Chapter 15
Using the KeY Prover

Wolfgang Ahrendt and Sarah Grebing

15.1 Introduction

This whole book is about the KeY *approach* and *framework*. This chapter now focuses on the KeY *prover*, 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 artifacts of the framework, like formulas of the used logic, proofs within the used calculus, elements of the used specification languages, among others. Therefore, these artifacts 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 prover, without necessarily having to read 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 do 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 the application of KeY to larger as well as more difficult scenarios. However, this would raise the threshold for getting started with the prover.

The KeY framework was designed from the beginning to be usable *without* having to read a thick book first. Software verification is a difficult task anyhow. Neither the system nor the used artifacts (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 formulas, allowing the user to relate back to the program when proving properties about it.

The *taclet* 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 automated 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, nontrivial tasks left for the user. It is the very purpose of the GUI to support those tasks well.
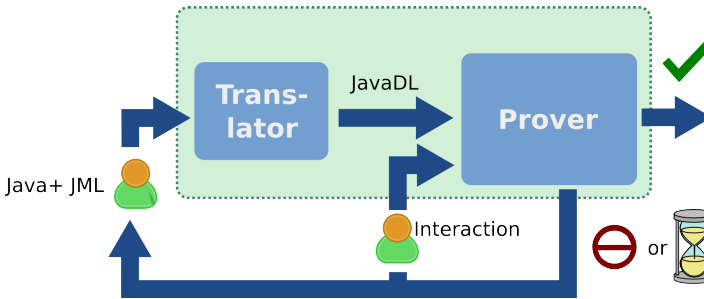

Figure 15.1 Verification process using the KeY system

The general proof process using the KeY system is illustrated in Figure 15.1. The user provides Java source code with annotations written in JML and passes them to the KeY system, which translates these artifacts into a proof obligation in Java Dynamic Logic. Now the user is left with the choice of trying to let the prover verify the problem fully automatically or of starting interactively by applying calculus rules to the proof obligation. If the user chooses to start the automated proof search strategy offered by the prover, the result can be one of the two: either the prover succeeds in finding a proof or the prover stops, because it was not able to apply more rules automatically (either because the maximal number of proof steps has been reached or the prover cannot find anymore applicable rules). This is the point in the proof process, where the user gets involved. The user now has to decide whether to guide the prover in finding a proof by applying certain rules (or proof steps) by hand or whether the look for mistakes in the annotation or the corresponding source code, which is one reason why the prover is not able to apply rules automatically anymore. Observing a wrong program behavior or specification leads the user to the correction of the mistake in the source code or annotation and to start the whole proof process over again.

When proving a property which is too involved to be handled fully automatically, certain steps need to be performed in an interactive manner. 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. 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 subentity 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. These tooltips will be described in 15.2.1. 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, and arbitrary undoing of proof steps.

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, customizing the calculus (for instance by choosing either of the mathematical or the Java semantics for integers), and generating proof obligations from specifications. Working with the KeY system has therefore many aspects, and there are many ways to give an introduction into those. In the following, we focus on the KeY prover only, taking an 'inside out' approach, describing *how* the prover and the user communicate *which artifacts* for *which purpose* with each other. In addition hints for the user are provided on how to proceed in the verification process interactively, when automation stops.

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 a step by step introduction of the concepts and usage of the KeY system. They are not suitable for giving any indication of the capabilities of the system. (See Chapter 16 instead.)

Before we start, there is one last basic issue to discuss 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 those 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 public release of the KeY system. 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, 2.6, on the KeY book's web page. Second, in order to not restrict the reader to that release only, we will try to minimize the dependency of the material on the current version of the system and its GUI. Third, whenever 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 the automated proof search strategy which is restricted to a highlighted term/formula by $\boxed{\texttt{Shift}}$ + click on it." There is a separate document shipped with the KeY system, the *Quicktour*[1] which is updated more often and describes the current GUI and features of the KeY system.

Menu navigation will be displayed by connecting the cascaded menu entries with "→", e.g., **Options → SMT Solvers Options**. Note that menu navigation is release dependent as well. Most functionalities in the KeY system can also be activated by keystrokes in order to be more efficient while performing proof tasks.

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 2.6, the release version related to this book, 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/thebook2. The example files can also be accessed via the **File → Load examples** as well. However, for this chapter we assume the reader to have downloaded the example files from the web-page and extracted them to a folder in the reader's system.

15.2 Exploring KeY Artifacts and Prover 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. There are two ways to start the stand-alone KeY prover. Either you download the archive of KeY 2.6, unpack it, and in the key directory execute the key.jar file, in the standard way .jar files are executed in your system and setup. Or you execute KeY directly from your browser, by navigating to the **webstart** link of KeY 2.6, and simply click it.

In both cases, the **KeY–Prover** main window pops up. 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. Please note that most of the GUI components are labeled with tooltips, which are visible when hovering over that component. They give useful information about the features of the system.

---

[1] Available from the Download pages at www.key-project.org.

15.2.1 Exploring Basic Notions And Usage: Building A Propositional Proof

In general, the KeY prover is made for proving formulas 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 proven. (The general format of `.key` files is documented in Appendix B.) For now, we look into the file `andCommutes.key` itself (using your favorite text editor):

—— KeY Problem File ——————————————————————————

```
\predicates {
        p;
        q;
}
\problem {
        (p & q) -> (q & p)
}
```
————————————————————————— KeY Problem File ——

The `\problem` block contains the formula to be proven (with `->` denoting the logical implication and `&` denoting the logical and). 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 selecting **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, as well as locations referenced by the file. This includes the source folder and its subdirectories.

Reading the Initial Sequent

Afterwards, we see 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 artifact on which the calculus operates. Sequents have the form

$$\phi_1, \ldots, \phi_n \Longrightarrow \phi_{n+1}, \ldots, \phi_m$$

with $\phi_1, \ldots, \phi_n$ and $\phi_{n+1}, \ldots, \phi_m$ being two (possibly empty) comma-separated lists of formulas, distinguished by the sequent arrow $\Longrightarrow$ (written as `==>` in both input and output of the KeY system). The intuitive meaning of a sequent is: if we assume all formulas $\phi_1, \ldots, \phi_n$ to hold, then *at least one* of the formulas $\phi_{n+1}, \ldots, \phi_m$ holds. In our particular calculus, the order of formulas within $\phi_1, \ldots, \phi_n$ and within $\phi_{n+1}, \ldots, \phi_m$ does not matter. Therefore, we can for instance write $\Gamma \Longrightarrow \phi \rightarrow \psi, \Delta$ to refer to sequents where *any* of the right-hand side formulas is an implication. Γ and Δ are both used to refer to arbitrary, and sometimes empty, lists of formulas. We refer to Chapter 2, Section 2.2.2, 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 manually for now, we want the system to minimize 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 menu item **Minimize interaction** option (at **Options** $\rightarrow$ **Minimize interaction**) is checked for the whole course of this chapter.

