
Using KeY

by

Wolfgang Ahrendt

10.1 Introduction

This whole book is about the KeY *approach* and *framework*. This chapter now focuses on the KeY *system*, and that entirely from the user's perspective. Naturally, the graphical user interface (GUI) will play an important role here. However, the chapter is not all about that. Via the GUI, the system and the user communicate, and interactively manipulate, several artefacts of the framework, like formulae of the used logic, proofs within the used calculus, elements of the used specification languages, among others. Therefore, these artefacts are (in parts) very important when using the system. Even if all of them have their own chapter/section in this book, they will appear here as well, in a somewhat superficial manner, with pointers given to in-depth discussions in other parts.

We aim at a largely self-contained presentation, allowing the reader to follow the chapter, and to *start* using the KeY system, without necessarily having to read several other chapters of the book before. The reader, however, can gain a better understanding by following the references we give to other parts of the book. In any case, we strongly recommend to read Chapter 1 beforehand, where the reader can get a picture of what KeY is all about. The other chapters are *not* treated as prerequisites to this one, which of course imposes limitations on how far we can go here. Had we built on the knowledge and understanding provided by the other chapters, we would be able to guide the user much further into to the application of KeY to larger resp. more difficult scenarios. However, this would raise the threshold for getting started with the system, thereby contradicting the philosophy of the whole project. The KeY framework was designed from the beginning for being usable *without* having to read a thick book first. Software verification is a difficult task anyhow. Neither the system nor the used artefacts (like the logic) should add to that difficulty, and are designed to instead lower the threshold for the user. The used logic, *dynamic logic* (DL), features transparency w.r.t. the

programs to be verified, such that the code literally appears in the formulae, allowing the user to relate back to the program when proving properties about it. The “tacet” language for the declarative implementation of both, rules and lemmas, is kept so simple that we can well use a rule’s declaration as a tooltip when the user is about to select the rule. The calculus itself is, however, complicated, as it captures the complicated semantics of JAVA. Still, most of these complications do not concern the user, as they are handled in a fully automatic way. Powerful strategies relieve the user from tedious, time consuming tasks, particularly when performing *symbolic execution*.

In spite of a high degree of automation, in many cases there are significant, non-trivial tasks left for the user. It is the very purpose of the GUI to support those tasks well. When proving a property which is too involved to be handled fully automatically, certain steps need to be performed in an interactive manner, in dialogue with the system. This is the case when either the automated strategies are exhausted, or else when the user deliberately performs a strategic step (like a case distinction) manually, *before* automated strategies are invoked (again). In the case of human-guided proof steps, the user is asked to solve tasks like: *selecting a proof rule* to be applied, *providing instantiations* for the proof rule’s *schema variables*, or *providing instantiations for quantified variables* of the logic. In turn, the system, and its advanced GUI, are designed to support these steps well. For instance, the selection of the right rule, out of over 1500(!), is greatly simplified by allowing the user to highlight any syntactical sub-entity of the proof goal simply by positioning the mouse. A dynamic context menu will offer only the few proof rules which apply to this entity. Furthermore, these menus feature tooltips for each rule pointed to. When it comes to interactive variable instantiation, *drag-and-drop* mechanisms greatly simplify the usage of the instantiation dialogues, and in some cases even allow to omit explicit rule selection. Other supported forms of interaction in the context of proof construction are the inspection of proof trees, the pruning of proof branches, stepwise backtracking, and the triggering of proof reuse.

Performing interactive proof steps is, however, only one of the many functionalities offered by the KeY system. Also, these features play their role relatively late in the process of verifying programs. Other functionalities are (we go backwards in the verification process): controlling the automated strategies, adding lemmas and generating corresponding proof obligations, customising the calculus (for instance by choosing either of the mathematical or the JAVA semantics for integers), and generating proof obligations from specifications. Those features (and several others to be discussed below) comprise what we call the “core KeY system”, “stand-alone KeY system”, or “stand-alone KeY prover”.

On top of the core system, there exist integrations into (currently two) standard tools for (JAVA) software development, as was discussed in the introduction to this book (Chap. 1, see particularly Fig. 1.1). One of them is

the commercial CASE¹ tool Borland Together Control Center, the other is the open source IDE Eclipse. In both cases, users can develop the whole software project, comprising both specifications and implementations, entirely in the frame of either of these (KeY-enhanced) tools, which offer the *extended functionality* of generating proof obligations from selected entities of specifications, and starting up the KeY prover accordingly.

Working with the KeY system has therefore many aspects, and there are many ways to give an introduction into those. In this chapter, we will take an “inside out” approach, starting with the *core* prover, describing *how* it communicates *which artefacts* for *which purpose* with the user, when proving a formula at hand.

In general, we will discuss the usage of the system by means of rather (in some cases extremely) simple examples. Thereby, we try to provide a good understanding of the various ingredients before their combination (seemingly) complicates things. Also, the usage of the prover will sometimes be illustrated by at first performing basic steps manually, and demonstrating automation thereafter. Please note that the toy examples used all over this chapter serve the purpose of step by step introducing the concepts and usage of the KeY system. They are not suitable for giving any indication of the capabilities of the system. (See Part IV instead.)

Before we start, there is one more basic issue which should be reflected on at this point. The evolution of both, the KeY *project* in general, and the KeY *system* in particular, has been very dynamic up to now, and will continue to be so. As far as the *system* and its GUI is concerned, it has been constantly improved and will be modified in the future as well. The author faces the difficult task of not letting the description of the tool’s usage depend too much on its current appearance. The grouping of menus, the visual placement of panes and tabs, the naming of operations or options, all that can potentially change. Also, on the more conceptual level, things like the configuration policy for strategies and rule sets, among others, cannot be assumed to be frozen for all times. Even the theoretical grounds will develop further, as KeY is indeed a *research project*. A lot of ongoing research does not yet show in the current release of the KeY system, like support for mainstream languages other than JAVA, support for *disproving* wrong formulae, or the combination of deductive verification with static analysis, to name just very few. These, and others, will enhance the framework, and find their way into the system. We make a strong effort, not only in this chapter, to make the material valuable for the understanding and usage also of the future KeY.

The problem of describing a dynamic system is approached from three sides. First, we will continue to keep available the book release of the system, KeY 1.0, on the KeY book’s web page. Second, in order to not restrict the reader to that release only, we will try to minimise the dependency of the material on the current version of the system and its GUI. Third, whenever

¹ CASE stands for Computer-Aided Software Engineering.

we talk about the specific location of a pane, tab, or menu item, or about key/mouse combinations, we stress the dynamic nature of such information *in this way*. For instance, we might say that “one can trigger the run of an automated strategy which is restricted to a highlighted term/formula *by Shift + click on it*”. Menu navigation will be displayed by connecting the cascaded menus/sub-menus with “→”, like “**Options** → **Decision Procedure Config** → **Simplify**”. Note that menu navigation is release dependent as well.

This chapter is meant for being read with the KeY system up and running. We want to explore the system *together* with the reader, and reflect on whatever shows up along the path. Downloads of KeY, particularly its book version, KeY 1.0, are available on the project page, www.key-project.org. The example input files, which the reader frequently is asked to load, can be found on the web page for this book, www.key-project.org/thebook, as well as in your KeY system’s installation, under `examples/BookExamples`.

10.2 Exploring Framework and System Simultaneously

Together with the reader, we want to open, for the first time, the KeY system, in order to perform first steps and understand the basic structure of the interface. We start the stand-alone KeY prover *by running* `bin/runProver` *in your KeY installation directory*. The **KeY-Prover** main window, together with a **Proof Assistant**² pops up. The latter is simply a message window, which comments on pre-selected menus or actions the user is about to make.

Like many window-based GUIs, the main window offers several menus, a toolbar, and a few panes, partly tabbed. Instead of enumerating those components one after another, we immediately load an example to demonstrate some basic interaction with the prover.

10.2.1 Exploring Basic Notions and Usage: Building a Propositional Proof

In general, the KeY prover is made for proving formulae in *dynamic logic* (DL), an extension of *first-order logic*, which in turn is an extension of *propositional logic*. We start with a very simple propositional formula, when introducing the usage of the KeY prover, because a lot of KeY concepts can already be discussed when proving the most simple theorem.

Loading the First Problem

The formula we prove first is contained in the file `andCommutes.key`. In general, `.key` is the suffix for what we call *problem files*, which may, among other things, contain a formula to be proved. (The general format of `.key` files is documented in Appendix B.) For now, we look into the file `andCommutes.key` itself (using your favourite editor):

² If the **Proof Assistant** does not appear, please check **Options** → **Proof Assistant**.


— KeY Problem File —

```

\predicates {
    p;
    q;
}
\problem {
    (p & q) -> (q & p)
}

```

KeY Problem File —

The `\problem` block contains the formula to be proved (with `&` and `->` denoting the logical “and” and “implication”, respectively). In general, all functions, predicates, and variables appearing in a problem formula are to be declared beforehand, which, in our case here, is done in the `\predicates` block. We load this file *by File* → **Load ...** (or selecting  in the tool bar) and navigating through the opened file browser. The system not only loads the selected `.key` file, but also the whole calculus, i.e., its rules.

Reading the Initial Sequent

Afterwards, we find the text `==> p & q -> q & p` displayed in the **Current Goal** pane. This seems to be merely the `\problem` formula, but actually, the arrow “`==>`” turns it into a *sequent*. KeY uses a sequent calculus, meaning that sequents are the basic artefact on which the calculus operates. Sequents have the form $\phi_1, \dots, \phi_n \Rightarrow \phi'_1, \dots, \phi'_m$, with ϕ_1, \dots, ϕ_n and ϕ'_1, \dots, ϕ'_m being two (possibly empty) comma-separated lists of formulae, distinguished by the sequent arrow “ \Rightarrow ” (written as “`==>`” in both input and output of the KeY system). The intuitive meaning of a sequent is: if we assume all formulae ϕ_1, \dots, ϕ_n to hold, then *at least one* of the formulae ϕ'_1, \dots, ϕ'_m holds. In our particular calculus, the order of formulae within ϕ_1, \dots, ϕ_n and within ϕ'_1, \dots, ϕ'_m does not matter. Therefore, we can for instance write “ $\Gamma \Rightarrow \phi \rightarrow \psi, \Delta$ ” to refer to sequents where *any* of the right-hand side formulae is an implication. (Γ and Δ are both used to refer to arbitrary, and sometimes empty, lists of formulae.) We refer to Chap. 2, Sect. 2.5, for a proper introduction of a (simple first-order) sequent calculus. The example used there is exactly the one we use here. We recommend to double-check the following steps with the on paper proof given there.

We start proving the given sequent with the KeY system, however in a very interactive manner, step by step introducing and explaining the different aspects of the calculus and system. This purpose is really the *only* excuse to *not* let KeY prove this automatically.

Even if we perform all steps “by hand” for now, we want the system to minimise interaction, e.g., by not asking the user for an instantiation if the system can find one itself. For this, please make sure that the “Minimize interaction” option (at **Options** → **Minimize interaction**) is checked.

Applying the First Rule

The sequent $\Rightarrow p \ \& \ q \ \rightarrow \ q \ \& \ p$ displayed in the **Current Goal** pane states that the formula $p \ \& \ q \ \rightarrow \ q \ \& \ p$ holds *unconditionally* (no formula left of “ \Rightarrow ”), and *without alternatives* (no other formula right of “ \Rightarrow ”). This is an often encountered pattern for proof obligations when starting a proof: sequents with empty left-hand sides, and only the single formula we want to prove on the right-hand side. It is the duty of the *sequent calculus* to, step by step, take such formulae apart, while collecting assumptions on the left-hand side, or alternatives on the right-hand side, until the sheer shape of a sequent makes it trivially true, which is the case when *both sides have a formula in common*. (For instance, the sequent $\phi_1, \phi_2 \Rightarrow \phi_3, \phi_1$ is trivially true. Assuming both, ϕ_1 and ϕ_2 , indeed implies that “at least one of ϕ_3 and ϕ_1 ” hold, namely ϕ_1 .) It is such primitive shapes which we aim at when proving.

