open premises are axioms or theorems). Red branches lead to open premises which have to be proven.
CHAPTER 2. PREDICATE LOGIC PROOFS IN KIV

• Color of nodes:
  – Colors of inner nodes: The color of the inner nodes marks the kind of rule that was used.
    * black: propositional rules not introducing case distinctions, i.e. conjunction left, negation left, disjunction right, implication right, negation right, and other “unimportant” rules like simplifier, elimination etc.
    * red: induction rules
    * blue: cut formula, disjunction left, implication left, equivalence left, conjunction right, equivalence right, case distinction, program calls
    * brown: weakening
    * green: quantifier instantiation
    * violet: rules for loops and whiles
  – Color for leave nodes: The color of the leave nodes sign how the corresponding branch was closed (if at all).
    * green: axiom or theorem of the specification.
    * blue: axiom or theorem of a subspecification.
    * red: open premise.

• filled vs. non-filled nodes: Premises are always drawn as non-filled nodes. Inner nodes of a tree are non-filled if the applied rule was chosen by the user. If a rule was applied automatically by an heuristic (see below) the node is filled.

• enumeration of premises: The open premises of a tree are enumerated. If the proof uses some theorems as lemmas their names are displayed (lemmas are not treated as open premises).

Clicking a proof tree

Clicking on a node with the left mouse button marks the node and displays the corresponding sequent in a sequent window. Clicking with the middle button displays the sequent in a new sequent window without marking that node. Clicking a further node with the middle mouse button opens a second, new sequent window and so on (useful if you want to see more than one sequent). Clicking a node with the right button marks it without displaying its sequent. By left-(or right-)clicking a part of the tree window where no node is displayed the marking is reset.

Operations

The menu operations contains a number of commands. But only a few of them are interesting:

Prune Tree  After clicking on a node with the left or right mouse button this menu item causes that the command Prune Tree is executed (see above). The tree is pruned at the marked point.

The key p is an abbreviation of this menu item.

Switch Goal  After the choice of an open premise with the left mouse key the proof is continued at this premise.

The key g is a shortcut for this item.

Quit  The tree window (and all its sequent windows) are closed. (Shortcut: key q)

Note that these menu items are disabled when they’re not applicable. E.g. Switch Goal needs that both a node is selected in the tree and that this node is an open premise in order to be applicable and thus enabled.

---

1 The marked node will be shown in a red rectangle.

2 The window will be titled Goal <nr>. For reference, in the proof tree window <nr> will be written on the right of the node.
\section*{Sequent Windows}

The size of sequent windows is variable. The system tries to increase the size of the window until the whole output (generated by the pretty printer) fits. If that is not possible scrollbars are added to the window.

Above the sequent in some cases a text is displayed. This text is extracted from the tree-comment of the (sub-)tree whose conclusion is the printed sequent. It normally contains the name (and in some case also the arguments) of the used rule. In case of open premises nothing is displayed. To close the sequent window press key $q$ in the window (or click the \textit{Close} button).

\section*{2.4 Rule set for predicate logic}

The rule set for predicate logic can be found in the next section (2.5). Here we describe the usage of three selected rules.

\subsection*{2.4.1 all left}

The quantifier rules \textit{all left} and \textit{exists right} require (or allow) an instantiation that must be entered interactively. We describe the rest for \textit{all left}, but everything also works for \textit{exists right}.

\begin{align*}
\forall x.\varphi, \Gamma \vdash \Delta \quad (\text{all left})
\end{align*}

$\varphi^\tau_x$ is the substitution of $x$ by $\tau$ in $\varphi$, $\tau$ may be an arbitrary term. The rule is also applicable for a list of quantified variables $\forall x_1, \ldots, x_n.\varphi$. In this case a list of terms $\tau_1, \ldots, \tau_n$ is used and a parallel substitution takes place.

You can select the rule from the rule list. If there is more than one formula with a universal quantifier in the antecedent, the first step is the selection of formula you want to apply the rule on. Just select the formula and click ‘Okay’. Instead of selecting the rule from the rule list you can also select the rule and the formula in one step with the mouse. Just click with the right mouse button anywhere in the formula.