Applying the First Rule

The sequent `==> p & q -> q & p` displayed in the **Current Goal** pane states that the formula `p & q -> q & p` holds *unconditionally* (no formula left of `==>`), and *without alternatives* (no other formula right of `==>`). 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 take such formulas apart, step by step, 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*[2]. (For instance, the sequent $\phi_1, \phi_2 \Longrightarrow \phi_3, \phi_1$ is trivially true. Assuming both, ϕ_1 and ϕ_2, indeed implies that "at least one of ϕ_3 and ϕ_1" holds, namely ϕ_1.) It is such primitive shapes which we aim at when proving.

'Taking apart' a formula refers to breaking it up at its top-level operator. The displayed formula `p & q -> 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 memorize the built in operator precedences. Instead, the term structure gets clear by moving the mouse pointer back and forth over the symbols in the formula area, as the subformula (or subterm) under the symbol currently pointed at always gets highlighted. In general to get the whole sequent highlighted, the user needs to point to the sequent arrow `==>`. To get the whole formula highlighted in our example, the

---

[2] There are two more cases, which are covered in Section 15.2.2 on page 513.

user needs to point to the implication symbol `->`, 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. A left mouse-click on `->` will open a context menu for rule selection, offering several rules applicable to this sequent, among them **impRight**, which in the usual text book presentation looks like this:

$$\text{impRight} \quad \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \to \psi, \Delta}$$

The conclusion of the rule, $\Gamma \Longrightarrow \phi \to \psi, \Delta$, is not simply a sequent, but a sequent *schema*. In particular, ϕ and ψ are *schema variables* for formulas, to be instantiated with the two subformulas of the implication formula appearing on the right-hand side of the current sequent. (Γ and Δ denote the sets of all formulas on the left- and right-hand side, respectively, which the rule is *not* concerned with. In this rule, Γ are all formulas on the left-hand side, and Δ are all formulas on the right-hand side except the matching implication formula.)

As for any other rule, the *logical meaning* of this rule is downwards (concerning validity): if a sequent matching the premiss $\Gamma, \phi \Longrightarrow \psi, \Delta$ is valid, we can conclude that the corresponding instance of the conclusion $\Gamma \Longrightarrow \phi \to \psi, \Delta$ is valid as well. On the other hand, the *operational meaning* during proof construction goes upwards: the problem of proving a sequent which matches $\Gamma \Longrightarrow \phi \to \psi, \Delta$ is reduced to the problem of proving the corresponding instance of $\Gamma, \phi \Longrightarrow \psi, \Delta$. 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 resulting from the rule's premiss. (This will be generalized to rules with multiple premisses later on.)

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 (the node selected in the proof tree in the **Proof** tab).

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 current (unfinished) proof as a tree. All nodes of the current proof are numbered consecutively, and labeled either by the name of the rule which was applied to that node, or by **OPEN GOAL** in case of a goal. The selected and highlighted node is always the one which is detailed in the **Current Goal** or **inner node** pane in the right part of the window. So far, this was always a goal, such that the pane was called **Current Goal**. But if the user clicks at an *inner node*, in our case on the one labeled with **impRight**, that node gets detailed in the right pane

now called **Inner Node**. It can not only show the sequent of that node, but also, if the checkbox **Show taclet info** is selected, the upcoming rule application.

Please observe that the (so far linear) proof tree displayed in the **Proof** tab has its root on the top, and grows downwards, as it 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:

$$\frac{p \wedge q \Longrightarrow q \wedge p}{\Longrightarrow p \wedge q \rightarrow q \wedge p}$$

For the on-paper presentation of the proof to be developed, we refer to Chapter 2.

Understanding the First Taclet

With the inner node still highlighted in the **Proof** tab, we click onto the checkbox **Show taclet info (Inner Nodes only)** in the left lower corner of the **Proof** tab. We now obtain the rule information in the **Inner Node** pane, saying (simplified):

```
—— 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 domain specific language for programming sequent calculus rules, 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 Chapter 4.

The taclet **impRight** corresponds to the traditional sequent calculus style presentation of **impRight** we gave earlier. The schema b -> c in the \find clause indicates that the taclet is applicable to sequents if one of its formulas is an implication, with b and c being schema variables matching the two subformulas 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. For this example the taclet is only applicable to implications on the *right-hand side* of the sequent (as b -> 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 -> q & p) gets deleted, and then the subformulas matching b and c (here p & q and q & p) are added to the sequent. To which side of the sequent p & q or q & p, respectively, are added 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 formulas 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 clause the taclet declares itself to be part of some heuristics, here the alpha heuristics which defines the priority with which the rule is applied during the execution of the 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 **Info** tab as part of the tabbed pane in the lower left corner, within the **Rules** →**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 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, especially those related to symbolic execution, are usually applied automatically.

Backtracking the Proof

So far, we performed only one tiny 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 selected (by clicking on it in the **Proof** tab). We now *undo* the proof step which led to this goal, by clicking at ↶ (Goal Back) in the task bar or using the short cut Ctrl + Z . In this example, 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. Please observe that **Goal Back** reverts always only the last rule application and not for instance, all rules applied automatically by the proof search strategy.

---

[3] Modulo leading updates, see Section 15.2.3.

Viewing and Customizing 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 `->` 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 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. Note that, in either case, schema variables can be already instantiated in what is shown in tooltips, also depending on the settings. For this chapter we control the options actively here, and discuss the respective outcome. We open the tooltip options window by **View → ToolTip options**, and make sure that all parts of taclets are displayed by making sure the **pretty-print whole taclet ...** checkbox is checked.

The effect of a taclet to the current proof situation is 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, say, 40 and have the **show uninstantiated taclet** checkbox unchecked. 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 -> 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 formulas `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). Please observe 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, respectively new, sequent. This is not the case in general. The same tooltip would show up when preselecting **impRight** on the sequent: `r ==> p & q -> q & p, s`.

A closer look at the tooltip text in its current form, reveals that the whole `\find` clause actually is redundant information for the user, 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 nonredundant piece of information in this case is therefore the `\replacewith` clause. Accordingly, the tooltips can be reduced to the minimum which is relevant for supporting the selection of the appropriate rule by unchecking **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, for the course of this chapter, please open again the **View → ToolTip options** again, set the "Maximum size ... of tooltips ... with schema variable instantiations displayed ..." to 50 and check both checkboxes, "pretty-print whole taclet ..." as well as "show uninstantiated taclet." Nevertheless, we will not print the `\heuristics` part of taclets in this text further on.

Splitting Up the Proof

We apply **impRight** and consider the new goal `p & q ==> 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 )
}
```
—— 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*. Written as a sequent calculus rule, it looks like this:

$$\text{andRight} \quad \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta}$$