“Taking apart” a formula in a sense refers to breaking it up at the top-level operator. The displayed formula $p \ \& \ q \ \rightarrow \ q \ \& \ p$ does not anymore show the brackets of the formula in the problem file. Still, for identifying the leading operator it is not required to memorise the built in operator precedences. Instead, the term structure gets clear when, with the mouse pointer, sliding back and forth over the formula area, as the sub-formula (or sub-term) under the symbol currently pointed at always gets highlighted. To get the whole formula highlighted, the user needs to point to the implication symbol “ \rightarrow ”, so this is where we can break up the formula.

Next we want to *select a rule* which is meant specifically to break up an implication on the right-hand side. This kind of user interaction is supported *by the system offering only those rules which apply to the highlighted formula, resp. term* (or, more precisely, to its leading symbol). A click on “ \rightarrow ” will open a context menu for rule selection, offering several rules applicable to this implication, among them **impRight**:

$$\text{impRight} \frac{\Gamma, \phi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \rightarrow \psi, \Delta}$$

Note that, strictly speaking, both the *premiss* $\Gamma, \phi \Rightarrow \psi, \Delta$ and the *conclusion* $\Gamma \Rightarrow \phi \rightarrow \psi, \Delta$ are not just plain sequents, but sequent *schemata*. In particular, ϕ and ψ are *schema variables*, to be instantiated with the two sub-formulae of the matching implication, when *applying* the rule.

As for any other rule, the *logical meaning* of this rule is downwards (concerning validity): if a sequent matching the premiss $\Gamma, \phi \Rightarrow \psi, \Delta$ is valid, we can conclude that the corresponding instance of the conclusion $\Gamma \Rightarrow \phi \rightarrow \psi, \Delta$ is valid as well. Accordingly, the *operational meaning* during proof construction goes upwards: the problem of proving a sequent which matches $\Gamma \Rightarrow \phi \rightarrow \psi, \Delta$ is reduced to the problem of proving the corresponding instance of $\Gamma, \phi \Rightarrow \psi, \Delta$. During proof construction, a rule is therefore applicable only to situations where the current goal matches the rule’s *conclusion*. The proof will then be extended by the new sequent re-

sulting from the rule’s premiss. (See below for a generalisation to multiple premisses).

To see this in action, we click at **impRight** in order to apply the rule to the current goal. This produces the new sequent $p \ \& \ q \ ==> \ q \ \& \ p$, which becomes the new current goal. By “goal”, we mean a sequent to which no rule is yet applied. By “current goal” we mean the goal in focus, to which rules can be applied currently.

Inspecting the Emerging Proof

The user may have noticed the **Proof** tab *as part of the tabbed pane in the lower left corner*. It displays the structure of the (unfinished) proof we have achieved so far, showing all the nodes of the current proof, numbered consecutively, and labelled either by the name of the rule which was applied to that node, or by “OPEN GOAL” in case of a goal. The *blue* highlighted node is always the one which is detailed in the big pane. So far, this was always a goal, such that the big pane was called “**Current Goal**”. But if the user clicks at an *inner node*, in our case on the one labelled with **impRight**, that node gets detailed in the big pane now called “**Inner Node**”. It shows not only the sequent of that node, but also the *Upcoming rule application*, in a notation we come to in a minute.

Note that the (so far linear) proof tree displayed in the **Proof** tab has its root on the top, and grows downwards, as is common for trees displayed in GUIs. On paper, however, the traditional way to depict sequent proofs is bottom-up, as is done all over in this book. In that view, the structure of the current proof (with the upper sequent being the current goal) is:

$$\frac{p \ \& \ q \ ==> \ q \ \& \ p}{==> \ p \ \& \ q \ -> \ q \ \& \ p}$$

For the on-paper presentation of the proof to be developed, we again refer to Section 2.5. Here, we concentrate on the development and presentation via the KeY GUI instead.

Understanding the First Taclet

With the inner node still highlighted in the **Proof** tab, we look at the rule information given in the **Inner Node** pane, saying:

— KeY Output —————

```
impRight {
  \find ( ==> b -> c )
  \replacewith ( b ==> c )
  \heuristics ( alpha )
}
```

————— KeY Output ———

What we see here is what is called a *taclet*. Taclets are a framework for sequent calculi, a declarative language for programming the rules of a sequent calculus. The taclet framework was developed as part of the KeY project. The depicted taclet is the one which in the KeY system *defines* the rule **impRight**. In this chapter, we give just a hands-on explanation of the few taclets we come across. For a good introduction and discussion of the taclet framework, we refer to Chap. 4.

The taclet **impRight** captures what is expressed in the traditional sequent calculus style presentation of **impRight** we gave earlier, and a little more. The schema “ $b \rightarrow c$ ” in the `\find` clause indicates that the taclet is applicable to sequents if one of its formulae is an implication, with b and c being schema variables matching the two sub-formulae of the implication. Further down the **Inner Node** pane, we see that b and c are indeed of kind “`\formula`”:

— KeY Output —

```
\schemaVariables {
  \formula b;
  \formula c;
}
```

— KeY Output —

The sequent arrow “`==>`” in “`\find(==> b -> c)`” further restricts the applicability of the taclet to the *top-level*³ of the sequent only, and, in this case, to implications on the *right-hand side* of the sequent (as “ $b \rightarrow c$ ” appears right of “`==>`”). The `\replacewith` clause describes how to construct the *new* sequent from the current one: first the matching implication (here $p \ \& \ q \rightarrow q \ \& \ p$) gets deleted (“`replace-`”), and then the sub-formulae matching b and c (here $p \ \& \ q$ and $q \ \& \ p$) are added (“`-with`”) to the sequent. Which side of the sequent $p \ \& \ q$ resp. $q \ \& \ p$ are added to is indicated by the relative position of b and c w.r.t. “`==>`” in the argument of `\replacewith`. The result is the new sequent $p \ \& \ q \ ==> q \ \& \ p$. It is a very special case here that `\find(==> b -> c)` matches the whole old sequent, and `\replacewith(b ==> c)` matches the whole new sequent. Other formulae could appear in the old sequent. Those would remain unchanged in the new sequent. In other words, the Γ and Δ traditionally appearing in on-paper presentations of sequent rules are omitted in the taclet formalism. (Finally, with `\heuristics(alpha)` the taclet declares itself to be part of the *alpha* heuristics, which only matters for the execution of automated strategies.)

The discussed taclet is the complete definition of the **impRight** rule in KeY, and all the system knows about the rule. The complete list of available taclets can be viewed in the **Rules** tab *as part of the tabbed pane in the lower left corner, within the “Taclet Base” folder*. To test this, we click that folder and scroll down the list of taclets, until **impRight**, on which we can click to be shown the same taclet we have just discussed. It might feel scary to see the

³ Modulo leading updates, see Sect. 10.2.3.

sheer mass of taclets available. Please note, however, that the *vast* majority of taclets is never in the picture when *interactively* applying a rule in any practical usage of the KeY system. Instead, most taclets are only used by automated symbolic execution of the programs contained in formulae (see below).

Backtracking the Proof

So far, we performed only one little step in the proof. Our aim was, however, to introduce some very basic elements of the framework and system. In fact, we even go one step back, with the help of the system. For that, we make sure that the OPEN GOAL is in focus (by clicking on it in the **Proof** tab). We can then *undo* the proof step which led to this goal, by clicking at **Goal Back** in the task bar. This action will put us back in the situation we started in, which is confirmed by both the **Current Goal** pane and the **Proof** tab. Note that **Goal Back**, here and in general, only undoes *one* step each time.

Viewing and Customising Taclet Tooltips

Before performing the next steps in our proof, we take a closer look at the *tooltips* for rule selection. (The reader may already have noticed those tooltips earlier.) If we again click at the implication symbol \rightarrow appearing in the current goal, and *pre-select* the **impRight** rule in the opened context menu *simply by placing the mouse at impRight, without clicking yet*, we get to see a tooltip, displaying something similar, or identical, to the **impRight** taclet discussed above. The exact tooltip text depends on option settings which the user can configure. Depending on those settings, what is shown in the tooltip is *just the taclet as is*, or a certain '*significant*' part of it, in both cases either *with* or *without schema variables* already being *instantiated*. It would be unwise to commit, in this chapter, to the tooltip settings currently in place in the reader's KeY system. Instead, we control the options actively here, and discuss the respective outcome.

We open the tooltip options window by **View** \rightarrow **ToolTip options**, and make sure that all parts of taclets are displayed *by making sure the "pretty-print whole taclet ..." checkbox is checked*. For now, we disable the instantiation of schema variables *by setting the "Maximum size ... of tooltips ... with schema variable instantiations displayed ..." to 0*. With these settings, the tooltips for pre-selected rules consist of the original taclets, nothing more, nothing less. (The reader might try this with the already familiar **impRight** rule.) This is a good setting for getting familiar with the taclets as such. The *effect* of a taclet to the current proof situation is, however, better captured by tooltips where the schema variables from the `\find` argument are already instantiated by their respective matching formula or term. *We achieve this by setting the "Maximum size ... of tooltips ... with schema variable instantiations displayed ..." to some higher value, say 40*. When trying the tooltip for

impRight with this, we see something like the original taclet, however with **b** and **c** already being instantiated with $p \ \& \ q$ and $q \ \& \ p$, respectively:

— Tooltip —

```
impRight {
  \find ( ==> p & q -> q & p )
  \replacewith ( p & q ==> q & p )
  \heuristics ( alpha )
}
```

— Tooltip —

This “instantiated taclet”-tooltip tells us the following: If we clicked on the rule name, the formula $p \ \& \ q \rightarrow q \ \& \ p$, which we `\find` somewhere on the *right-hand side* of the sequent (see the formula’s relative position compared to `==>` in the `\find` argument), would be `\replace(d)with` the two formulae $p \ \& \ q$ and $q \ \& \ p$, where the former would be added to the *left-hand side*, and the latter to the *right-hand side* of the sequent (see their relative position compared to `==>` in the `\replacewith` argument). Note that, in this particular case, where the sequent only contains the matched formula, the arguments of `\find` and `\replacewith` which are displayed in the tooltip *happen to be* the *entire* old, resp., new sequent. This is not the case in general. The same tooltip would show up when preselecting **impRight** on the sequent: $r \ ==> p \ \& \ q \rightarrow q \ \& \ p, s$.

A closer look at the tooltip text in its current form (i.e., with the schema variables already being instantiated), reveals that the whole `\find` clause actually is redundant, as it is essentially identical with the anyhow highlighted text within the **Current Goal** pane. Also, the taclet’s name is already clear from the preselected rule name in the context menu. On top of that, the `\heuristics` clause is actually irrelevant for the *interactive* selection of the rule. The only non-redundant piece of information is therefore the `\replacewith` clause (in this case). Consequently, the tooltips can be reduced to the minimum which is relevant for supporting the selection of the appropriate rule *by un-checking* “pretty-print whole taclet ...” *option again*. The whole tooltip for **impRight** is the one-liner:

— Tooltip —

```
\replacewith ( p & q ==> q & p )
```

— Tooltip —

In general, the user might play around with different tooltip options in order to see which settings are most helpful. However, in the following steps, we assume these tooltips to show the full and unchanged taclet, so we switch back to our first setting *by checking* “pretty-print whole taclet ...” *and setting the* “Maximum size ... of tooltips ... with schema variable instantiations displayed ...” *to 0 again*.