The next step is the input of the instantiation. A window appears that displays the formula, the list of variables to instantiate, a list of suggestions (or no suggestion), a text field for your input, and three buttons. Now you have three possibilities. You can select a substitution from the list of suggestions. The substitution appears in the text field. Or you can enter a substitution yourself. For every variable to instantiate you enter one term. Several terms are separated by comma. E.g., if the variables to substitute are $[x, y]$, you must enter something like $3, x + y$. (You can also type $[3, x + y]$.) This means that $x$ will be substituted by 3, and $y$ by $x + y$ in parallel! (i.e. the second substitution is not equivalent to $3 + y$.) The third possibility is to choose a previously typed substitution by clicking on the down arrow at the right of the text field. No matter how you selected the substitution, you can still edit it in the text field. This is very convenient if a suggestion is slightly wrong. Just select and edit it.

After you are satisfied with your substitution, one more decision remains: You can discard or keep the quantified formula. Discarding it is useful if you are sure that you don’t need it anymore, because it makes the sequent smaller, easier to read, and you are not tempted to use the rule again with useless instances. However, the goal may become unprovable if your current instance is not correct or you must instantiate the quantifier more than once. It’s your choice! If you want to keep the quantified formula select the button ‘Okay (Keep)’, otherwise select ‘Okay (Discard)’.

If your substitution contains a variable that is not free in the sequent, a confirmation window appears whether you really want to use that substitution. Normally, a new variable (not occurring free in the sequent) indicates a typing error, because such a substitution is normally useless. However, there are cases (that occur in the exercises!) where the substitution is really irrelevant. Confirm the window this ‘Yes’ or ‘No’, and the rule is applied.
CHAPTER 2. PREDICATE LOGIC PROOFS IN KIV

2.4.2 insert equation

Equations can be inserted with the rule \textit{insert equation}

\[
\frac{\sigma = \tau, \Gamma \vdash \Delta}{\sigma = \tau, \Gamma \vdash \Delta} \quad \text{\textit{(insert equation)}}
\]

In this rule \(\sigma = \tau\) is an arbitrary equation of the antecedent. The system displays all possible equations together with their symmetric counterpart \(\tau = \sigma\). Since predicates in the antecedent can be read as \textit{predicate = true} (and predicates in the succedent as \textit{predicate = false}) they are also listed. Beware: every equation and predicate is offered for insert equation even if there is no useful usage for the equation!

Now you can select one equation to insert. There are two buttons: the left is labeled \textit{Okay (keep)}, the right \textit{Okay (discard)}. ‘keep’ means that the equation remains in the new goal (as shown above), ‘discard’ means that the equation will be discarded after insertion (with \textit{weakening}). Beware: it is your decision! If you discard the equation the goal may become unprovable. (This happens if the equation is false, e.g. \(0 = 1\), or if the right hand side contains variables that still occur in the sequent after insert equation.)

After selecting the equation to insert (and whether to keep or to discard it) the positions where the equation can be inserted are displayed (marked with \#\#\#). It is possible to substitute all occurrences of \(\sigma\) by \(\tau\) (the normal case) or just one (or several occurrences). You will not get this question if no or only one possibility exists.

The rule can also be applied in a context sensitive manner. If you click with the right mouse button anywhere on the left hand side of an equation, a window with the text \textit{insert equation (left to right)} appears. This will insert the equation left to right everywhere, i.e. all occurrences of the right side are substituted by the left side, and the equation is discarded. If you click on the right hand side the left side is replaced by the right side, again everywhere, and again the equation is discarded. If you want to keep the equation you have to use the rule from the rule list.

2.4.3 insert lemma

To use axioms or theorems the following rule exists:

\[
\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Theta(\Gamma'\), \Delta} \quad \text{\textit{(insert lemma)}}
\]

\begin{itemize}
  \item \(\Gamma' \vdash \Delta'\) is the lemma (theorem)
  \item \(\Theta\) is a parallel substitution for the free variables of the lemma
\end{itemize}

In order to apply a theorem \(\Gamma' \vdash \Delta'\) it is necessary to enter a substitution \(\Theta\) for the free variables of the theorem to obtain the instances for the actual goal. The system shows the free variables of the sequent and you have to enter a list of terms \(\text{term}_1, \text{term}_2, \ldots, \text{term}_n\) for every variable separated by comma. The list of terms may be optionally enclosed in square brackets. The selection of the substitution is identical to the quantifier instantiation \textit{all left}. However, there is only one ‘Okay’ button because you can’t ‘keep’ the lemma. If you need it more than once you have to use the rule several times.