We now generalize 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* premises 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) premises. If we apply **andRight** in the system, the **Proof** tab shows the proof branching into two different **Case**s. In fact, both branches feature an **OPEN GOAL**. At least one of them is currently visible in the **Proof** tab, and highlighted 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, 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 ⊞/⊟.[4] If we expand the yet collapsed branch, we see the full structure of the proof, with both **OPEN GOAL**s being displayed. We can even switch the current goal by clicking on any of the **OPEN GOAL**s.[5]

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

$$\frac{\dfrac{p \wedge q \Longrightarrow q \qquad p \wedge q \Longrightarrow p}{p \wedge q \Longrightarrow q \wedge p}}{\Longrightarrow p \wedge q \rightarrow q \wedge p}$$

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).

There are also several other mechanisms in the KeY system which help inspecting the current proof state. Instead of expanding/collapsing whole branches, it is also possible to hide intermediate proof steps in the current proof tree. This can be done by right-clicking onto the proof tree in the **Proof** tab and selecting the context menu entry **Hide Intermediate Proofsteps**. This results in a more top-level view on the proof tree – only branching nodes, closed and open goals are displayed. Still, some proof trees tend to get quite large with a lot of open and closed goals. For a better overview over the open goals, there is also an option to hide closed goals in the **Proof** tab. It can be accessed similar to the **Hide Intermediate Proofsteps** option. The KeY system incorporates another feature supporting the comprehension of the proof, allowing for textual comments to the proof nodes. This feature is accessible by right clicking onto the proof node in the proof tree and choosing the menu entry **Edit Notes**. A dialog appears in which the user can enter a note which is then attached to the chosen proof node. This note can later be read when hovering with the mouse over the chosen proof node.

Closing the First Branch

To continue, we select **OPEN GOAL** p & q ==> q again. Please recall that we want to reach a sequent where identical formulas appear on both sides (as such sequents are trivially true). We are already very close to that, just that p & q remains to be

---

[4] Bulk expansion, and bulk collapsing, of proof branches is offered by a context menu via right click on any node in the **Proof** tab.

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

decomposed. Clicking at & offers the rule **andLeft**, as usual with the tooltip showing the taclet, here:

—— Tooltip ————————————————————————————————————

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

———————————————————————————————————— Tooltip ——

which corresponds to the sequent calculus rule:

$$\mathsf{andLeft} \;\frac{\Gamma,\phi,\psi \Longrightarrow \Delta}{\Gamma,\phi \wedge \psi \Longrightarrow \Delta}$$

We apply this rule, and arrive at the sequent `p, q ==> 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 formulas.) In the sequent calculus, sequents of the form $\Gamma,\phi \Longrightarrow \phi,\Delta$ are considered valid *without any need of further reduction*.

This argument is also represented by a rule, namely:

$$\mathsf{close} \;\frac{*}{\Gamma,\phi \Longrightarrow \phi,\Delta}$$

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 formulas *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 **close**, however, depends on *two* formulas (or, more precisely, on two formula occurrences). The second formula is taken care of by the \assumes clause in the **close** taclet:

—— Taclet ————————————————————————————————————

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

———————————————————————————————————— Taclet ——

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

This discussion of the **close** sequent rule and the corresponding **close** 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 **close** by a taclet, the reader might click on the q on the *left-hand side* of `p, q ==> q`, and preselect the taclet **close**.

The tooltip will show the taclet:

—— Tooltip —————————————————————————————————————

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

————————————————————————————————————— Tooltip ——

We, however, proceed by applying the taclet **close** on the right-hand side formula q. 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 colored 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 have a node labeled 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 (nonbranching) **andLeft**, once and for all, before the (branching) **andRight**. *In general it is a good strategy to delay proof branching as much as possible and thereby avoiding double work on the different branches.* Without this strategy, more realistic examples with hundreds or thousands of proof steps would become completely unfeasible.

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 redo the proof from the point where we split too early. Instead of reloading the problem file, we can *prune* the proof at the node labeled with **andRight** by right-click on that node, and selecting the context

menu entry **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**. We close the two remaining goals, p, q ==> q and p, q ==> p by applying **close** 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 proven the validity of the *sequent* at the root of the proof tree, here ==> p & q -> q & p. As this sequent has only one formula, placed on the right-hand side, we have actually proven validity of that formula p & q -> 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 artifacts. 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 main menu entry: **Proof** → **Abandon** or the shortcut $\boxed{\texttt{Ctrl}}$ + $\boxed{\texttt{W}}$. 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 menu entry **Reload**, the toolbar button ⟳ or the shortcut $\boxed{\texttt{Ctrl}}$ + $\boxed{\texttt{R}}$. Afterwards, we see a second *'proof task'* being displayed in the **Proofs** pane. One can even switch between the different tasks by clicking in that pane. The newly opened proof shows the **Current Goal** ==> p & q -> q & p, just as last time. For the automated proof process with KeY, we are able to set options in the **Proof Search Strategy** tab. One option is the slider controlling the maximal number of automated rule applications. It should be at least **1000**, which will suffice for all examples in this chapter.

By pressing the "Start/Stop automated proof search" button ▷ in the toolbar, we start the automated proof search strategy. A complete proof is constructed immediately. Its shape (see **Proof** tab) depends heavily on the current implementation of the proof search strategy and the use of the *One Step Simplifier* ➧. One step simplification in the KeY system means that the automated prover performs several simplification rules applicable at once by a single rule application. One example for such a simplification rule is the rule **eq_and** which simplifies the formula true & true to true. Which rules the prover has used can be found in the proof tree in a node labeled with **One Step Simplification**. This option comes in very handy when proofs tend to

get large. Because of the summarization of simplification rules, the proof tree is
more readable. For the following examples we assume that the *One Step Simplifier*
is turned off (the toolbar icon ➡ is unselected) and we point out to the reader when
to toggle it to on. However, the automatically created proof will most likely look
different from the proof we constructed interactively before. For a comparison, we
switch between the tasks in the **Proofs** pane.