Splitting Up the Proof

We apply **impRight** and consider the new goal $p \ \& \ q \implies q \ \& \ p$. For further decomposition we could break up the conjunctions on either sides of the sequent. By first selecting $q \ \& \ p$ on the right-hand side, we are offered the rule **andRight**, among others. The corresponding tooltip shows the following taclet:

— Tooltip —

```
andRight {
  \find ( ==> b & c )
  \replacewith ( ==> b );
  \replacewith ( ==> c )
  \heuristics ( beta, split )
}
```

— Tooltip —

Here we see *two* `\replacewith`s, telling us that this taclet will construct *two* new goals from the old one, meaning that this is a *branching* rule.⁴ Written as a sequent calculus rule, it looks like this:

$$\text{andRight} \frac{\Gamma \implies \phi, \Delta \quad \Gamma \implies \psi, \Delta}{\Gamma \implies \phi \ \& \ \psi, \Delta}$$

We now generalise the earlier description of the meaning of rules, to also cover branching rules. The *logical meaning* of a rule is downwards: if a certain instantiation of the rule’s schema variables makes *all* premisses valid, then the corresponding instantiation of the conclusion is valid as well. Accordingly, the *operational meaning* during proof construction goes upwards: The problem of proving a goal which matches the conclusion is reduced to the problem of proving *all* the (accordingly instantiated) premisses.

If we apply **andRight** in the system, the **Proof** tab shows the proof branching into two different **Cases**. In fact, both branches carry an OPEN GOAL. At least one of them is currently visible in the **Proof** tab, and highlighted *blue* to indicate that this is the new current goal, being detailed in the **Current Goal** pane as usual. The other OPEN GOAL might be hidden in the **Proof** tab (depending on the system settings), as the branches *not* leading to the current goal appear *collapsed* in the **Proof** tab by default. A collapsed/expanded branch can however be expanded/collapsed *by clicking on* \boxplus/\boxminus .⁵ If we expand the yet collapsed branch, we see the full structure of the proof, with

⁴ Note that it is not the “**split**” argument of the **heuristics** clause which makes this rule branching. The “**split**” is only the name of the heuristics this taclet claims to be member of. The fact that the taclet would branch a proof is the *reason* for (and not a *consequence* of) making it member of the **split** heuristics.

⁵ Bulk expansion, resp., bulk collapsing of all proof branches is offered by a context menu *via right click* in the **Proof** tab.

both OPEN GOALS being displayed. We can even switch the current goal by clicking on any of the OPEN GOALS.⁶

An on-paper presentation of the current proof would look like this:

$$\frac{\frac{p \ \& \ q \ \Rightarrow \ q \qquad p \ \& \ q \ \Rightarrow \ p}{p \ \& \ q \ \Rightarrow \ q \ \& \ p}}{\Rightarrow \ p \ \& \ q \ \rightarrow \ q \ \& \ p}$$

The reader might compare this presentation with the proof presented in the **Proof** tab by again clicking on the different nodes (or by clicking just anywhere within the **Proof** tab, and browsing the proof using the arrow keys).

Closing the First Branch

To continue, we put the OPEN GOAL $p \ \& \ q \ \Rightarrow \ q$ in focus again. Please recall that we want to reach a sequent where identical formulae appear on both sides (as such sequents are trivially true). We are already very close to that, just that $p \ \& \ q$ remains to be decomposed. Clicking at $\&$ offers the rule **andLeft**, as usual with the tooltip showing the taclet, here:

— Tooltip —

```
andLeft {
  \find ( b & c ==> )
  \replacewith ( b, c ==> )
  \heuristics ( alpha )
}
```

— Tooltip —

which corresponds to the sequent calculus rule:

$$\text{andLeft} \frac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \ \& \ \psi \Rightarrow \Delta}$$

We apply this rule, and arrive at the sequent $p, \ q \ \Rightarrow \ q$. We have arrived where we wanted to be, at a goal which is *trivially true* by the plain fact that one formula appears on both sides, *regardless* of how that formula looks like. (Of course, the sequents we were coming across in this example were all trivially true in an intuitive sense, but always only because of the particular form of the involved formulae.) In the sequent calculus, sequents of the form $\Gamma, \phi \Rightarrow \phi, \Delta$ are considered valid *without any need of further reduction*. This argument is also represented by a rule, namely:

$$\text{closeGoal} \frac{}{\Gamma, \phi \Rightarrow \phi, \Delta}$$

⁶ Another way of getting an overview over the open goals, and switch the current goal, is offered by the **Goals** tab.

In general, rules with no premiss *close* the branch leading to the goal they are applied to, or, as we say in short (and a little imprecise), *close the goal* they are applied to.

The representation of this rule as a taclet calls for two new keywords which we have not seen so far. One is `\closegoal`, having the effect that taclet application does not produce any new goal, but instead closes the current proof branch. The other keyword is `\assumes`, which is meant for expressing assumptions on formulae *other than* the one matching the `\find` clause. Note that, so far, the applicability of rules always depended on *one* formula only. The applicability of `closeGoal` however depends on *two* formulae (or, more precisely, on two formula occurrences). The second formula is taken care of by the `\assumes` clause in the `closeGoal` taclet:

— Taclet —

```
closeGoal {
  \assumes ( b ==> )
  \find ( ==> b )
  \closegoal
  \heuristics ( closure )
}
```

— Taclet —

Note that this taclet is not symmetric (as opposed to the `closeGoal` sequent rule given above). To apply it interactively on our **Current Goal** $p, q \implies q$, we have to put the *right-hand side* q into focus (cf. “`\find(==> b)`”). But the `\assumes` clause makes a taclet applicable only in the presence of further formulas, in this case the identical formula on the *left-hand side* (cf. “`\assumes(b ==>)`”).⁷

This discussion of the `closeGoal` sequent rule and a corresponding `closeGoal` taclet shows that taclets are more fine grained than rules. They contain more information, and consequently there is more than one way to represent a sequent rule as a taclet. To see another way of representing the above *sequent rule* `closeGoal` by a taclet, the reader might click on the q on the *left-hand side* of $p, q \implies q$, and pre-select the taclet `closeGoalAntec`. The tooltip will show the taclet:

⁷ This is not the whole truth. In KeY, one can even enforce the application of taclets with the `assumes` clause not *syntactically* satisfied by the sequent. In that case, an additional branch is created, allowing us to *prove* the assumptions not yet present in the sequent. We may, however, ignore that possibility for the time being. Moreover, in the case of closing rules, the usage of this feature is particularly useless, as it leads to a loop in the proof.

 — Tooltip —

```
closeGoalAntec {
  \assumes ( ==> b )
  \find ( b ==> )
  \closegoal
}
```

 — Tooltip —

We, however, proceed by applying the taclet **closeGoal** on the right-hand side formula q . (With the current settings, the taclet application should happen instantly. In case there opens a dialogue, the reader might select **Cancel**, check the **Minimize interaction** option, and re-apply **closeGoal**.) After this step, the **Proof** pane tells us that the proof branch that has just been under consideration is closed, which is indicated by that branch ending with a “Closed goal” node *coloured green*. The system has automatically changed focus to the next OPEN GOAL, which is detailed in the **Current Goal** pane as the sequent $p \ \& \ q \ ==> \ p$.

Pruning the Proof Tree

We apply **andLeft** to the $\&$ on the left, in the same fashion as we did on the other branch. Afterwards, we *could* close the new goal $p, q \ ==> \ p$, but we refrain from doing so. Instead, we compare the two branches, the closed and the open one, which both carry a node labelled with **andLeft**. When inspecting these two nodes again (by simply clicking on them), we see that we broke up the same formula, the left-hand side formula “ $p \ \& \ q$ ”, on both branches. It appears that we branched the proof too early. Instead, we should have applied the (non-branching) **andLeft**, once and for all, before the (branching) **andRight**. *This is a good strategy in general, to delay proof branching as much as possible, thereby avoiding double work on the different branches.* Without this strategy, more realistic examples with hundreds or thousands of proof steps would become completely infeasible.

In our tiny example here, it seems not to matter much, but it is instructive to apply the late splitting also here. We want to re-do the proof from the point where we split too early. Instead of re-loading the problem file, we can *prune* the proof at the node labelled with **andRight** by *right-click on that node, and selecting Prune Proof*. As a result, large parts of the proof are pruned away, and the second node, with the sequent $p \ \& \ q \ ==> \ q \ \& \ p$, becomes the **Current Goal** again.

Closing the First Proof

This time, we apply **andLeft** *before* we split the proof via **andRight**. The two remaining goals, $p, q \ ==> \ q$ and $p, q \ ==> \ p$, we close by applying **closeGoal** to the right-hand q and p , respectively. By closing all branches, we have


actually closed the entire proof, as we can see from the **Proof closed** window popping up now.

Altogether, we have proved the validity of the *sequent* at the root of the proof tree, here $\implies p \ \& \ q \ \rightarrow \ q \ \& \ p$. As this sequent has only one formula, placed on the right-hand side, we have actually proved validity of that *formula* $p \ \& \ q \ \rightarrow \ q \ \& \ p$, the one stated as the `\problem` in the file we loaded.


Proving the Same Formula Automatically

As noted earlier, the reason for doing all the steps in the above proof manually was that we wanted to learn about the system and the used artefacts. Of course, one would otherwise prove such a formula automatically, which is what we do in the following.

Before loading the same problem again, we can choose whether we abandon the current proof, or alternatively keep it in the system. Abandoning a proof would be achieved via the menu: **Proof** \rightarrow **Abandon Task**. It is however possible to keep several (finished or unfinished) proofs in the system, so we suggest to start the new proof while keeping the old one. This will allow us to compare the proofs more easily.

Loading the file `andCommutes.key` again can be done in the same fashion as before *or alternatively via the “Load last opened file.” button*  *in the toolbar*. The system might then ask whether the problem should be opened in a new environment or in the already existing one. We choose **Open in new environment** (and do so also in the following, unless stated otherwise). The system might further ask whether the previous proof should be marked for reuse. We **Cancel** that dialog here (but refer to Chap. 13 on the topic of *proof reuse*). Afterwards, we see a second ‘*task*’ being displayed in the **Task** pane. One can even switch between the different tasks by clicking in that pane.

The newly opened proof shows the **Current Goal** $\implies p \ \& \ q \ \rightarrow \ q \ \& \ p$, just as last time. In order to let KeY prove this automatically, we first have to select a *proof search strategy*, which is done in the **Proof Search Strategy** tab. The most important strategy offered there is the strategy for JAVA dynamic logic **Java DL**, with its variations for loop/method treatment. However, our sequent here does not contain programs, and therefore falls in the pure first-order fragment of the logic. (Here, it is even only propositional.) Therefore, the strategy for pure first-order logic **FOL** is appropriate, and we select that. (The slider controlling the maximal number of automatic rule applications should be at least **1000**, which will suffice for all examples in this chapter).

By pressing the “automated proof search” button , we start running the chosen strategy. A complete proof is constructed immediately. Its shape (see **Proof** tab) depends heavily on the current implementation of the FOL strategy. However, it is most likely different from the proof we constructed interactively before. (For a comparison, we switch between the tasks in the **Task** pane.)

Rewrite Rules

With *the current implementation of* the FOL strategy, only the first steps of the automatically constructed proof, **impRight** and **andLeft**, are identical with the interactively constructed proof from above, leading to the sequent $p, q \implies q \ \& \ p$. After that, the proof does *not* branch, but instead uses the rule **replaceKnownLeft**:

Tactlet


```
replaceKnownLeft {
  \assumes ( b ==> )
  \find ( b )
  \sameUpdateLevel
  \replacewith ( true )
  \heuristics ( replace_known )
}
```

Tactlet

It has the effect that any formula (`\find(b)`) which has *another appearance* on the left side of the sequent (`\assumes(b ==>)`) can be replaced by **true**. Note that the `\find` clause does not contain “`==>`”, and therefore does not specify where the formula to be replaced shall appear. However, only one formula at a time gets replaced.