The rule adds three new premises to the proof tree. The first one is the theorem itself. This premise stays open and is managed by the \textit{correctness management} which takes care that no cyclical proof dependencies occur. (For example you prove the theorem A with the lemma B and the lemma B with the theorem A). For the second premise you have to show that the (instances of the) precondition for the lemma holds. This is the conjunction of the formulas of the antecedent of the lemma, written as \(\Theta\Gamma'\). If the lemma has no conditions then \(\Theta\Gamma'\) is true and the statement is an axiom. In the last premise the results (i.e. the disjunction of the formulas of the succedent, often only one formula) are added to the previous goal.
A similar rule is insert axiom. An axiom or theorem (with all quantifiers added) is added to the antecedent of the goal. If you instantiate the quantifiers with all left, discard the formula, and make a case distinction (provided the axiom is $\varphi \rightarrow \psi$), you obtain the same premises as with insert lemma.

## 2.5 Basic rules

The following notation is used for the rules: $\varphi, \psi$ denote arbitrary formulas, $\Gamma, \Delta$ stand for arbitrary lists of formulas. $\sigma, \tau$ denote terms. The rules are written in a little bit simplified notation. Even though we write $\Gamma \vdash \varphi \land \psi, \Delta$ the rule conjunction left is applicable on any conjunction in the antecedent, not only on the first formula of the antecedent. The correct notation would be $\Gamma_1, \varphi, \Delta_2 \vdash \Delta$.

### 2.5.1 Axioms

\[
\begin{align*}
\varphi, \Gamma &\vdash \varphi, \Delta \quad \text{(axiom)} \quad \text{false, } \Gamma &\vdash \Delta \quad \text{(false left)} \\
\Gamma &\vdash \tau = \tau, \Delta \quad \text{(reflexivity right)} \quad \text{true, } \Gamma &\vdash \Delta \quad \text{(true right)}
\end{align*}
\]

### 2.5.2 Propositional and equational rules

\[
\begin{align*}
\Gamma &\vdash \varphi, \Delta \quad \text{(negation left)} & \quad \psi, \Gamma &\vdash \Delta \quad \text{(negation right)} \\
\varphi, \psi, \Gamma &\vdash \Delta \quad \text{(conjunction left)} & \quad \Gamma &\vdash \varphi, \psi, \Delta \quad \text{(conjunction right)} \\
\varphi, \Gamma &\vdash \Delta \quad \text{(disjunction left)} & \quad \psi, \Gamma &\vdash \Delta \quad \text{(disjunction right)} \\
\varphi, \psi, \Gamma &\vdash \Delta \quad \text{(implication left)} & \quad \Gamma &\vdash \varphi \rightarrow \psi, \Delta \quad \text{(implication right)} \\
\varphi, \psi, \Gamma &\vdash \Delta \quad \text{(equivalence left)} & \quad \varphi, \Gamma &\vdash \psi, \Delta \quad \psi, \Gamma &\vdash \varphi, \Delta \quad \text{(equivalence right)} \\
\Gamma &\vdash \Delta' \quad \text{(weakening, } \Gamma' \subseteq \Gamma, \Delta' \subseteq \Delta) & \quad \Gamma &\vdash \Delta \quad \varphi, \Gamma &\vdash \Delta \quad \text{(cut formula)} \\
\sigma = \tau, \Gamma &\vdash \Delta' \quad \text{(insert equation)} & \quad \Gamma' &\vdash \Delta \quad \sigma = \tau, \Gamma &\vdash \Delta' 
\end{align*}
\]

### 2.5.3 Quantifiers

Note: $\varphi^x_{\tau}$ is the substitution of $x$ by $\tau$ in $\varphi$.

- $\varphi^x_{\tau}, \forall \ x. \varphi, \Gamma \vdash \Delta$ (all left) \\
- $\exists \ x. \varphi, \Gamma \vdash \Delta$ (exists right)

$\tau$ may be an arbitrary term.

The rule is also applicable for a list of quantified variables $\forall \ x_1, \ldots, x_n. \varphi$. In this case a list of terms $\tau_1, \ldots, \tau_n$ is used and a parallel substitution takes place.
The quantified formula can optionally be discarded.

- \( \varphi_y, \Gamma \vdash \Delta \) (exists left)
- \( \Gamma \vdash \varphi_y, \Delta \) (all right)

\( y \) is a new variable, i.e. one that does not occur in the free variables of \( \varphi, \Gamma, \Delta \).