Rewrite Rules

With the current implementation of the proof search 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 ==> q & p.
After that, the proof does *not* branch, but instead uses the rule **replace_known_left**:

──── Taclet ──

```
replace_known_left {
  \assumes ( b ==>  )
  \find ( b )
  \sameUpdateLevel
  \replacewith ( true )
}
```

─── Taclet ────

   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.
   Taclets with a \find clause not containing the sequent arrow ==> are called
rewrite taclets or *rewrite rules*. The argument of \find is a schema variable of
kind \formula or \term, matching formulas or terms, respectively, at *arbitrary*
positions, which may even be nested. The position can be further restricted. The
restriction \sameUpdateLevel in this taclet is however not relevant for the current
example. When we look at how the taclet was used in our proof, we see that indeed
the *subformula* p of the formula q & p has been rewritten to true, resulting in the
sequent p, q ==> true & p. The following rule application simplifies the true
away, after which **close** is applicable again.

Saving and Loading Proofs

Before we leave the discussion of the current example, we save the just accomplished
proof (admittedly for no other reason than practicing the saving of proofs). For that,
we either use the shortcut ⎡Ctrl⎤ + ⎡S⎤, or select the main menu item **File** → **Save**, or
the button 📥 in the toolbar. The opened file browser dialogue allows us 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 regardless of whether they are finished or not. An unfinished proof can be continued when loaded again. Loading a proof (finished or unfinished) 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.

15.2.2 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 formulas, 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 c;
}
\problem {
        ( \forall s x; f(f(x)) = f(x) )  ->  f(c) = f(f(f(c)))
}
```

———————————————————————————————— KeY Problem File ——

The file declares a function `f` (of type $s \rightarrow s$) and a constant `c` (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(c)` and `f(f(f(c)))` are equal, given `f` is a projection.

Instantiating Quantified Formulas

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(c) = f(f(f(c)))`.

We proceed by deriving an additional assumption (i.e., left-hand side formula) `f(f(c)) = f(c)`, by instantiating `x` with `c`. 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 `c` onto the quantifier `\forall` by clicking at `c`, 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(c)) = f(c)`.

There is something special to this proof step: even if 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**. We then make sure, that the checkbox **Show taclet info (Inner Nodes only)** at the bottom of the **Proof** tab is checked, such that the **Inner Node** pane displays (simplified):

─── KeY Output ──

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

── KeY Output ───

`{\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

We can use the equation `f(f(c)) = f(c)` to simplify the term `f(f(f(c)))`, meaning we *apply* the equation to the `f(f(c))` subterm of `f(f(f(c)))`. 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(c))` subterm of `f(f(f(c)))`.[6] In the current system, there opens a context menu, allowing to select a taclet with the display name **applyEq**.[7]

Afterwards, the right-hand side equation has changed to `f(c) = f(f(c))`, 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(c) = f(f(c))`, and apply **eqSymm**. The resulting goal has two identical formulas on both sides of the sequent, so we *could* apply **close**. Just to demonstrate the other possibility, we undo the last rule application using **Goal Back** ↶, leading us back to the **Current Goal** `f(f(c)) = f(c),...==> f(c) = f(f(c))`.

---

[6] More detailed, we move the mouse over the "=" symbol, such that the whole of `f(f(c)) = f(c)` is highlighted. We click, hold, and move the mouse, over the second "`f`" in `f(f(f(c)))`, such that exactly the subterm `f(f(c))` gets highlighted. Then, we release the mouse.

[7] Possibly, there are more than one offered; for our example, it does not matter which one is selected.

   The other option is to apply the left-hand equation to `f(f(c))` on the right (via drag and drop). Afterwards, we have the *tautology* `f(c) = f(c)` on the right. By selecting that formula, we get offered the taclet **eqClose**, which transforms the equation into `true`. (If the reader's system does not offer **eqClose** in the above step, please make sure that **One Step Simplification** is unchecked in the **Options** menu, and try again.)

Closing 'by True' and 'by False'

So far, all goals we ever closed featured identical formulas 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 **closeTrue**, which is defined as:

—— Taclet ——————————————————————————————

```
closeTrue {
  \find ( ==> true )
  \closegoal
}
```

—————————————————————————————— 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 ——————————————————————————————

```
closeFalse {
  \find ( false ==> )
  \closegoal
}
```

—————————————————————————————— Taclet ——

This is actually a very important type of closable sequent. In many examples, a sequent can be proven by showing that the assumptions (i.e., the left-hand side formulas) 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" feature offered by the KeY prover to instantiate schema variables needed to apply a rule. 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 reconstruct the (in spirit) same proof, this time using such a dialogue explicitly.

After again loading the problem file `projection.key`, 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 ==> )
}
──────────────────────────────────────── 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, respectively, just by matching the highlighted sequent formula `\forall s x; f(f(x)) = f(x)` with the `\find` argument `\forall u; b`. The instantiation of t is, however, left open, to be chosen by the user. We can type c in the corresponding input field of the dialogue,[8] and click **Apply**. As a result, the `f(f(c)) = f(c)` 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.

Skolemizing Quantified Formulas

We will now consider a slight generalization of the theorem we have just proved. Again assuming that f is a projection, instead of showing `f(c) = f(f(f(c)))` for a particular c, 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 ─────────────────────────────────────
\forall s x; f(f(x)) = f(x) ==> \forall 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:[9]