Tactlets with a “`==>`”-free `\find` clause are called *rewrite tactlets* or *rewrite rules*. The argument of `\find` is a schema variable of kind `\formula` or `\term`, matching formulae resp. terms at *arbitrary* positions, which may even be nested. (The position can be further restricted. The restriction `\sameUpdateLevel` in this tactlet is however not relevant for the current example.) When we look at how the tactlet was used in our proof, we see that indeed the *sub-formula* q of the formula $q \ \& \ p$ has been rewritten to **true**, resulting in the sequent $p, q \implies \mathbf{true} \ \& \ p$. The following rule application simplifies the **true** away, after which **closeGoal** is applicable again.

Saving a Proof

Before we leave the discussion of the current example, we *save* the just accomplished proof (admittedly for no other reason than practising the saving of proofs). For that, we select **File** \rightarrow **Save ... or alternatively the “Save current proof.” button**  *in the toolbar*. The opened file browser dialogue allows to locate and name the proof file. A sensible name would be `andCommutes.proof`, but any name would do, *as long as the file extension is “.proof”*. It is completely legal for a proof file to have a different naming than the corresponding problem file. This way, it is possible to save several proofs for the same problem.

Proofs can actually be saved at any time, regardless of whether they are finished or not. An unfinished proof can be continued when loaded again.

10.2.2 Exploring Terms, Quantification, and Instantiation: Building First-Order Proofs

After having looked at the basic usage of the KeY prover, we want to extend the discussion to more advanced features of the logic. The example of the previous section did only use propositional connectives. Here, we discuss the basic handling of first-order formulae, containing terms, variables, quantifiers, and equality. As an example, we prove a `\problem` which we load from the file `projection.key`:

— KeY Problem File —

```

\sorts {
    s;
}
\functions {
    s f(s);
    s a;
}
\problem {
    ( \forall s x; f(f(x)) = f(x) ) -> f(a) = f(f(f(a)))
}

```

— KeY Problem File —

The file first declares a function `f` (of type `s → s`) and a constant `a` (of sort `s`). The first part of the `\problem` formula, `\forall s x; f(f(x)) = f(x)`, says that `f` is a *projection*: For all `x`, applying `f` twice is the same as applying `f` once. The whole `\problem` formula then states that `f(a)` and `f(f(f(a)))` are equal, given `f` is a projection.

Instantiating Quantified Formulae

We prove this simple formula interactively, for now. After loading the problem file, and applying `impRight` to the initial sequent, the **Current Goal** is: `\forall s x; f(f(x)) = f(x) ==> f(a) = f(f(f(a)))`.

We proceed by deriving an additional assumption (i.e., left-hand side formula) `f(f(a)) = f(a)`, by instantiating `x` with `a`. For the interactive instantiation of quantifiers, KeY supports *drag and drop* of terms over quantifiers (whenever the instantiation is textually present in the current sequent). In the situation at hand, we can drag any of the two “a” onto the quantifier `\forall` by clicking at “a”, holding and moving the mouse, to release it over the “forall”. As a result of this action, the new **Current Goal** features the *additional* assumption `f(f(a)) = f(a)`.

There is something special to this proof step: Though it was triggered interactively, we have not been specific about which taclet to apply. The **Proof** pane, however, tells us that we just applied the taclet `instAll`. To see the very taclet, we can click at the previous proof node, marked with `instAll`, such that the **Inner Node** pane displays:

— KeY Output —

```

instAll {
  \assumes ( \forall u; b ==> )
  \find ( t )
  \add ( {\subst u; t}b ==> )
}

```

— KeY Output —

“ $\{\text{subst } u; t\}b$ ” means that (the match of) u is substituted by (the match of) t in (the match of) b , during taclet application.

Making Use of Equations

Now we can use the new equation $f(f(a)) = f(a)$ to simplify the term $f(f(f(a)))$, meaning we *apply* the equation to the $f(f(a))$ subterm of $f(f(f(a)))$. This action can again be performed via drag and drop, here by dragging the equation on the left side of the sequent, and dropping it over the $f(f(a))$ subterm of $f(f(f(a)))$.⁸ *In the current system, there opens a context menu, allowing to select either of two taclets with the identical display name **applyEquality**. The taclets are however different, see their tooltips. For our example, it does not matter which one is selected.*

Afterwards, the right-hand side equation has changed to $f(a) = f(f(a))$, which looks almost like the left-hand side equation. We can proceed either by swapping one equation, or by again applying the left-hand side equation on a right-hand side term. It is instructive to discuss both alternatives here.

First, we select $f(a) = f(f(a))$, and apply **commuteEq**. The resulting goal has two identical formulae on both sides of the sequent, so we *could* apply **closeGoal**. But instead, just to demonstrate the other possibility as well, we backtrack (via **Goal Back**), leading us back to the **Current Goal** $f(f(a)) = f(a), \dots ==> f(a) = f(f(a))$.

The other option is to apply the left-hand equation to $f(f(a))$ on the right (via drag and drop). Afterwards, we have the *tautology* $f(a) = f(a)$ on the right. By preselecting that formula, we get offered the taclet **closeEq**, which transforms the equation into **true**.

Closing “by True” and “by False”

So far, all goals we ever closed featured identical formulae on both sides of the sequent. We have arrived at the second type of closable sequents: one with **true** on the *right* side. We close it by highlighting **true**, and selecting the taclet **closeByTrue**, which is defined as:

⁸ More detailed, we move the mouse over the “=” symbol, such that the whole of $f(f(a)) = f(a)$ is highlighted. We click, hold, and move the mouse, over the second “f” in $f(f(f(a)))$, such that exactly the subterm $f(f(a))$ gets highlighted. Then, we release the mouse.

```

— Taclet —
closeByTrue {
  \find ( ==> true )
  \closegoal
  \heuristics ( closure )
}

```

Taclet —

This finishes our proof.

Without giving an example, we mention here the third type of closable sequents, namely those with **false** on the *left* side, to be closed by:

```

— Taclet —
closeByFalse {
  \find ( false ==> )
  \closegoal
  \heuristics ( closure )
}

```

Taclet —

This is actually a very important type of closable sequent. In many examples, a sequent can be proved by showing that the assumptions (i.e., the left-hand side formulae) are contradictory, meaning that **false** can be derived on the left side.

Using Taclet Instantiation Dialogues

In our previous proof, we used the “drag-and-drop-directly-in-goal” feature offered by the KeY prover. This kind of user interaction can be seen as a shortcut to another kind of user interaction: the usage of *taclet instantiation dialogues*. While the former is most convenient, the latter is more general and should be familiar to each KeY user. Therefore, we re-construct the (in spirit) same proof, this time using such a dialogue explicitly.

After again loading the problem file `projection.key` (and **Cancelling** the **re-use** dialogue), we apply **impRight** to the initial sequent, just like before. Next, to instantiate the quantified formula `\forall s x; f(f(x)) = f(x)`, we highlight that formula, and apply the taclet **allLeft**, which is defined as:

```

— Taclet —
allLeft {
  \find ( \forall u; b ==> )
  \add ( {\subst u; t}b ==> )
  \heuristics ( gamma )
}

```

Taclet —

This opens a **Choose Taclet Instantiation** dialogue, allowing the user to choose the (not yet determined) instantiations of the taclet’s schema variables. The taclet at hand has three schema variables, *b*, *u*, and *t*. The instantiations of *b* and *u* are already determined to be $f(f(x)) = f(x)$ and *x*, just by matching the highlighted sequent formula $\forall s\ x; f(f(x)) = f(x)$ with the $\backslash\text{find}$ argument $\forall\text{forall}\ u; b$. The instantiation of *t* is, however, left open, to be chosen by the user. We can type “*a*” in the corresponding input field of the dialogue,⁹ and click **Apply**. As a result, the $f(f(a)) = f(a)$ is added to the left side of the sequent. The rest of the proof goes exactly as discussed before. The reader may finish it herself.

Loading a Proof

We want to compare the proof which we just have constructed interactively with a proof the FOL strategy would construct *automatically*. The user could load the same $\backslash\text{problem}$ again, and run the FOL strategy. However, to not make the discussion too dependent on the current implementation of the FOL strategy, we instead *load a proof* which was automatically constructed, and saved, at the time this chapter was written.

The loading of a proof is done in exactly the same way as loading a problem file, with the only difference that a *.proof* file is selected (instead of a *.key* file). We load the proof `projectionAutomat.proof`.

Discovering Meta Variables and Constraints

We inspect the loaded proof. The sequent of the first inner node (labelled with “1:”) tells us that this is actually a proof of the same problem as before. (The name of the proof file is just an indication, nothing more.) We can see that the FOL strategy decided to, as a first step, reorient the equation $f(a) = f(f(f(a)))$. Afterwards, nothing special is happening until node “3:”. To the sequent of that node, the FOL strategy applied **allLeft**, as we did in our previous proof. However, the resulting goal (number “4:”) looks different:

```

— KeY Output —————
f(f(X_0)) = f(X_0),
\forall s x; f(f(x)) = f(x)
==>
f(f(f(a))) = f(a)
————— KeY Output ———

```

⁹ Alternatively, one can also drag and drop syntactic entities from the **Current Goal** pane into the input fields of such a dialogue, and possibly edit them afterwards. This is not attractive in the current example, but becomes essential in other cases, for instance when generalising induction hypotheses, see Chap. 11.

Note that the formula newly introduced by **allLeft** is $f(f(X_0)) = f(X_0)$ (and not, like in the previous proof, $f(f(a)) = f(a)$). Not only has the FOL strategy chosen an instantiation (of the schematic taclet variable τ) which is different from our previous choice. The fact that the instantiation “ X_0 ” starts with a *capital letter* tells us that this is a *meta variable*, which intuitively stands for a term yet to be determined. Before we discuss this new concept a bit more, we check out the next node. The sequent of node number “5:” contains the *constrained formula* $f(f(a)) = f(a) \ll [X_0 = a]$, which intuitively says something like: “The (unconstrained) formula $f(f(a)) = f(a)$ is only present really if the constraint $X_0 = a$ is fulfilled, otherwise we imagine it not being there.”

Meta variables, and constraints over meta variables, are concepts which serve the purpose of proof *automation*. They are less important for (purely) *interactive* proving. The KeY project, however, follows an integrated approach, where automated and interactive proof steps are intertwined. Therefore, a KeY user should at least have a rudimentary idea of what meta variables and constraints are all about.

Understanding Meta Variables and Constraints (to a Certain Extent)

Instantiation of quantified variables is a crucial task, and typically more difficult than in the example at hand. For automated strategies it is often infeasible to guess the right instance at the point of quantifier instantiation (like when applying **allLeft**). To help solving this problem, the automated theorem proving community has invented the notion of meta variables¹⁰. These variables are not part of the actual logic under consideration. Rather, they are employed on the meta level, used as place-holders (for concrete terms) in the proof.

Meta variables allow to *delay* the guessing of *concrete instances*. During proof construction, they are introduced as *generic instances*, to be refined later on with the help of constraints¹¹. A typical point when to refine a meta variable is for instance a situation where a concrete instance *would* allow to close the current goal.

A thorough understanding of meta variable constraint handling goes beyond the scope of this chapter (\Rightarrow Sect. 4.3), and luckily is *not needed* for a user in order to effectively use the KeY prover. Some rudimentary understanding is however helpful, and we try to provide that here by reconstructing the current proof interactively, in a kind of slow motion picture of the fully automatic proof construction.

¹⁰ This kind of variables are known in the tableau-style theorem proving community under the name of “free variables”, see Fitting [1996].

¹¹ Historically, refining meta variables was not done with the help of constraints, but via destructive substitution. The usage of constraints for this purpose was invented within the KeY project, by Giese [2001], mainly to achieve a backtracking free calculus.