The rule is also applicable for a list of quantified variables \( \forall x_1, \ldots, x_n. \varphi \). In this case a list of new variables \( y_1, \ldots, y_n \) is used and a parallel substitution takes place.

2.5.4 Theory rules

- \( \vdash \varphi(c) \)
  \[ \varphi(x) \vdash \varphi(f(x)) \] (structural induction)
  \( \varphi = \forall x'. \Gamma \rightarrow \Delta \) with \( x' = \text{Free}((\Gamma \rightarrow \Delta) \setminus x) \)

The specification contains a generation principle sort\((x)\) generated by \( c, f \)

- \( \vdash \forall x. \varphi \)
  \[ \forall x. \varphi, \Gamma \vdash \Delta \] (insert axiom/insert spec-axiom)
  \( \varphi \) is an axiom, \( \not \in \) the free variables of \( \varphi \).

insert axiom applies a lemma from the current specification, insert spec-axiom from a sub-specification.

- \( \vdash \Theta(\Gamma'), \Delta \)
  \[ \Theta(\Delta''), \Gamma \vdash \Delta \] (insert lemma/insert spec-lemma)
  \( \Gamma' \vdash \Delta' \) is the lemma (theorem)
  \( \Theta \) is a parallel substitution for the free variables of the lemma.

insert lemma applies a lemma from the current specification, insert spec-lemma from a sub-specification.

- \( \vdash \varphi \rightarrow \sigma = \tau \)
  \[ \Gamma \vdash \Theta(\varphi), \Delta \]
  \[ \Theta(\varphi), \Gamma^{\Theta(\tau)} \vdash \Delta^{\Theta(\sigma)} \] (insert rewrite lemma)
  \( \vdash \varphi \rightarrow \sigma = \tau \) is the rewrite theorem.
  \( \Theta \) is a parallel substitution for the free variables of the theorem. (Computed automatically.)

2.6 Heuristics for basic rules

If you have worked for a while with the basic rules of the sequent calculus you will see that a lot of steps always repeat. For example, you will always apply rules for propositional connectives that do not introduce case distinctions. Such regular patterns of the proof can be detected by the system and it can apply the corresponding rules automatically. Heuristics take the decision from the user what has to be done next. But the user has to decide which heuristics to use and which to omit. They can be switched on/off at any position in the proof. If a heuristic is not applicable on the actual sequent the system tries to apply the next heuristic. If there are no more applicable heuristics the system stops the automatic proof attempt and asks the user for an interaction. The system displays all rules that are applicable on the current goal.

The heuristics are chosen by the menu entry Control - Heuristics. If you are working with the basic rules of the sequent calculus a two column window appears where the left column contains all heuristics for basic rules, and the right column the currently selected, active heuristics. You can add heuristics by clicking on them in the left window. The order of the application of the heuristics corresponds to the order of their selection. By clicking on the OK button the heuristics are applied on the current goal. You can turn the heuristics on or off by clicking 'Use Heuristics'.
in the lower left bottom of the proof window. If no heuristics are selected, and ‘Use Heuristics’ is
deselected, clicking on the field is a shortcut to the menu entry Control – Heuristics.

Heuristics are only heuristics: They may do the wrong thing, or something useless. And
the order of the heuristics can have considerable impact on the size of the proof. (E.g. if you do case
distinctions too early.) The following heuristics are for the basic rules:

- **axiom**: This heuristic searches for a possibility to apply the axiom rules axiom, false left,
  true right, and reflexivity right.

  This heuristic closes the goal, i.e. it can’t do anything wrong or useless, and should be always
  used as the first heuristic.

- **prop simplification**: applies the propositional rules with one premise negation left, negation
  right, conjunction left, disjunction right, and implication right.

  Can’t do something wrong or useless.

- **prop split**: applies the propositional rules with two premises disjunction left, conjunction
  right, implication left, equivalence left, and equivalence right.

  Can’t do something wrong, but may do case distinctions too early.

- **smart basic case distinction**: tries to apply propositional rules with two premises where
  one premise can be immediately closed, so that no ‘real’ case distinction is introduced.

  Can’t do something wrong or useless.

- **insert equation**: inserts equation with the rule insert equation without discarding the
  equation.

  May be useless, but not wrong.