```
───── KeY Output ─────────────────────────────────────
\forall s x;  f(f(x)) = f(x) ==> f(y_0) = f(f(f(y_0)))
──────────────────────────────────── KeY Output ─────
```

---

[8] 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.

[9] Note that the particular name y_0 can differ.

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. Therefore, y_0 can be nothing but a constant. Moreover, y_0 is a *new* symbol.

Eliminating quantifiers by introducing new constants is called *Skolemization* (after the logician Thoralf Skolem). In a sequent calculus, universal quantifiers (\forall) 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 (\forall) on the left, see our discussion of **allLeft** above. (It is instructive to look at all four cases in combination, see Chapter 2, Figure 2.1.)

Skolemization 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 that might show up during proving.

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

—— KeY Output ——————————————————————————————

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

————————————————————————————————— KeY Output ——

It tells us that the rule removes the quantifier matching \forall u;, and that (the match of) u is \substituted by the \newly generated Skolem constant sk in the remaining formula (matching b).

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

15.2.3 Exploring Programs in Formulas: 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 formulas (and subformulas) are evaluated. In particular, a formula and its subformulas 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 (mixfix) 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 formulas, $\langle p \rangle$ 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.[10] 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 formulas 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 formulas, but they can very well be proper DL formulas, 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. Concretely, p is a sequence of (zero, one, or more) Java statements. Accordingly, the logic is called JavaDL.

The following is an example of a JavaDL formula:

$$x < y \rightarrow \langle \texttt{t = x; x = y; y = t;} \rangle \, y < x \qquad (15.1)$$

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 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. This formula (15.1) 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 formulas, 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 JavaDL, please refer to Chapter 3.

---

[10] These descriptions have to be generalized when nondeterministic programs are considered, which is not the case here.

Feeding the Prover with a DL Problem File

The file `exchange.key` contains the JavaDL formula (15.1), in the concrete syntax used in the KeY system:[11]

─── KeY Problem File ───────────────────────────────────

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

─────────────────────────────── KeY Problem File ───

When comparing this syntax with the notation used in (15.1), we see that diamond modality brackets ⟨ and ⟩ are written as `\<{` and `}\>` within the KeY system. 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 JavaDL formulas with KeY.

Using the Prover as an Interpreter

We consider the file `executeByProving.key`:

─── 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

─────────────────────

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

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 JavaDL strategy (by clicking the play button ▶). 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 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, labeled with number **0:**, 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 formulas 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 **1:**), which 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.

Updates are part of the version of dynamic logic 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 disentangled from the manipulation of the program in the modality. This enables the calculus to perform *symbolic execution* in a natural way, and has been very fruitful contribution of the KeY project. First of all, updates allow proofs to symbolically execute programs in their natural direction, which is useful for proof inspection and proof interaction. Moreover, the update mechanism is heavily exploited when using the prover for other purposes, like test generation (Chapter 12), symbolic debugging (Chapter 11), as well as various analyzes for security (Chapter 13) or runtime verification [Ahrendt et al., 2015], to name a few.

Elementary updates in essence are a restricted kind of 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 Definition 3.8, Chapter 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 can further be considered as *explicit substitutions* that are yet to be applied. This viewpoint will get clearer further-on.

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 a single 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**, after the rule application of **emptyModality** in the current proof). Now that the modality is gone, the update `{i:=3 || j:=7}` is applied in form of a *substitution*, to the formula following the update, `p(i,j)`. The reader can follow this step when scrolling down the proof. Altogether, this leads to a delayed turning of program assignments into substitutions in the logic, as compared to other variants of DL (or of Hoare logic). We will return to the generation, parallelization, and application of updates on page 524.

Employing Active Statements

We now focus on the connection between programs in modalities on the one hand, and taclets on the other hand. For that, we load `updates.key`. When moving the mouse around over the single formula of the **Current Goal**,

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

we realize 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 color, 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 to be executed next. In the current example, this simply translates to the *first* statement.

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

```
─── Tooltip ──────────────────────────────────────────────────────────────

assignment {
  \find (
    \modality{#allmodal}{ ..
                                #loc=#se;
                          ... }\endmodality post
  )
  \replacewith (
    {#loc:=#se}
      \modality{#allmodal}{ ..  ... }\endmodality post
  )
}

                                                               ─ 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, in this case all #allmodal, including ⟨.⟩. in particular. 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* are not normally *active*. By *block* we mean both unlabeled and labeled Java blocks, as 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 an abrupt termination statement is active, like break,

`continue`, `throw`, or `return`. 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 labeled 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 nonblock (i.e., active) statement, whereas "`...`" matches the statements following the active statement, plus the corresponding closings of the opened blocks.[12]

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 formulas typically having many solutions. After all, it is the ability of handling symbolic values which makes theorem proving more powerful than testing. It allows us 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 c; }
\programVariables { int i, j; }
\problem {
        {i:=c}
           \<{ 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 `c` 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:=c}` is necessary because it would be illegal to have an assignment `i=c;` inside the modality, as `c` 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

---

[12] "`..`" and "`...`" correspond to π and ω, respectively, in the rules in Chapter 3.

respect to `c`. We get those by clicking on the play button and therefore running KeY's proof search strategy on this problem which results in `==> p(1+c,4+c)` being the remaining **OPEN GOAL**. This tells us that `1+c` and `4+c` 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' (see 15.2.3) method discussed above forms the principle of *symbolic execution*, which lies at the heart of the KeY prover.

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

—— KeY Problem File ——————————————————————————————————————

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

—— KeY Problem File ——

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 emulates the evaluation order exactly as it is specified in the Java language description by symbolic execution, and that the calculus does not allow any other evaluation order. We prove this formula automatically here.

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 subformula of a more complex DL formula. For instance in $\psi \rightarrow \langle p \rangle \varphi$, the diamond formula is below an implication (see also formula (15.1)). A DL subformula 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 ——

Please observe 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 rather a logical variable.

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 c, 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*.

We see here that, after only one proof step, the sequent is essentially not 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 formulas only.

Proving DL Problems with Program Variables

So far, most DL `\problem` formulas *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 formulas 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 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.

Demonstrating the Update Mechanism

In typical applications of the KeY prover, the user is not concerned with the update mechanism. Still, this issue is so fundamental for the KeY approach that we want to put the reader in the position to understand its basic principles. We do this by running an example in a much more interactive fashion than one would ever do for other purposes. (For a theoretical treatment, please refer to Section 3.4).

Let us reconsider the formula

$$x < y \rightarrow \langle \texttt{t = x; x = y; y = t;} \rangle y < x$$

and (re)load the corresponding problem file, `exchange.key` (see above 15.2.3) into the system. Also, we make sure that the "One Step Simplifier" button in the toolbar is unselected such that we can illustrate the update mechanism fully transparent.

The initial **Current Goal** looks like this:

—— KeY Output ———————————————————————————

```
==>
    x < y
 -> \<{ t=x;
        x=y;
        y=t;
      }\> y < x
```

——————————————————————————————— KeY Output ——

We prove this sequent interactively, just to get a better understanding of the basic steps usually performed by automated strategies.

We first apply the **impRight** rule on the single formula of the sequent. Next, the first assignment, `t=x;`, 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 `{t:=x}`[13] appeared in front of the modality. We perform the same step on the leading assignment `x=y;`. Afterwards, the sequent has the two subsequent updates `{t:=x}{x:=y}` leading the formula.

This is the time to illustrate a very essential step in KeY-style symbolic execution, which is *update parallelization*. A formula $\{u_1\}\{u_2\}\varphi$ says that φ is true after the *sequential* execution of u_1 and u_2. Update parallelization transforms the sequential steps (u_1 and u_2) into a *single*, parallel step $u_1 \,||\, u_2'$, leading to the formula $\{u_1 \,||\, u_2'\}\varphi$,

---

[13] 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.

where u_2' is the result simplifying $\{u_1\}\, u_2$, i.e., applying u_1 to the u_2. This will get clearer by continuing our proof in slow motion.