For this, we again load `projection.key` (while keeping the proof from `projectionAutomat.proof` in the system, for comparison). First, we apply `commuteEq` on $f(a) = f(f(f(a)))$, and afterwards `impRight`. Selecting `al-Left` on the quantified formula opens the **Choose Taclet Instantiation** dialogue, just as before. This time, we point the reader to the message pane of that dialogue, telling us that the “Instantiation is OK”, already. This might come as a surprise, as the instantiation of t is still left open. However, the proof system here allows the user to *not* commit to a concrete instantiation for t , in which case the system will instantiate t with a new meta variable, in the current implementation with “ X_0 ”. This is exactly the effect of clicking **Apply** now (while leaving the `Instantiation` field for t empty).

As a result, the new equation $f(f(X_0)) = f(X_0)$ is added on the left-hand side. Next, we would like to use that equation, to rewrite the $f(f(a))$ sub-term of right-hand side formula $f(f(f(a))) = f(a)$. It is intuitively clear that this is only possible *if* X_0 is equal to a . In the calculus, this “if” is reflected in the following way: *Rewriting $f(f(a))$ in $f(f(f(a))) = f(a)$ with $f(f(X_0)) = f(X_0)$ results in $f(f(a)) = f(a) \ll [X_0 = a]$.*

This is not the whole truth yet, as we can see when performing this step in the system. We drag the equation $f(f(X_0)) = f(X_0)$, and drop it over $f(f(a))$. In the resulting sequent, we can see that the original formula $f(f(f(a))) = f(a)$ was kept, in addition to the rewritten, constrained formula. By not throwing away this formula, we still cover the case where X_0 is *not* equal to a .

Now we can close our proof, by applying `closeGoal` on $f(f(a)) = f(a) \ll [X_0 = a]$. On the surface, this will immediately finish our proof. Internally, both $f(f(X_0)) = f(X_0)$ and $f(f(a)) = f(a) \ll [X_0 = a]$ need to serve as instantiation for the single schema variable b of the taclet `closeGoal`, and therefore need to be identified. What happens is that the application of the taclet first matches the two formulae, i.e., it rewrites $f(f(X_0)) = f(X_0)$ to $f(f(a)) = f(a) \ll [X_0 = a]$, and actually closes the goal afterwards.

Please recall that the creation, manipulation, and usage of meta variables and their constraints is entirely done by the system, not by the user. Therefore, the purpose of the above explanations is mainly to allow the user to interact on proof goals which were constructed automatically. (For some more discussion, we refer to Sect. 4.3.)

As a general advice, when instantiating quantifiers interactively, we recommend to use *concrete* instances instead of meta variables whenever the user is clear about which instance is needed for the current proof.

Skolemising Quantified Formulae

We will now consider a slight generalisation of the theorem we have just proved. Again, we assume that f is a projection. But instead of showing $f(a) = f(f(f(a)))$, for a particular a , we show $f(y) = f(f(f(y)))$ for

all y . For this we load `generalProjection.key`, and apply `impRight`, which results in the sequent:

— KeY Output —

```
\forallall s x; f(f(x)) = f(x) ==> \forallall s y; f(y) = f(f(f(y)))
```

— KeY Output —

As in the previous proof, we will have to instantiate the quantified formula on the left. But this time we also have to deal with the quantifier on the right. Luckily, that quantifier can be eliminated altogether, by applying the rule `allRight`, which results in:¹²

— KeY Output —

```
\forallall s x; f(f(x)) = f(x) ==> f(y_0) = f(f(f(y_0)))
```

— KeY Output —

We see that the quantifier disappeared, and the variable y got replaced. The replacement, `y_0`, is a *constant*, which we can see from the fact that `y_0` is not quantified. Note that in our logic each logical variable appears in the scope of a quantifier binding it.¹³ Therefore, `y_0` can be nothing but a constant. Moreover, `y_0` is a *new* symbol.

Eliminating quantifiers by introducing new constants is called *skolemisation* (after the logician Thoralf Skolem). In a sequent calculus, universal quantifiers (`\forallall`) on the right, and existential quantifiers (`\exists`) on the left side, can be eliminated this way, leading to sequents which are equivalent (concerning provability), but simpler. This should not be confused with quantifier *instantiation*, which applies to the complementary cases: (`\exists`) on the right, and (`\forallall`) on the left, see our discussion of `allLeft` above. (It is instructive to look at all four cases in combination, see Sect. 2.5.4, Chapt. 2.)

Skolemisation is a simple proof step, and is normally done fully automatically. We only discuss it here to give the user some understanding about new constants (or functions, see below) that might show up during proving.

To see the taclet we have just applied, we select the inner node labelled with `allRight`. The **Inner Node** pane reveals the taclet:

— KeY Output —

```
allRight {
  \find ( ==> \forallall u; b )
  \varcond ( \new(sk, \dependingOn(b)) )
  \replacewith ( ==> {\subst u; sk}b )
  \heuristics ( delta )
}
```

— KeY Output —

¹² Note that the particular name `y_0` can differ, depending on the implementation.

¹³ This is not the case for *meta* variables, as they are not logical variables.

It tells us that the rule removes the quantifier matching `\forall` `u`; and that (the match of) `u` is `\substituted` by (the match of) `sk` in the remaining formula (matching `b`). During application of this taclet, the schema variable `sk` will be instantiated with a Skolem term, in many cases a Skolem constant only. The instantiation of `sk` is restricted by the schema variable condition `\varcond`: First of all, the instantiation of `sk` must be a `\new` term. Second, the particular instantiation of `sk` is determined `\dependingOn` the (match of) `b`.

Two things remain to be explained here. Why are Skolem *constants* not always sufficient, and how do proper Skolem *terms* depend on the formula at hand? Both issues are related to the potential presence of “meta variables” in a sequent. Without that, new constants would indeed be sufficient. But in the presence of meta variables, newly introduced constants could later be identified with “older” meta variables, leading to unsound reasoning. This is the reason why Skolem *terms* are used in the general case. Those terms are of the form $f(X_1, \dots, X_n)$, with f being a new function symbol, and X_1, \dots, X_n being the meta variables appearing in the formula this term is `\dependingOn`.¹⁴ Here, we do not give an example where proper Skolem *terms* appear. However, these explanations should help the user to not feel uncomfortable when confronted with automatically introduced Skolem constants/functions/terms.

The rest of our current proof goes exactly like for the previous problem formula. Instead of further discussing it here, we simply run the “FOL” strategy to resume the proof.

Employing External Decision Procedures

Apart from strategies, which apply taclets automatically, KeY also employs external decision procedure tools for increasing the automation of proofs. The field of decision procedures is very dynamic, and so is the way in which KeY makes use of them. The user can choose among the available decision procedure tools under **Options** → **Decision Procedure Config**. With **Simplify** selected there, we load `generalProjection.key` once more, and push the **Run Simplify** button in the tool bar. This closes the proof in one step(!), as the **Proof** tab is telling us. Decision procedures can be very efficient on certain problems. On the down side, we sacrificed proof transparency here.

In a more realistic setting, we use decision procedures towards the end of a proof (branch), to close first-order goals which emerged from proving problems that go beyond the scope of decision procedures.

¹⁴ Note that a (finite) term is always syntactically different from its proper sub-terms. Therefore, the newly introduced term $f(X_1, \dots, X_n)$ can never be instantiated in a way that makes it syntactically equal to either of X_1, \dots, X_n . This property is actually the sole purpose of the form $f(X_1, \dots, X_n)$. The fact that *only* the meta variable of the quantified formula, not those of the whole sequent, are needed goes back to a result by Hähnle and Schmitt [1994].

10.2.3 Exploring Programs in Formulae: Building Dynamic Logic Proofs

Not first-order logic, and certainly not propositional logic, is the real target of the KeY prover. Instead, the prover is designed to handle proof obligations formulated in a substantial extension of first-order logic, *dynamic logic* (DL). What is dynamic about this logic is the notion of the world, i.e., the interpretation (of function/predicate symbols) in which formulae (and sub-formulae) are evaluated. In particular, a formula and its sub-formulae can be interpreted in *different* worlds.

The other distinguished feature of DL is that descriptions of how to construct one world from another are explicit in the logic, in the form of *programs*. Accordingly, the worlds represent computation *states*. (In the following, we take “state” as a synonym for “world”.) This allows us to, for instance, talk about the states both *before* and *after* executing a certain program, *within the same formula*.

Compared to first-order logic, DL employs two additional (mix-fix) operators: $\langle . \rangle$. (diamond) and $[.]$. (box). In both cases, the first argument is a *program*, whereas the second argument is another DL formula. With $\langle p \rangle \varphi$ and $[p] \varphi$ being DL formulae, $\langle p \rangle$ and $[p]$ are called the *modalities* of the respective formula.

A formula $\langle p \rangle \varphi$ is valid in a state if, from there, an execution of p terminates normally and results in a state where φ is valid. As for the other operator, a formula $[p] \varphi$ is valid in a state from where execution of p does *either* not terminate normally *or* results in a state where φ is valid.¹⁵ For our applications the diamond operator is way more important than the box operator, so we restrict attention to that.

One frequent pattern of DL formulae is “ $\varphi \rightarrow \langle p \rangle \psi$ ”, stating that the program p , when started from a state where φ is valid, terminates, with ψ being valid in the post state. (Here, φ and ψ often are pure first-order formulae, but they can very well be proper DL formulae, containing programs themselves.)

Each variant of DL has to commit to a formalism used to describe the programs (i.e., the p) in the modalities. Unlike most other variants of DL, the KeY project’s DL variant employs a real programming language, namely JAVA CARD. Concretely, p is a sequence of (zero, one, or more) JAVA CARD statements. Accordingly, the logic is called JAVA CARD DL.

The following is an example of a JAVA CARD DL formula:

$$x < y \rightarrow \langle \text{int } t = x; x = y; y = t; \rangle y < x \quad (10.1)$$

It says that in each state where the program variable x has a value smaller than that of the program variable y , the sequence of JAVA statements

¹⁵ These descriptions have to be generalised when indeterministic programs are considered, which is not the case here.

“`int t = x; x = y; y = t;`” terminates, and afterwards the value of `y` is smaller than that of `x`. It is important to note that `x` and `y` are *program* variables, not to be confused with *logical* variables. In our logic, there is a strict distinction between both. Logical variables must appear in the scope of a quantifier binding them, whereas program variables cannot be quantified over. The formula (10.1) has no quantifier because it does not contain any logical variables.

As we will see in the following examples, both program variables and logical variables can appear mixed in terms and formulae, also together with logical constants, functions, and predicate symbols. However, inside the modalities, there can be nothing but (sequents of) *pure* JAVA statements.

For a more thorough discussion of JAVA CARD DL, please refer to Chap. 3.

Feeding the Prover with a DL Problem File

The file `exchange.key` contains the JAVA CARD DL formula (10.1), in the concrete syntax used in the KeY system:¹⁶

— KeY Problem File —

```
\programVariables { int x, y; }
\problem {
    x < y
    -> \<{
        int t = x;
        x=y;
        y=t;
    }\> y < x
}
```

— KeY Problem File —

When comparing this syntax with the notation used in (10.1), we see that diamond modality brackets “`<`” and “`>`” are written as “`\<`” and “`\>`” within the KeY system. (In future versions, “`{`” and “`}`” might become obsolete here, such that “`<`” and “`>`” would suffice.) What we can also observe from the file is that all program variables which are *not* declared in the JAVA code inside the modality (like “`t`” here) must appear within a `\programVariables` declaration of the file (like “`x`” and “`y`” here).

Instead of loading this file, and proving the problem, we try out other examples first, which are meant to slowly introduce the principles of proving JAVA CARD DL formulae with KeY.