- **Quantifier closing**: applies the rules all left and exists right if it can find an instantiation
  that will close the goal.

  Can’t do something wrong or useless.

- **discard quantifier**: applies the rules exists left and all right.

  Can’t do something wrong or useless.

- **Quantifier**: applies the rules all left and exists right if it finds (or rather guesses) an instan-
  tiation that may be useful for the proof. This guessing is not perfect at all (and can not be
  due to the undecidability of predicate logic), and therefore this heuristic is “dangerous” in
  multiple regard: It can happen that this heuristic tries unnecessary instances for a quantifier,
  thereby producing a large proof tree. It can also happen that this heuristic does not find
  the correct instance and you have to insert it yourself (maybe several times if unnecessary
  instances arose). In the worst case this heuristic tries again and again an incorrect instance
  an the system does not stop at all. In this case press the Stop button. This will enforce
  a stop. But for all that this heuristic often finds the correct instance and is therefore very
  useful.

- **batch mode**: This ‘heuristic’ just switches to the next goal. It should be only used as the
  last heuristic. Its effect is that all branches of the proof are treated by the heuristics as much
  as possible.
2.7 Exercise 1

For every exercise (except the last one) the basic rules have to be used, i.e. the option *Use basic rules* (by selecting the menu command **Control – Options**) must be switched on. This should be the case by default.

**Exercise 2.1 Propositional logic**

Work on the specification `proplogic` and switch on the option to use basic rules (with the menu command **Control – Options**). Then prove without heuristics the three propositional axioms of the Hilbert calculus:

- **Hilbert-1:** \(\vdash \varphi \rightarrow (\psi \rightarrow \varphi)\)
- **Hilbert-2:** \(\vdash (\varphi \rightarrow \psi) \rightarrow (\neg \psi \rightarrow \neg \varphi)\)
- **Hilbert-3:** \(\vdash ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)) \rightarrow (\varphi \rightarrow (\psi \rightarrow \chi))\)

**Exercise 2.2 (Example) Propositional logic**

Consider the following logical puzzle:

The oracle says:

The left and the right path lead to Delphi if the middle path leads somewhere else.

If at most one of the left and middle path lead to the oracle, then the right path is wrong.

However, if the right or left path leads to Delphi, the middle path definitely does not lead there.

Which of the three paths leads to Delphi?

To solve this puzzle we can use three propositional constants `left`, `right`, and `middle`. Their intuitive meaning is: If the constant is true the path leads to Delphi, otherwise not. For example, if `left` is true the left path leads to Delphi. We can now formalize the three sentences (or known facts):

\[
\begin{align*}
\neg \text{middle} & \rightarrow \text{left} \land \text{right} \\
\neg (\text{left} \land \text{middle}) & \rightarrow \neg \text{right} \\
\text{left} \lor \text{right} & \rightarrow \neg \text{middle}
\end{align*}
\]

Together we have

\[
\text{(facts)} = (\neg \text{middle} \rightarrow \text{left} \land \text{right})
= \land (\neg (\text{left} \land \text{middle}) \rightarrow \neg \text{right})
= \land (\text{left} \lor \text{right} \rightarrow \neg \text{middle})
\]

(Note the brackets!) Now we have to find a solution. Here it is important to read exactly what is requested. Let’s assume we found out (by hard thinking) that the middle path leads to Delphi. This means a possible solution is `middle`. On the other hand, we may have found out that the middle path leads to Delphi and the other two paths do not. This means another (more complete) solution is `middle \land \neg \text{left} \land \neg \text{right}`. Now we can prove two things:

1. The solution fulfills the facts, i.e. the solution is indeed a solution. (This also proves that the facts are not contradictory!)

Prove \(\langle \text{solution} \rangle \vdash \langle \text{facts} \rangle\):

\[
\begin{align*}
\text{middle} \land \neg \text{left} \land \neg \text{right} \\
\vdash (\neg \text{middle} \rightarrow \text{left} \land \text{right}) \\
\land (\neg (\text{left} \land \text{middle}) \rightarrow \neg \text{right}) \\
\land (\text{left} \lor \text{right} \rightarrow \neg \text{middle})
\end{align*}
\]
(Note the brackets because of the priorities of ¬, ∧, →!) This only works for complete solutions, i.e. if a value is given to all constants.