With the mouse over the curly bracket of the leading update, we select the rule **sequentialToParallel2**. (Its tooltip tells the same story as the previous sentences.) The resulting, parallel update is `{t:=x || {t:=x}x:=y}`. As t does not appear on the right side of `x:=y`, the parallel update can be simplified to `{t:=x || x:=y}`. (Parallelization is trivial for independent updates.) In the system, we select `{t:=x}x:=y`, and apply **simplifyUpdate3**. Then, by using the **assignment** rule a third time, we arrive at the nested updates `{t:=x || x:=y}{y:=t}` (followed by the empty modality). Parallelizing them (application of the rule **sequentialToParallel2**) results in the single parallel update `{t:=x || x:=y || {t:=x || x:=y}y:=t}`. Applying the rule **simplifyUpdate3** simplifies the rightmost of the three updates, `{t:=x || x:=y}y:=t` and removes `x:=y`, as it has no effect on `y:=t`.

Only now, when processing the resulting update `{t:=x}y:=t` further, we are at the heart of the update parallelization, the moment where updates turn from *delayed* substitutions to real substitutions. The reader can see that by applying the rule **applyOnElementary** on `{t:=x}y:=t`, and then **applyOnPV** (apply on Program Variable) on `{t:=x}t`. With that, our parallel update looks like `{t:=x || x:=y || y:=x}`. Its first element is not important anymore, as t does not appear in the postcondition `x < y`. It can therefore be dropped (**simplifyUpdate2** on the leading curly bracket). The reader may take a moment to consider the result of symbolic execution of the original Java program, the final update `{x:=y || y:=x}`. It captures the effect of the Java code `t=x;x=y;y=t;` (in so far as it is relevant for remainder for the proof) in a *single, parallel* step. The right-hand sides of the updates `x:=y` and `y:=x` are evaluated in the *same* state, and assigned to the left-hand sides at once.

With the empty modality highlighted in the **OPEN GOAL**, we can apply the rule **emptyModality**. It deletes that modality, and results in the sequent `x < y ==> {x:=y || y:=x}(y < x)`. When viewing the (parallel) update as a substitution on the succeeding formula, it is clear that this sequent should be true. The reader is invited to show this interactively, by using the rules **applyOnRigidFormula**, **simplifyUpdate1** and **applyOnPV** a few times, followed by **close**.

Let us stress again that the above demonstration serves the single purpose of gaining insight into the update mechanism. Never ever would we apply the aforementioned rules interactively otherwise. The reader can replay the proof automatically, with the One Step Simplifier switched on or off, respectively. In either case, the proof is quite a bit longer than ours, due to many normalization steps which help the automation, but compromise the readability.

Using Classes and Objects

Even though the DL problem formulas 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 (15.1) —————————————————————————————

```
\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 is getting older at its birthday(). As a side note, this is an example where
an update 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 ——

   When loading the file into the KeY system, the reader may recognize a difference
between the proof obligation given in the problem file and the initial proof obligation
in the KeY system:

—— KeY Output (15.2) —————————————————————————————————————

```
==>
 \forall int x;
   {heap:=heap[p.age := x]}
     (    x >= 0
      -> \<{ p.birthday();
           }\> p.age >  x)
```

——— KeY Output ——

Note that, in the display of the prover, the update {p.age:=x} from the problem
file is now written as {heap:=heap[p.age := x]}. Both updates are no different;
the first is an abbreviation of the second, using a syntax which is more familiar
to programmers. The expanded version, however, reveals the fact that this update,
whether abbreviated or not, changes the value of a variable named heap. We explain
this in the following, thereby introducing the representation of object states in KeY.

In the context of object-oriented programming, the set of all objects—including their internal state—is often referred to as the *heap*. This is an implicit data structure, in so far as it cannot be directly accessed by the programmer. Instead, it is implicitly given via the creation and manipulation of individual objects. However, in KeY's dynamic logic, the heap is an explicit data structure, and is stored in a variable called `heap` (or a variant of that name).[14] For the sake of clarity, we first discuss the abstract data type of heaps in a classical algebraic notation, before turning to KeY's concrete syntax shortly. Let us assume two functions *store* and *select* with the following type signature:

$$store : Heap \times Object \times Field \times Any \rightarrow Heap$$

$$select : Heap \times Object \times Field \rightarrow Any$$

store models the writing of a value (of *Any* type) to a given field of a given object, in a given heap. The result is a new heap. The function *select* looks up the value of a given field of a given object, in a given heap. The following axioms describe the interplay of *store* and *select*.

$$select(store(h,o,f,x),o,f) \doteq x$$

$$f \neq f' \vee o \neq o' \quad \rightarrow \quad select(store(h,o,f,x),o',f') \doteq select(h,o',f')$$

Please observe that we deliberately simplified these axioms for presentation. The real formalization has to distinguish select functions for different field types, has to check type conformance of *x*, and take special care of object creation. Please refer to Section 2.4.3 for a full account on this.

However, in the user interface of the KeY system, the above notation would give unreadable output for real examples. In particular, we would get deeply nested *store* terms during symbolic execution of a program (with one *store* per assignment to a field). Therefore, KeY uses the following, shorter syntax. Instead of *store(h,o,f,x)*, we write `h[o.f:=x]`, denoting a heap which is identical to `h` everywhere but at `o.f`, whose value is `x`. With that, a nested store like *store(store(h,o1,f1,x1),o2,f2,x2)* becomes `h[o1.f1:=x1][o2.f2:=x2]`, presenting the heap operations in their natural order. The *select* operation is also abbreviated. Instead of *select(h,o,f)*, we write `o.f@h`, denoting the access to `o.f` in heap `h`. With that, the above axioms become

$$\texttt{o.f@h[o.f:=x]} = \texttt{x} \tag{15.2}$$

$$\texttt{f} \neq \texttt{f'} \vee \texttt{o} \neq \texttt{o'} \quad \rightarrow \quad \texttt{o'.f'@h[o.f:=x]} = \texttt{o'.f'@h} \tag{15.3}$$

Please note that the symbol `:=` in `h[o.f:=x]` does not denote an update. Instead, it is part of the mix-fix presentation $_\sqcup[_\sqcup._\sqcup:=_\sqcup]$ of *store*. In particular, `h[o.f:=x]` does not, in itself, change `h`. Instead, it constructs a new heap that is (in most cases) different from `h`. An actual change to `h` has to be done extra, in an update like

---

[14] The object representation described here is implemented in KeY 2.0 onward, and significantly differs from the earlier object representation which was described in the first book about KeY [Beckert et al., 2007].

`h := h[o.f:=x]`. Only after that, h has a new value, given by applying *store* to the old value of h.

In proofs, during symbolic execution, KeY uses largely a specific heap variable called exactly `heap`, which is constantly modified in updates (resulting from assignments). There are some exceptions, however, where a proof node talks about more than one heap, for instance to distinguish the heap before and after execution of a method call. But as the one variable called `heap` dominates the picture, special shorthand notations are offered for this case. The select expression `o.f@heap` can be abbreviated by `o.f`, and the update `heap := heap[o.f:=x]` can be abbreviated by `o.f:=x`. Note that these abbreviations only apply to the single variable called exactly `heap`, not otherwise.

After this excursion on heap manipulation and presentation, let us look back to the KeY problem file `methodCall.key`, and KeY's presentation after loading the problem, see (15.2 from above). We now know that, in `methodCall.key`, the update `p.age:=x` abbreviates `heap := heap[p.age:=x]`, and that the postcondition `p.age > x` abbreviates `p.age@heap > x`. The first abbreviation was immediately expanded by KeY when loading the file, whereas the second one will be expanded later-on during the proof.

Calling Methods in Proofs

We now want to have a closer look at the way KeY handles method calls. We make sure that `methodCall.key` is (still) loaded and set the option **Arithmetic treatment** in the **Proof Search Strategy** tab to **Basic** and the option **Method treatment** to **Contract** or **Expand**. The reader is encouraged to reflect on the validity of the problem formula a little, before reading on.—Ready?—Luckily, we have a prover at hand to be certain, so we press the play button.

The strategy stops with the a number of **OPEN GOAL**s, one of them being `p = null, x_0 >= 0 ==>` [15]. 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`. Interpreting the **Proof** pane leads to the same reading. The first split, triggered by the taclet **methodCall**,

---

[15] Note that the particular index of the name x_0 can differ.

leads to two unclosed proof branches. The shorter one, marked as **Null Reference (p = null)**, leads immediately to an **OPEN GOAL** where the strategy gets stuck.

The file `methodCall2.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.

We now look at the first split in the proof (and click on the node before the split). 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 first inner node looks like this:

```
─── KeY Output (15.3) ──────────────────────────────────────
 x_0 >= 0
==>
 p = null,
 {heap:=heap[p.age:=x_0]}
   \<{ p.birthday()@Person;
     }\> p.age >= 1 + x_0
─────────────────────────────────────────── 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, see Section 3.7.1).

At this point, the strategy was ready to apply **methodBodyExpand**.[16] After that, the code inside the modality looks like this:

```
method-frame(source=birthday()@Person,this=p): {
  if (this.age >= 0) {
    this.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 `nonvoid` 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 Section 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 is

---

[16] This is the case even if **Method treatment** was chosen to be **Contract** instead of **Expand**. If no contract is available, the **Contract** strategy will still expand the method body.

`if (this.age>=0) {this.age++;}`. The rule **methodBodyExpand** has also intro-
duced the update `heapBefore_birthday:=heap`. This is necessary because, in
general, the formula succeeding the modality may refer to values that were stored in
the heap at the beginning of the method call. (An example for that is presented in
Section 15.3.) However, in the current proof, this update is simplified away in the
next step, because in the formula following the modality, there is no reference to
values in the heap from before calling the method.

Controlling Strategy Settings

The expansion of methods is among the more problematic steps in program verifica-
tion (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 midsize examples become infeasible to verify. These are good reasons
for giving the user more control over this crucial proof step.

KeY therefore allows the user to configure the automated strategies such that
they *refrain* from expanding methods automatically.[17] We try this out by loading
`methodCall2.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 (Figure 15.3). We can highlight the active
statement, apply first the taclet **methodCall**. After this step we *could* call **method-
BodyExpand** interactively. KeY would then *only* apply this very taclet, and stop
again.

Controlling Taclet Options

We draw out attention back to the proof of `methodCall2.key`. This proof 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
(**close**). It is, however, untypical that absence of null references can be derived so
easily. Often, the "null branches" complicate proofs substantially.

In the KeY system the handling of null references and other runtime exceptions
can be adjusted by setting *taclet options* We open the taclet option dialogue, via the
main menu **Options** → **Taclet options**. Among the option categories, we select the
runtimeExceptions, observe that **ban** is chosen as default, and change that by selecting
allow instead. Even if the effect of this change on our very example is modest, we
try it out, to see what happens in principle.[18] We then load `methodCall.key` and
push the play button. The proof search strategy stops with two open goals in the **Null
Reference (p = null)** branch. Allowing runtime exceptions in the KeY system results

---

[17] For a discussion of loop treatment, please refer to Chapter 3 and Section 16.3.

[18] Please note that changing back to default settings of KeY can be enforced by deleting the `.key`
directory in the user's home directory and restarting KeY.

in the treatment of these exceptions as specified in the Java language specification, i.e., that exceptions are thrown if necessary and have to be considered. KeY is able to not only consider explicit exceptions, such as throwing exceptions "by-hand," it is also able to map the behavior of the JVM, i.e., to treat implicit exceptions. The proof tree branches at the point where the strategy reaches the method call **p.birthday** in the modality. The branching of the proof tree results from the taclet **methodCall**. One branch deals with the case, that the object on which the method is called is nonnull and the other branch deals with the case that the object is null. Depending on the setting of the taclet option **runtimeException** the *null branch* representing the exceptional case in the proof looks different. At the moment we have set the option for runtime exceptions to allow. Therefore, in the *null branch* the method call in the modality is replaced by `throw new java.lang.NullPointerException ()`. So an exception is instantiated and thrown which allows the verified code to catch it and to continue execution in the exceptional case. In this case the exception has to be symbolically executed and it has to be proven that the postcondition also holds after the exception had occurred in the program.

Loading the same file with setting the option **runtimeException** to **ban** results in a proof stopping in the *null-branch* as well. If the user bans runtime exceptions in the KeY system, KeY treats any occurrence of a runtime exception as an irrecoverable program failure. The reader can reproduce this by comparing the node before the branching of the proof—into **Null Reference (p=null)** and **Normal Execution (p!=null)**—and the nodes after the split. In the node after the split the modality and the formula succeeding the modality (postcondition) in the succedent is fully replaced by false. This means that the program fails and therefore the postcondition will be false. If the succedent has more formulas than the modality and the postcondition, it is still possible to close the proof with the remaining parts of the sequent (in our case the context). The formula is replaced by false for two reasons. The first reason is that we do not want to take the runtime exceptions into account, therefore we replace the modality as well as the postcondition by false. Now the prover can not consider the case of an exception in a modality like it is the case in the option set to allow. Secondly, it makes the verification easier because the user and the prover do not have to deal with the symbolic execution of the implicit exception. For the remaining examples we switch the option **runtimeException** to **ban**.

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`. The 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 separate section on Java integers (Section 5.4), discussing the different variants in the semantics and the calculus. Please note that KeY 2.6 comes with the mathematical integer semantics chosen as default option, to optimize usability for beginners. However, for a sound treatment of integers, the user should switch to either of the other semantics.

15.3 Understanding Proof Situations

We have so far used simple toy examples to introduce the KeY system to the reader. However, the application area of the KeY system is verification of real-world Java programs, which are specified using the Java Modeling Language (JML). Proving the correctness of larger programs with respect to their specification can be a nontrivial task. s In spite of a high degree of automation, performing the remaining interactive steps can become quite complex for the user.

In this section we give some hints for where to search for the necessary information, and how to proceed the verification process.

We will introduce these hints on an example which will be described in more detail in Chapter 16.

There are several potential reasons why the automated strategy stops in a state where the proof is still open.