Using the Prover as an Interpreter

We consider the file `executeByProving.key`:

¹⁶ Here as in all `.key` files, line breaks and indentation do not matter other than supporting readability.

— KeY Problem File —


```

\predicates { p(int,int); }
\programVariables { int i, j; }
\problem {
  \<{ i=2;
      j=(i=i+1)+4;
  }\> p(i,j)
}

```

— KeY Problem File —

As the reader might guess, the `\problem` formula is not valid, as there are no assumptions made about the predicate `p`. Anyhow, we let the system *try* to prove this formula. By doing so, we will see that the KeY prover will essentially *execute* our (rather obscure) program “`i=2; j=(i=i+1)+4;`”, which is possible because all values the program deals with are *concrete*. The execution of JAVA programs is of course not the purpose of the KeY prover, but it serves us here as a first step towards the method for handling symbolic values, *symbolic execution*, to be discussed later.

We load the file `executeByProving.key` into the system. Then, we run the automated JAVA CARD DL strategy (by clicking the play button  with the **Java DL** strategy selected in the **Proof Search Strategy** tab). The strategy stops with `==> p(3,7)` being the (only) OPEN GOAL, see also the **Proof** tab. This means that the proof *could* be closed *if* `p(3,7)` was provable, which it is not. But that is fine, because all we wanted is letting the KeY system compute the values of `i` and `j` after execution of “`i=2; j=(i=i+1)+4;`”. And indeed, the fact that proving `p(3,7)` would be sufficient to prove the original formula tells us that that 3 and 7 are the final values of `i` and `j`.

We now want to inspect the (unfinished) proof itself. For this, we select the first inner node, labelled with number “1.”, which contains the original sequent. By using the down-arrow key, we can scroll down the proof. The reader is encouraged to do so, before reading on, all the way down to the OPEN GOAL, to get an impression on how the calculus executes the JAVA statements at hand. This way, one can observe that one of the main principles in building a proof for a DL formula is to perform *program transformation* within the modality(s). In the current example, the complex second assignment `j=(i=i+1)+4;` was transformed into a sequence of simpler assignments. Once a leading assignment is simple enough, it moves out from the modality, into other parts of the formula (see below). This process continues until the modality is empty (“`\<{\}\>`”). That empty modality gets eventually removed by the taclet **emptyModality**.

Discovering Updates

Our next observation is that the formulae which appear in inner nodes of this proof contain a syntactical element which is not yet covered by the

above explanations of DL. We see that already in the second inner node (number "2:."), which *in the current implementation* looks like:

— KeY Output —

```

==>
  {i:=2}
  \<{
    j=(i=i+1)+4;
  }\> p(i,j)

```

— KeY Output —

The “`i:=2`” within the curly brackets is an example of what is called “*updates*”. When scrolling down the proof, we can see that leading assignments turn into updates when they move out from the modality. The updates somehow accumulate, and are simplified, in front of a “shrinking” modality. Finally, they get applied to the remaining formula once the modality is gone.

Understanding Updates (to a Certain Extent)

Updates are part of the JAVA CARD DL invented within the KeY project. Their main intention is to represent the *effect* of some JAVA code they replace. This effect can be accumulated, manipulated, simplified, and applied to other parts of the formula, in a way which is (to a certain extent) disentangled from the manipulation of the program in the modality. This allows a separation of concerns which has been fruitful for the design and usage of the calculus and the automated strategies.¹⁷

Elementary updates in essence are a restricted kind of JAVA assignment, where the right-hand side must be a *simple* expression, which in particular is *free of side effects*. Examples are “`i:=2`”, or “`i:=i + 1`” (which we find further down in the proof). From elementary updates, more complex updates can be constructed (see Def. 3.8, Chap. 3). Here, we only mention the most important kind of compound updates, *parallel updates*, an example of which is “`i:=3 || j:=7`” further down in the proof.

Updates extend traditional DL in the following way: if φ is a DL formula and u is an update, then $\{u\}\varphi$ is also a DL formula. Note that this definition is recursive, such that φ in turn may have the form $\{u'\}\varphi'$, in which case the whole formula looks like $\{u\}\{u'\}\varphi'$. (The strategies try to transform such subsequent updates into one, parallel update.) As a special case, φ may not contain any modality (i.e., it is purely first-order). This situation occurs in the current proof in form of the sequent `==> {i:=3 || j:=7}p(i,j)` (close to the OPEN GOAL). Once the modality is gone, the update is *applied*, in the form of a *substitution*, to the (now only first-order) formula following the update, as the reader can see when scrolling the proof. Altogether, this leads

¹⁷ In the presence of pointers, like the object references in JAVA, the concept of updates serves as an alternative to having an explicit heap *as data* in the logic.

to a delayed turning of program assignments in into substitutions in the logic, as compared to other variants of DL (or of Hoare logic).

In this sense, we can say that updates are *lazily* applied. On the other hand, they are *eagerly* simplified, as we will see in the following (intentionally primitive) example. For that, we load the file `updates.key`. Then, the initial **Current Goal** looks like this:

— KeY Output —

```
==>
\<{
  i=1;
  j=3;
  i=2;
}\> i = 2
```

— KeY Output —

We prove this sequent interactively (twice even), just to get a better understanding of the basic steps usually performed by automated strategies. In the first round, we focus on the role of updates in the proof, whereas the discussion of the used taclets is postponed to the second round (see below).

The first assignment, `i=1;`, is simple enough to be moved out from the modality, into an update. We can perform this step by pointing on that assignment, and applying the **assignment** rule. In the resulting sequent, that assignment got removed and the update $\{i:=1\}$ ¹⁸ appeared in front. We perform the same step on the leading assignment `j=3;`. Afterwards, and surprisingly, the **Current Goal** does *not* contain the corresponding update, $\{j:=3\}$. But a closer look on the **Proof** pane shows that the prover actually performed two steps. After the first, we actually had the two subsequent updates $\{i:=1\}\{j:=3\}$ (as we can see when selecting the corresponding inner node). However, on this goal the prover called the built in *update simplifier*, automatically. That update simplifier is a very powerful proof rule, and one of the few rules which are not represented by a taclet. In this case, the update simplifier detected that the update $\{j:=3\}$ is irrelevant for the validity of the sequent, and simplified it away.

We continue by calling the **assignment** rule a third time (which requires that the **OPEN GOAL** is selected again). When looking at the resulting **Current Goal**, we note that indeed the assignment `i=2;` turned into the update $\{i:=2\}$, but this time, the older update $\{i:=1\}$ got lost. The reason is again that the prover *eagerly* applies the update simplifier, which this time turned the two updates $\{i:=1\}\{i:=2\}$ (see the corresponding inner node) into $\{i:=2\}$ only.

With the empty modality highlighted in the **OPEN GOAL**, we can apply the rule **emptyModality**. It deletes that modality, and results in the sequent

¹⁸ Strictly speaking, the curly brackets are not part of the update, but rather surround it. It is however handy to ignore this syntactic subtlety when discussing examples.

$\Rightarrow \{i:=2\}(i = 2)$ ¹⁹. However, we cannot immediately see that sequent, because again the update simplifier resolved that goal, by applying the update *as a substitution*. We refrain from finishing this proof interactively, and just press the play button instead.

Before moving on, we note that the examples which are discussed here are not intended (and not sufficient) for justifying the presence of updates in the logic really. Such a discussion would certainly exceed the scope of this chapter. We only mention here that one of the major tasks of the update simplifier is the proper handling of object aliasing.

Employing Active Statements

We are going to prove the same problem again, this time focusing on the connection between programs in modalities on the one hand, and taclets on the other hand. For that, we load `updates.key` again. When moving the mouse around over the single formula of the **Current Goal**,

```
\<{
  i=1;
  j=3;
  i=2;
}\> i = 2
```

we realise that, whenever the mouse points anywhere between (and including) “\<{” and “}\>”, the whole formula gets highlighted. However, the first statement is highlighted in a particular way, with a different colour, regardless of which statement we point to. This indicates that the system considers the first statement `i=1`; as the *active statement* of this DL formula.

Active statements are a central concept of the DL calculus used in KeY. They control the application/applicability of taclets. Also, all rules which modify the program inside of modalities operate on the active statement, by rewriting or removing it. Intuitively, the active statement stands for the statement next to be executed. In the current example, this simply translates to the *first* statement.

We click anywhere within the modality, and *preselect* (only) the taclet **assignment**, just to view the actual taclet presented in the tooltip:

— Tooltip —

```
assignment {
  \find (
    \modality{#normalassign}{ ..
      #loc=#se;
      ... }\endmodality post
  )
```

¹⁹ Note that `i = 2` here is a formula, not a JAVA assignment. An assignment would end with a “;”, and could only appear within a modality.

```

\replacewith (
  {#loc=#se}
  \modality{#normalassign}{ .. ... }\endmodality post
)
\heuristics ( simplify_prog_subset, simplify_prog )
}

```

Tooltip —

The `\find` clause tells us how this taclet matches the formula at hand. First of all, the formula must contain a modality followed by a (not further constrained) formula `post`. Then, the first argument of `\modality` tells which kinds of modalities can be matched by this taclets. (We ignore that argument here, mentioning just that the standard modality `(.)` is covered.) And finally, the second argument of `\modality`, “`.. #loc=#se; ...`” specifies the code which this taclet matches on. The convention is that everything between “`..`” and “`...`” matches the *active statement*. Here, the active statement must have the form “`#loc=#se;`”, i.e., a statement assigning a simple expression to a location, here `i=1;`. The “`...`” refers to the rest of the program (here `j=3; i=2;`), and the match of “`..`” is empty, in this particular example. Having understood the `\find` part, the `\replacewith` part tells us that the active statement moves out into an update.

After applying the taclet, we point to the active statement `j=3;`, and again preselect the **assignment**. The taclet in the tooltip is the same, but we note that it matches the highlighted *sub*-formula, below the leading update. We suggest to finish the proof by pressing the play button.

The reader might wonder why we talk about “active” rather than “first” statements. The reason is that our calculus is designed in a way such that *block statements* never are “active”. By “block” we mean both unlabelled and labelled JAVA blocks, and well as try-catch blocks. If the first statement inside the modality is a block, then the active statement is the first statement *inside* that block, if that is not a block again, and so on. This concept prevents our logic from being bloated with control information. Instead, the calculus works inside the blocks, until the whole block can be *resolved* (because it is either empty or a **break**, resp., **throw** is active). The interested reader is invited to examine this by loading the file `activeStmt.key`. Afterwards, one can see that, as a first step in the proof, one can pull out the assignment `i=0;`, even if that is nested within a labelled block and a try-catch block. We suggest to perform this first step interactively, and prove the resulting goal automatically, for inspecting the proof afterwards.

Now we are able to round up the explanation of the “`..`” and “`...`” notation used in DL taclets. The “`..`” matches the opening of leading blocks, up to the first non-block (i.e., active) statement, whereas “`...`” matches the statements following the active statement, plus the corresponding closings of the opened blocks.²⁰

²⁰ “`..`” and “`...`” correspond to π and ω , respectively, in the rules in Chap. 3.

Executing Programs Symbolically

So far, all DL examples we have been trying the prover on in this chapter had in common that they worked with concrete values. This is very untypical, but served the purpose of focusing on certain aspects of the logic and calculus. However, it is time to apply the prover on problems where (some of) the values are either completely unknown, or only constrained by formulae typically having many solutions. After all, it is the ability of handling symbolic values which makes theorem proving more powerful than testing. It allows to verify a program with respect to *all* legitimate input values!