If we want to check a partial solution, we can prove:

\[ \neg \langle \text{partial solution} \rangle \vdash \neg \langle \text{facts} \rangle \]

\[ \neg \text{middle} \]
\[ \vdash \neg \left( \neg \text{middle} \rightarrow \text{left} \land \text{right} \right) \]
\[ \land \left( \neg \left( \text{left} \land \text{middle} \right) \rightarrow \neg \text{right} \right) \]
\[ \land \left( \text{left} \lor \text{right} \rightarrow \neg \text{middle} \right) \]

2. The solution follows from the facts. (This means the solution is unique.)

Prove \( \langle \text{facts} \rangle \vdash \langle \text{solution} \rangle \):

\[ \neg \text{middle} \rightarrow \text{left} \land \text{right}, \]
\[ \neg \left( \text{left} \land \text{middle} \right) \rightarrow \neg \text{right}, \]
\[ \text{left} \lor \text{right} \rightarrow \neg \text{middle} \]
\[ \vdash \text{middle} \land \neg \text{left} \land \neg \text{right} \]

It is somewhat unsatisfactory that we have to know the solution before we can prove anything. However, we can also use the system to find a solution. We begin ‘proving’ with only the facts, i.e. \( \langle \text{facts} \rangle \vdash \langle \text{empty succedent} \rangle \):

\[ \neg \text{middle} \rightarrow \text{left} \land \text{right}, \]
\[ \neg \left( \text{left} \land \text{middle} \right) \rightarrow \neg \text{right}, \]
\[ \text{left} \lor \text{right} \rightarrow \neg \text{middle} \]
\[ \vdash \]

Using the heuristics we get a proof tree that contains one open premise:

\[ \text{middle} \vdash \text{right}, \text{left}, \text{right} \]

And this is the solution! \textbf{middle} must be true and \textbf{left} and \textbf{right} must be false. The open premise also shows that the facts are not contradictory. However, the result may be more complicated to interpret:

- We must use only equivalence rules, i.e. rules that are true in both directions (i.e. the conclusion is equivalent to the conjunction of the premises). This means that e.g. the rule\textbf{ weakening formulas} may not be used.

- One constant may be missing. If the result is just \( \vdash \text{right}, \text{left} \), then we don’t know anything about the middle path (it may or may not lead to Delphi). It depends on the problem whether this is a correct solution.

- Often the resulting proof tree has more than one open premise. Then every premise yields a possible solution that fulfills the facts. However, this does not mean that the solution follows from the facts (i.e. is unique). The solution is unique if all premises yield the same solution (and this solution contains all constants).

If we have the two premises

\[ \text{middle} \vdash \text{left}, \text{right} \]

we have two solutions, \textbf{middle} \land \neg \textbf{left} (and we don’t know anything about \textbf{right}), and a second solution \( \neg \textbf{right} \) (and we don’t know anything about \textbf{middle} and \textbf{left}). This is certainly not a correct solution!

If the premises are \textbf{middle} \vdash \textbf{left}, and \textbf{middle} \vdash \textbf{right}, the solution is not unique. \textbf{middle} must be true, but we don’t know anything about \textbf{left} or \textbf{right}. (This may be a correct solution.)

- The resulting proof tree is closed. This means that the facts are contradictory. (Which may be a correct solution.)
CHAPTER 2. PREDICATE LOGIC PROOFS IN KIV

Your task: Work on the specification proplogic, use basic rules, and any heuristics you like (a good selection is all heuristics – how does their order influence the proofs?). Prove the three theorems delphi-try, delphi-solution, delphi-fulfills, and compare the proof trees. (Of course, delphi-try is not provable.)

Exercise 2.3 Propositional logic

Work on the specification proplogic, use basic rules, and any heuristics you like. Formalize and prove the following puzzle using propositional logic. Use the propositional constants a, b, c. What is their intuitive meaning?

Mr. McGregor, a London shopkeeper, phoned Scotland Yard that his shop had been robbed. Three suspects A, B, C were rounded up for questioning. The following facts were established:

1. Each of the men A, B, C had been in the shop on the day of the robbery, and no one else had been in the shop that day.
2. If A was guilty, then he had exactly one accomplice.
3. If B is innocent, so is C.
4. If exactly two are guilty, then A is one of them.
5. If C is innocent, so is B.

Whom did Inspector Craig indict?

Add the formalization as a theorem in the file sequents (see section 2.1.2). Note that you should enter a sequent, not a formula, and that you can only use the symbols defined in the specification.

Exercise 2.4 Propositional logic

Work on the specification proplogic, use basic rules, and any heuristics you like. Formalize and prove the following puzzle using propositional logic. Use the propositional constants af, bf, ad, bd. (What is their intuitive meaning? a = ‘amplifier’, b = ‘bycicle’, f = ‘flux generator’, d = ‘dragon trap’)

Engineer Trurl wants to construct two machines (a probabilistic flux generator and a universal dragon trap) from a heap of scrap metal. The most valuable components at his hands are a chance amplifier and a bicycle. His colleague Klapauzius asks, ‘Is it true that if you need the bicycle for the probabilistic flux generator, and the chance amplifier for the dragon trap if and only if you also need the bicycle for it, you then don’t have to install the amplifier in the flux generator?’ Trurl ponders ‘If this statement is true, then I need the bicycle for exactly one machine, and the same holds for the chance amplifier. On the other hand, if I need the bicycle at all then the statement must be false. But in each case I do not need both components for both machines.’

Now, Which component is needed for which machine?

Show that a unique solution exists. What is it?

Exercise 2.5 Predicate logic

Work on the specification predlogic, use basic rules or not (you can change it with the menu Control – Options and Use Basic Rules), and any heuristics you like. Prove the following theorems:
1. Allneg : $\vdash (\forall x. \neg p(x)) \iff \neg \exists x. p(x)$;
2. Exneg : $\vdash (\exists x. \neg p(x)) \iff \neg \forall x. p(x)$;
3. Allimpleft : $\vdash ((\forall x. p(x)) \rightarrow q(y)) \iff \exists x. p(x) \rightarrow q(y)$;
4. Allimpright : $\vdash (p(x) \rightarrow \forall y . q(y)) \iff \forall y. p(x) \rightarrow q(y)$;
5. Eximpleft : $\vdash ((\exists x. p(x)) \rightarrow q(y)) \iff \forall x. p(x) \rightarrow q(y)$;
6. Eximpright : $\vdash (p(x) \rightarrow \exists y . q(y)) \iff \exists y. p(x) \rightarrow q(y)$;
7. Exor : $\vdash (\exists x. p(x) \lor q(x)) \iff ((\exists x. p(x)) \lor (\exists x. q(x)))$;
8. Exands : $\vdash (\exists x. p(x) \land q(y)) \iff (\exists x. p(x)) \land q(y)$;
9. Alland : $\vdash (\forall x. p(x) \land q(x)) \iff ((\forall x. p(x)) \land (\forall x. q(x)))$;
10. Allands : $\vdash (\forall x. p(x) \land q(y)) \iff (\forall x. p(x)) \land q(y)$;
11. Allors : $\vdash (\forall x. p(x) \lor q(y)) \iff (\forall x. p(x)) \lor q(y)$;
12. Allors : $\vdash (\forall x. p(x) \lor q(x)) \iff (\forall x. p(x)) \lor q(x)$;

Note that these rules allow to shift quantifiers in a formula (possibly with renaming of bounded variables). Specifically, all quantifiers can be shifted to the beginning of a formula so that the body is quantifier free (the only problem being equivalences, which must be first split into a conjunction of two implications). The resulting normal form is not unique. It is called a prenex form.

**Exercise 2.6 Predicate logic**

In this exercise you have to invent formulas yourself. Note that you cannot use arbitrary variables, but only those that are declared in the specification! (Try the menu command View – Specification or click in daVinci on the node and select ‘View’.)

Work on the specification predlogic, use basic rules or not, and any heuristics you like. Solve the following tasks