First, we load the problem `symbolicExecution.key`:

— KeY Problem File —

```
\predicates { p(int,int); }
\functions { int a; }
\programVariables { int i, j; }
\problem {
    {i:=a}
    \<{
        j=(i=i+1)+3;
    }\> p(i,j)
}
```

— KeY Problem File —

This problem is a variation of `executeByProving.key` (see above), the difference being that the initial value of “i” is *symbolic*. The “a” is a logical *constant* (i.e., a function without arguments), and thereby represents an unknown, but fixed value in the range of `int`. The update `{i:=a}` is necessary because it would be illegal to have an assignment `i=a;` inside the modality, as “a” is not an element of the JAVA language, not even a program variable. This is another important purpose of updates in our logic: to serve as an interface between logical terms and program variables.

The problem is of course as unprovable as `executeByProving.key`. All we want this time is to let the prover compute the symbolic values of `i` and `j`, with respect to `a`. We get those by running the **Java DL** strategy on this problem, which results in $\Rightarrow p(1+a, 4+a)$ being the remaining OPEN GOAL. This tells us that `1+a` and `4+a` are the final values of `i` and `j`, respectively. By further inspecting the proof, we can see how the strategy performed symbolic computation (in a way which is typically very different from interactive proof construction). That intertwined with the “execution by proving” method discussed above forms the principle of *symbolic execution*, which lies at the heart of the KeY prover.

Another example for this style of formulae is the `\problem` which we load from `postIncrement.key`:

— KeY Problem File —

```
\functions { int a; }
\programVariables { int i; }
\problem {
    {i:=a}
    \<{
        i=i*(i++);
    }\> a * a = i
}
```

— KeY Problem File —

Depending on the reader’s understanding of JAVA, the validity of this formula is not completely obvious. But indeed, the obscure assignment `i=i*(i++);` computes the square of the original value of `i`. The point is the exact evaluation order within the assignment at hand. It is of course crucial that the calculus allows to, by symbolic execution, emulate the evaluation order exactly as it is specified in the JAVA language description, *and* that the calculus does not allow any other evaluation order.

We prove this formula automatically and, as always, suggest that the reader scrolls through the proof afterwards, not to check all details, but to get an impression on how KeY symbolically executes the program.

Quantifying over Values of Program Variables

A DL formula of the form $\langle p \rangle \varphi$, possibly preceded by updates, like $\{u\} \langle p \rangle \varphi$, can well be a sub-formula of a more complex DL formula. One example is the form $\psi \rightarrow \langle p \rangle \varphi$, where the diamond formula is below an implication (see, for instance, formula (10.1)). A DL sub-formula can actually appear below arbitrary logical connectives, *including quantifiers*. The following problem formula from `quantifyProgVals.key` is an example for that.

— KeY Problem File —

```
\programVariables { int i; }
\problem {
    \forall int x;
    {i := x}
    \<{
        i = i*(i++);
    }\> x * x = i
}
```

— KeY Problem File —

Note that it would be illegal to have an assignment `i=x;` inside the modality, as “`x`” is not an element of the JAVA language, but a *logical* variable instead.

This formula literally says that, `\forall` initial values `i`, it holds that after the assignment `i` contains the square of that value. Intuitively, this

seems to be no different from stating the same for an *arbitrary but fixed* initial value “a”, as we did in `postIncrement.key` above. And indeed, if we load `quantifyProgVals.key`, and as a first step apply the taclet `allRight`, then the **Current Goal** looks like this:

— KeY Output —

```
==>
  {i:=x_0}
  \<{
    i=i*(i++);
  }\> x_0 * x_0 = i
```

— KeY Output —

Note that `x_0` cannot be a logical variable (as was `x` in the previous sequent), because it is not bound by a quantifier. Instead, `x_0` is a *Skolem constant* (cf. the earlier discussion of Skolem terms).

We see here that, after only one proof step, the sequent is essentially no different from the initial sequent of `postIncrement.key`. This seems to indicate that quantification over values of program variables is not necessary. That might be true here, but is not the case in general! The important proof principle of *induction* applies to quantified formulae, only! Using KeY for inductive proving is so important that there is a separate chapter (\Rightarrow Chap. 11) reserved for that issue.

Proving DL Problems with Program Variables

So far, most DL `\problem` formulae *explicitly* talked about *values*, either concrete ones (like “2”) or symbolic ones (like the logical constant “a” and the logical variable “x”). It is however also common to have DL formulae which do not talk about any (concrete or symbolic) values explicitly, but instead only talk about *program variables* (and thereby *implicitly* about their values). As an example, we use yet another variation of the post increment problem, contained in `postIncrNoUpdate.key`:

— KeY Problem File —

```
\programVariables { int i, j; }
\problem {
  \<{
    j=i;
    i=i*(i++);
  }\> j * j = i
}
```

— KeY Problem File —

Here, instead of initially updating `i` with some symbolic value, we store the value of `i` into some other program variable. The equation after the modality then is a claim about the relation between (the implicit values of) the

program variables, in a state after program execution. When proving this formula automatically with KeY, we see that the proof has no real surprise as compared to the other variants of post increment. Please observe, however, that the entire proof does not make use of any symbolic value, and only talks about program variables, some of which are introduced within the proof.

In this context, it is very natural to come back to the formula

$$x < y \rightarrow (\text{int } t = x; x = y; y = t;) y < x$$

which we discussed in the beginning of this section (Sect. 10.2.3). Also this formula only talks about program variables. It assumes the (values of) the variables having a certain relation in the initial state, and states that (the values of) these variables have a different relation after execution of the program.

We load the corresponding problem file, `exchange.key` (which was displayed on page 434) into the system. After proving this problem automatically, we want to point the reader to one interesting detail. When scrolling down this proof, we see the usual course of symbolic execution: programs are transformed into one another, simple assignments turn into updates, and updates are simplified. We stop at the inner node where the modality is already gone, and the last remaining update is *about to* disappear for the rest of the proof (by being applied as a substitution). *Currently* this inner node looks like $y \geq 1 + x \implies \{x:=y, y:=x\}(y \leq -1 + x)$. In contrast to previous examples, here it really matters that the update $\{x:=y, y:=x\}$ is a *parallel* one. The variables `x` and `y` switch their values at once, and no auxiliary variable is needed at this point.

Calling Methods in Proofs

Even though the DL problem formulae discussed so far all contained real JAVA code, we did not see either of the following central JAVA features: classes, objects, or method calls. The following small example features all of them.

We consider the file `methodCall.key`:

— KeY Problem File —

```

\javaSource "methodExample/"; // location of class definitions
\programVariables { Person p; }
\problem {
  \forall int x;
    {p.age:=x} // assign initial value to "age"
    ( x >= 0
      -> \<{
          p.birthday();
        }\> p.age > x)
}

```

— KeY Problem File —

The `\javaSource` declaration tells the prover where to look up the sources of classes (and interfaces) used in the file. In particular, the JAVA source file `Person.java` is contained in the directory `methodExample/`. The problem formula states that a `Person p` is getting older at its `birthday()`. (On the side, the reader may note that the update here does not immediately precede a modality, but a more general DL formula.)

Before loading this problem file, we look at the source file `Person.java` in `methodExample/`:

— JAVA —

```
public class Person {
    private int age = 0;
    public void setAge(int newAge) { this.age = newAge; }
    public void birthday() {
        if (age >= 0) age++;
    }
}
```

— JAVA —

The reader is encouraged to reflect on the validity of the above problem formula a little, before reading on.—Ready?—Luckily, we have a prover at hand to be certain. We load `methodCall.key` into KeY and, without hesitation, press the play button (assuming that **Java DL** is the selected strategy).

The strategy stops with the OPEN GOAL “`p = null, x_0 >= 0 ==>`” left.²¹ There are different ways to read this goal, which however are logically equivalent. One way of proving any sequent is to show that its left-hand side is false. Here, it would be sufficient to show that `p = null` is false. An alternative viewpoint is the following: in a sequent calculus, we always get a logically equivalent sequent by throwing any formula to the respective other side, but negated. Therefore, we can as well read the OPEN GOAL as if it was “`x_0 >= 0 ==> p != null`”. Then, it would be sufficient to show that `p != null` is true.

Whichever reading we choose, we cannot prove the sequent, because we have no knowledge whatsoever about `p` being `null` or not. When looking back to our problem formula, we see that indeed the formula is not valid, because the case where `p` is `null` was forgotten. The postcondition `p.age > x` depends on the method body of `birthday()` being executed, which it cannot in case `p` is `null`. We can even read this off from the structure of the uncompleted proof in the **Proof** pane. When tracing the branch of the OPEN GOAL, back to the first split, we can see that the proof failed in the branch marked as “Null Reference (`p = null`)”. It was the taclet `methodCall` which triggered this split.

²¹ If not, please select `nullCheck` as the `nullPointerPolicy` (see page 446) and load `methodCall.key` again.

The file `methodCall12.key` contains the patch of the problem formula. The problem formula from above is preceded by “`p != null ->`”. We load that problem, and let KeY prove it automatically without problems. In this proof, we want to have a closer look on the way KeY handles method calls. Like in the previous proof, the first split was triggered by the taclet `methodCall`. Then, in the branch marked as “Normal Execution (`p != null`)”, the second inner node (after some update simplification) looks like this:

— KeY Output (10.1) —

```

x_0 >= 0
==>
p = null,
{p.age:=x_0}
\<{
  p.birthday();
}\> x_0 <= -1 + p.age

```

— KeY Output —

We should not let confuse ourselves by `p = null` being present here. Recall that the comma on the right-hand side of a sequent essentially is a logical *or*. Also, as stated above, we can always imagine a formula being thrown to the other side of the sequent, but negated. Therefore, we essentially have `p != null` as an *assumption* here. Another thing to comment on is the `@Person` notation in the method call. It represents that the calculus has decided which *implementation* of `birthday` is to be chosen (which, in the presence of inheritance and hiding, can be less trivial than here).

At this point, the strategy was ready to apply `methodBodyExpand`. After that, the code inside the modality looks like this:

```

method-frame(source=Person,this=p): {
  if (age>=0) {
    age++;
  }
}

```

This `method-frame` is the only really substantial extension over JAVA which our logic allows inside modalities. It models the execution stack, and can appear nested in case of nested method calls. Apart from the class and the `this` reference, it can also specify a return variable, in case of non-void methods. However, the user is rarely concerned with this construction, and if so, only passively. We will not discuss this construct further here, but refer to Chap. 3, Sect. 3.6.5 instead. One interesting thing to note here, however, is that method frames are considered as *block statements* in the sense of our earlier discussion of active statements, meaning that *method frames are never active*. For our sequent at hand, this means that the active statement of the discussed formula is `if (age>=0) {age++};`, as one can also see from the taclet which was applied next.

Controlling Strategy Settings

The expansion of methods is among the more problematic steps in program verification (together with the handling of loops). In place of recursion, an automated proof strategy working with method expansion might not even terminate. Another issue is that method expansion goes against the principle of *modular* verification, without which even mid-size examples become infeasible to verify. These are good reasons for giving the user more control over this crucial proof step.

KeY therefore allows to configure the automated strategies in a way that they *refrain* from expanding methods automatically.²² We try this out by loading `methodCall12.key` again, and selecting **None** as the **Method treatment** option in the **Proof Search Strategy** tab. Then we start the strategy, which now stops exactly at the sequent which we discussed earlier (Sect. 10.1). We can highlight the active statement, and *could* call `methodBodyExpand` interactively. KeY would then *only* apply this very taclet, and stop again. Therefore, we first check the **Autoresume Strategy** checkbox, and then apply `methodBodyExpand`. The strategy will resume automatically, and close the proof.

Controlling Taclet Options

The proof of `methodCall12.key` has a branch for the `null` case (“Null Reference (`p=null`)”), but that was closed after a few steps, as `p = null` is already present, explicitly, on the right side of the sequent (`closeGoal`). It is, however, untypical that absence of null references can be derived so easily. Often, the “null branches” complicate proofs substantially. The KeY system allows to use a variant of the calculus which ignores the problem of null references. This is actually only one of the issues which are addressed by *taclet options* (see Sect. 4.4.2).

We open the taclet option dialogue, via **Options** → **Taclet options defaults**. Among the option categories, we select the `nullPointerPolicy`, observe that `nullCheck` is chosen as default, and change that by selecting `noNullCheck` instead. Even if the effect of this change on our very example is modest, we try it out, to see what happens in principle. We again load `methodCall12.key`, press play, and observe that indeed the finished proof has only one branch.

One has to be aware that this change has a dramatic consequence: it affects the soundness of the calculus. To demonstrate this, we load the original problem formula from `methodCall12.key` again. By running the automated strategy with the current taclet options, we can see that the system now is able to prove the non-valid formula! As a consequence, one should only switch off the proper null handling if one is, for whatever reason, not interested in problems that originate from null references. Another scenario is that one first tries to prove a problem under the simplifying assumption of no null references being present, which allows to focus attention to other complications of the

²² For a discussion of loop treatment, please refer to Chaps. 11 and 3.

problem at hand. Thereafter, one can re-prove the problem *with* null check again, with help of the re-use facility (see Chap. 13).

Integer Semantics Options

We briefly mention another very important taclet option, the **intRules**. Here, the user can choose between different semantics of the primitive JAVA integer types **byte**, **short**, **int**, **long**, and **char**. Options are: the mathematical integers (easy to use, but not fully sound), mathematical integers with overflow check (sound, reasonably easy to use, but unable to verify programs which depend on JAVA's modulo semantics), and the true modulo semantics of JAVA integers (sound, complete, but difficult to use). This book contains a full chapter on JAVA Integers (Chap. 12), discussing the different variants in the semantics and the calculus. Fig. 12.1 displays the corresponding GUI dialogue. Please note that KeY 1.0 comes with the mathematical integer semantics chosen as default option, to optimise usability for beginners. However, for a sound treatment of integers, the user should switch to either of the other semantics. As an alternative, we suggest to use the *proof reuse* feature of KeY (see Chap. 13). One can first construct a proof using the mathematical integer option, and then replay it with the mathematical overflow semantics selected.

10.3 Generating Proof Obligations

We have so far applied KeY on several examples which were meant to demonstrate the most essential features of the logic, the calculus, the prover, and, in particular, the usage of the prover. All those examples had in common that the proof obligations were hand crafted, and stored in `.key` files.

However, even if the logical framework and the prover technology forms an essential part of the KeY project, the whole KeY approach to formal methods is not all about that. Instead, it is very much about the *integration* of verification technology into more conventional software development methods, as was outlined in the introductory chapter of this book (Chap. 1). Sect. 1.1 gave an overview on how we use modern *object-oriented modelling* approaches as *hooks* for formal verification. In particular, KeY so far employs two modelling/specification languages: UML's Object Constraint Language (OCL) and the Java Modeling Language (JML). These languages, their usage and their theory, are described in Chap. 5 in this book.

KeY interfaces with OCL as well as JML, by translating them (and the specified JAVA code) into *proof obligations* in JAVA CARD DL. This issue, and the rich theory behind it, is described in Chap. 8.

But not only does KeY interface with certain standard specification languages. It also interfaces with *standard tools* for software development, currently the commercial CASE tool Borland Together, and the freely available IDE Eclipse. An overview over the architectural setup of this integration

was given in Fig. 1.1 (Chap. 1). Following that figure from the right to the left, we have essentially four scenarios, varying in the origin of proof obligations (POs):

1. *Hand-crafted* POs, to be loaded from `.key` files.
2. *Automatically generated* POs
 - a) from JML-augmented JAVA source files, using
 - i. the JML browser of the KeY stand-alone system.
 - ii. Eclipse with the KeY plug-in.
 - b) from OCL-augmented UML diagrams and JAVA source files, using Borland Together with KeY extensions.

Scenario 1 has been practised in the course of the previous section. Below, we focus on Scenario 2.

Using the JML Specification Browser of the KeY Stand-Alone System

JAVA classes and their methods are specified in JML using *class invariants* and *method contracts*.²³ It is part of the concept of JML that specifications are included in JAVA source code, in the form of particular *comments*. If we want to verify that JML specifications are respected by the corresponding implementations, we can let the KeY system generate corresponding proof obligations (in JAVA CARD DL) from these JML comments. The KeY stand-alone system supports this by offering a *JML specification browser*.

In the directory `Bank-JML`, the reader finds the JAVA sources of a banking scenario²⁴. Using an editor, we can see that the `.java` files in `Bank-JML` indeed contain JML specifications. We focus on the file `ATM.java`, and therein on the contract of the method `enterPIN` (textually located in the comment preceding the method). This contract is also displayed in Fig. 5.14, Sect. 5.3.1 (Chap. 5). The same section contains a detailed explanation of this very JML contract!

A JML contract can be composed from several more elementary contracts, connected by the keyword “`also`”. The contract of `enterPIN` is composed from three such parts, the last of which specifies the case where a wrong PIN has been entered too often:

— JML (10.2) —

```

public normal_behavior
requires insertedCard != null;
requires !customerAuthenticated;
requires pin != insertedCard.correctPIN;
requires wrongPINCounter >= 2;
assignable insertedCard, wrongPINCounter,
           insertedCard.invalid;

```

²³ Method contracts are referred to as “operation contracts” in Chap. 5.

²⁴ This scenario is used in a course at Chalmers University, see Chap. 1.



```

ensures    insertedCard == null;
ensures    \old(insertedCard).invalid;
ensures    !customerAuthenticated;

```

JML

The displayed part of the JML contract gives rise to two POs, to be generated by the JAVA CARD DL translation of KeY: one PO for verifying the “assignable” conditions, and one PO for verifying the “ensures” conditions. The latter PO we want to generate, and prove, with the KeY system.

First of all, we activate the JML browser on the directory **Bank-JML**, by loading the entire directory, containing *JAVA+JML sources* into the system. This is done in the same manner as loading problem files and proofs by **File** → **Load ...** or clicking at  in the tool bar. It is important that we have the directory, here **Bank-JML**, selected when pressing the **Open** button (not any of the contained files).

The system now analyses the *JAVA+JML sources* in **Bank-JML**, and opens the **JML Specification Browser** window. In its **Classes** pane, the available classes are grouped after the packages they belong to. The application classes of our scenario all belong to the package **bank**, so we make sure that folder is expanded, and select the class **ATM**. The **Methods** pane shows the methods of class **ATM**. After selecting **enterPIN**, the **Proof Obligations** pane allows choosing a PO connected to that method, to be loaded into the system. We ignore the Assignable POs for now, and among the three others choose the one which corresponds to the piece of JML quoted above (10.2), and press **Load Proof Obligation**.

We find ourselves in a familiar situation: a new proof task is loaded into the system, and the initial sequent is presented in the **Current Goal** pane. The sequent looks very substantial. How this PO was constructed cannot be discussed here in detail. (We refer to Chap. 8.) Still, we comment a bit on the overall structure of this PO, with the intention to demystify its lengthy appearance.

The PO is an implication, with “**inReachableState**” acting as basic condition under which the rest of the PO must be true. The predicate **inReachableState** restricts the states to those reachable by *any* JAVA computation. For instance, **inReachableState** implies that all referenced (non-null) objects are actually created.

The remaining PO starts with some quantifiers and updates. Thereafter, we have an implication basically saying: “the (translated) **requires** part, together with the (translated) class invariant, implies that the (translated) **ensures** part holds after the method”. Note that it is the translated class invariant which makes the PO so long. That however is not a burden from the proving perspective. To the contrary: being on the left side of the implication, the invariant only provides additional assumptions that may, or may not, be used for establishing the right-hand side.

By simply pressing the play button, we make KeY proving this PO automatically.

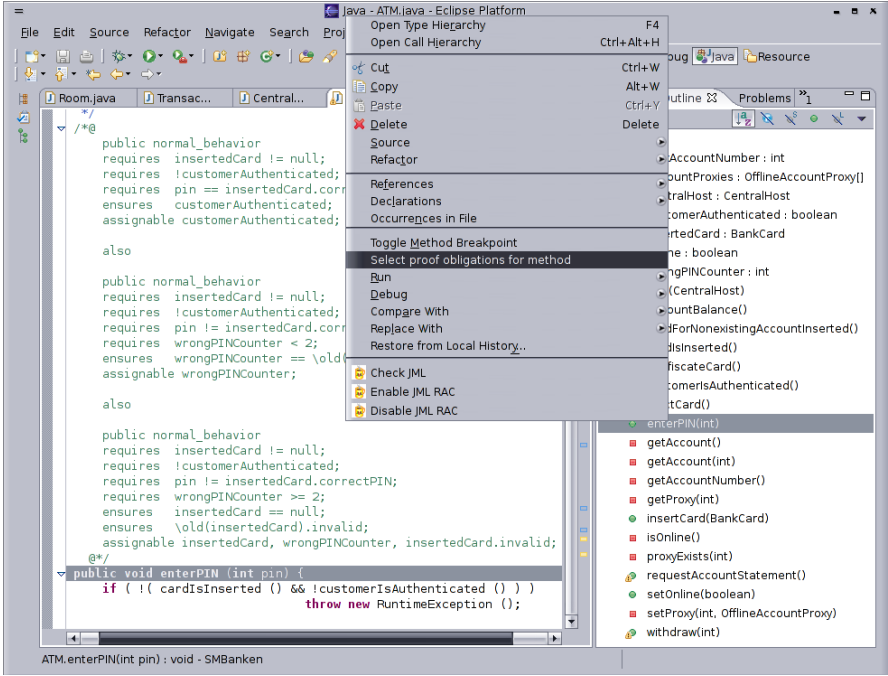


Fig. 10.1. KeY-Eclipse integration

Using Eclipse with KeY Plug-in

Contemporary software development makes more and more use of tools which integrate the different activities around the development of programs. One such tool is the freely available IDE Eclipse²⁵, which currently is the most widely distributed IDE for JAVA. It provides powerful coding support, like code templates, code completion, and import management. Eclipse also features a well documented plug-in interface.

KeY also comes as one such Eclipse plug-in. When developing JAVA+JML code within Eclipse, the usual context menus offer the additional functionality of selecting proof obligations, as indicated in Fig. 10.1 (for our ATM.`enterPIN` example). The KeY plug-in will start up automatically, generate the selected proof obligation, and present it in the prover window, ready for automated, resp. interactive proving. We refer to documentation and tutorials, available from the KeY project's web page for updated information about how to install—and use—a KeY-equipped Eclipse platform as a front end in the verification of JML specified JAVA programs.

²⁵ www.eclipse.org

Using Borland Together with KeY Extensions

CASE tools go beyond IDEs in their integration of software *modelling* activities, which normally includes support of (various aspects of) UML. The KeY project propagates the use of formal methods early in the software process, and therefore makes a serious attempt to integrate facilities for specification and verification into *tools* of that kind. Such an integration has been exemplified by augmenting the commercial CASE tool Borland Together with KeY extensions.

In this context, the hook for formal methods consists of UML/OCL. In the KeY-extended Borland Together, UML class diagrams can be decorated with OCL constraints. The creation of such constraints is supported a) by parsing, b) by KeY OCL idioms and KeY OCL patterns, with an corresponding pattern instantiation mechanism (see Chap. 6), and c) by a structural, multi-lingual editor, for simultaneous editing, and cross translation, of constraints in OCL respectively natural language (English, German) (see Chap. 7).

As for the verification side, context menus allow to, for instance, choose proof obligations directly from the class diagram view of a project. Also here, KeY will generate the chosen proof obligation, and start up the prover, ready to prove the goal at hand. We again refer to the KeY project's web page for the version-sensitive information of how to install—and use—the KeY extensions on top of Borland Together.