1. prove exall-allex : $\vdash (\exists x. \forall y. pr(x,y)) \rightarrow \forall y. \exists x. pr(x,y)$;
2. prove allex-exex : $\vdash (\forall x. \exists y. pr(x,y)) \rightarrow \exists x. \exists y. pr(x,y)$;
3. prove not-exall : $\vdash (p(a, a), p(b, b), \neg pr(a, b), \neg pr(b, a), \forall x. x = a \lor x = b) \vdash (\forall y. \exists x. pr(x,y)) \rightarrow \exists x. \forall y. pr(x,y)$;
4. Show that the following quantifier shift is not valid by adding a suitable precondition so that the sequent becomes provable (similar to the previous task).
   not-exand : (add precondition here) $\vdash \neg ((\exists x. p(x) \land q(x)) \rightarrow ((\exists x. p(x)) \land (\exists x. q(x))))$;
5. Show that the following quantifier shift is not valid.
   not-allor : (add precondition here) $\vdash \neg ((\forall x. p(x) \lor q(x)) \rightarrow ((\forall x. p(x)) \lor (\forall x. q(x))))$;
6. Transform the following formulas into a prenex form and prove that the original form is equivalent to the prenex formula.
   prenex1 : $\vdash ((\forall x. p(x) \rightarrow pr(x,y)) \rightarrow ((\exists y. p(y)) \rightarrow (\exists z. pr(y,z)))) \leftrightarrow (add prenex form here)$;
7. prove pelletier19 : $\vdash \exists x. \forall y. \forall z. (p(y) \rightarrow q(z)) \rightarrow (p(x) \rightarrow q(x))$.

This exercise shows a weakness of the sequent calculus because it is not possible to apply rules inside a formula. The first rule to apply must be exists right, but it is unclear what instance to use.
The formula is in prenex form. Untransform the formula on a piece of paper using the shift rules of the previous exercise to obtain a formula where the quantifiers are inside the propositional junctors. Solve this goal (again on paper). Inspect which quantifiers are instantiated with which variables (and why). Then solve the original goal.

**Exercise 2.7 Predicate logic**

Work on the specification predlogic, use basic rules or not, and any heuristics you like. Formalize and prove the following statements:

1. A barber is a man who shaves precisely those men who do not shave themselves.
   
   It follows that no barber exists.
   
   (use \(\text{barber}(x)\) and \(\text{shaves}(x,y)\))

2. Everybody loves my baby, but my baby loves nobody but me.
   
   This means that ‘me’ and ‘baby’ are identical.
   
   (use \(\text{loves}(x,y)\), \(\text{baby}\), and \(\text{me}\))

3. If everyone loves somebody and no one loves everybody, then someone loves some and doesn’t love others.
   
   (use \(\text{loves}\))

Remember that you can only use the symbols of the specification. This is also true for variables. The specification contains the variables \(x, y, z\), etc., that you have to use.

**Exercise 2.8 Natural numbers**

In the specification \(\text{nat}\), the natural numbers are specified. This specification contains the constant 0, the successor function \(\text{succ}\), and an (infix) addition function \(+\). The axioms, which describe the natural numbers are

- distinctiveness: \(0 \neq \text{succ}(n)\);
- injectivity: \(\text{succ}(m) = \text{succ}(n) \iff m = n\);
- add-zero: \(n + 0 = n\);
- add-succ: \(m + \text{succ}(n) = \text{succ}(m + n)\)

plus the induction principle expressed through the generated by clause as described in chapter 1.3.

(Note: you can view the specification with the command \text{View} – \text{Specification}.)

Prove with the basic rules and without heuristics the correctness of the following theorems:

- \(\text{lem-01}: \vdash 0 + n = n\)
- \(\text{lem-02}: \vdash \text{succ}(m) + n = \text{succ}(m + n)\)
- \(\text{com}: \vdash m + n = n + m\)

Hint: The first two theorems are propositions which should help to prove \(\text{com}\). Therefore prove the theorems in the above order without the use of \(\text{com}\). All proofs need induction.

**Exercise 2.9 Rewriting**

This exercise shows that it must not be tedious to prove the theorem from the exercise above. The convenient proof technique is called “term rewriting”. This proof method uses equations \(\sigma = \tau\) in the following way as “rewrite rules”: If there is an instance of the term \(\sigma\) in the goal it will always be substituted through the corresponding instance of \(\tau\). (This requires that all variables of \(\tau\) also appear in \(\sigma\).) Therefore \(\tau\) should be ‘easier’ than \(\sigma\). The substitution of terms through easier ones can happen recursively as long as possible. Term rewriting is (beneath other things) done by the \text{simplifier} rule from the normal calculus.

Switch off the basic rules by using the menu command \text{Control} – \text{Options}, and unmark the option \text{Use Basic Rules}. Now try the proofs for \(\text{lem-01}, \text{lem-02}, \text{and com}\) again (with \text{Proof Reprove}). If a theorem was successfully proved, it will used as a simplifier rule in the following proofs. You can also use heuristics for the proof if you want. Therefore select \text{PL Heuristics + Struct. Ind.}. 

Bibliography