We first start with a simple case: the number of proof steps (adjustable in the slider in the **Proof Search Strategy** pane) is reached. In this case, one may simply restart the automated strategy by pressing the play button in the toolbar again and let the prover continue with the proof search. Or alternatively, first increase the number of maximal rule applications in the slider and then restart the strategy to try to continue with the automated proof search. This can already lead to a closed proof.

However, if incrementing the number of proof steps does not lead to a successful proof, one of the following reasons may be responsible for the automated strategy to stop:

- there is a bug in the specification, e.g., an insufficient precondition,
- there is a bug in the program
- the automated proof search fails to find a proof and

    - (some) rule applications have to be done manually,
    - or automated strategies have to be adjusted,
    - or both.

In the first two cases there is a mismatch between the source code and the specification, and the automated proof search strategy is not able to find a proof because there is none. Here the user has to review the source code and the specification in order to fix the bug.

In the third case we are limited by the proof complexity of dynamic logic. Here the user has to guide the prover by providing the right information, e.g., instantiations of quantifiers, such that the prover can carry on.

We cannot give a nostrum that would allow the user to decide which of the three cases is responsible for the prover to stop. (In fact, this case distinction is undecidable.) We rather reach a point in the interactive proof process where the user may have to understand aspects of the open goal in order to provide the right information or to identify mistakes in the program or in the specification. In the following, we give some hints for the comprehension of open goals.

The first step in understanding what happened during the proof process is to have a look at the proof tree. The user should start at the original proof obligation and follow the proof tree to the open goal(s). The *labels* at the nodes in the proof tree already give good hints what happened. The user may first draw the attention to the labels which are highlighted light-blue. These indicate the nodes where taclets have been applied that perform symbolic execution. Here the user gets an impression which point in the control flow of the program is presented in the open goal. Moreover, looking at *branching points* in the proof tree can give very useful insights. Looking closer into the node before the proof branches may give good hints about what is (supposed to be) proven in either of the branches.

Recall the example dealing with a method call (`methodCall.key`, KeY Problem File (15.1), page 526), where the proof splits into two branches: the case where the object on which the method is called is assumed to be not null (**p!=null** on the left side of the sequent, or, equivalently, **p=null** on the right side of the sequent) and the case where the object is assumed to be null (**p = null** on the left side of the sequent). The labels as well as the taclet applied to the node directly before the proof split give the information what has to be proven (i.e., in the mentioned example that the postcondition holds in both cases, *p* being null and *p* being not null).

The next step in understanding the proof situation is to take a closer look at the sequent of an open goal. First of all the sequent consists of a number of formulas. Depending on the progress of the symbolic execution of the program during proof construction and the original formula, there will also be a formula containing a modal operator and Java statements.

A good strategy is to first finish the symbolic execution of the program by letting the prover continue with the proof search on the branch with the open goal, such that the modality is removed from the sequent. This strategy is also implemented in the KeY system as so called macro proof step, which basically is a collection of proof steps and strategies and accessible by right-clicking onto the sequent arrow and selecting the context menu entry **Auto Pilot → Finish Symbolic Execution**.

If this task is successful, we are often left with a formula in pure first-order logic of which the validity has to be shown. However, this strategy does not always succeed. If the user is left with a sequent still containing a modal operator, the reader should be aware that the sequent remains in the prestate. This means that all formulas in the sequent refer to the state before executing the program. (But please observe that *sub*formulas, following updates or modalities, are evaluated in different states.)

When directly looking at the sequent of an open goal the user should also keep in mind the intuitive meaning of sequents: the left-hand side of the sequent is assumed and one of the right-hand side formulas has to be proven. As a special case, a sequent is valid if the left-hand side is contradictory, which may have to be exhibited by further proof steps.

The user should also keep in mind that $\Gamma \Rightarrow o = null, \Delta$ is equivalent to $\Gamma, o \neq null \Rightarrow \Delta$. This means that, instead of intuitively trying to prove $o = null$ or Δ, we can think of proving Δ under the assumption $o \neq null$, which is effectively the same. The reader may again recall an example from `methodCall.key`, where this was discussed earlier.

When the user is left with a pure first-order logic formula, it may be the case that parts of the invariants or the postcondition can not be proven. To identify those parts, there is a strategy which in many cases helps to get further insights. This strategy is also implemented as proof macro **Full Auto Pilot** and it is accessible by right-clicking onto the sequent arrow and selecting the context menu entry **Auto Pilot → Full Auto Pilot**. We will first describe how this method basically works and apply this method to an example afterwards.

After exhausting the automated strategy, the reader should split the proof interactively doing case distinctions of each conjunct of the postcondition using for example the taclet **andRight** or the cut rule. (This can also be achieved by using the proof macro **Propositional → Propositional Expansions w splits**.) After this case distinction each branch contains only one conjunct of the postcondition. Now the user should try to close each branch separately by either using the automated proof search strategy on each open goal or by applying the proof macro **Close provable goals below** to the node before the splits (right-clicking onto the node in the **Proof** pane in the proof tree and selecting the proof macro) The branches which do not close may not be provable and give hints on which part of the postcondition might be problematic.

For this we load the file `PostIncMod.java`, which is a slightly modified version of the first example in Chapter 16. For demonstration purposes we have incorporated a little mistake in the code or its specifications. For a detailed description of the example we point the reader to Chapter 16.

—— Java + JML ————————————————————————————

```java
public class PostIncMod{
  public PostIncMod rec;
  public int x,y;

  /*@ public invariant rec.x >= 0 && rec.y>= 0; @*/

  /*@ public normal_behavior
    @ requires true;
    @ ensures rec.x == \old(rec.y)+1 && rec.y == \old(rec.y)+1;
    @*/
  public void postInc(){
    rec.x = rec.y++;
  }
}
```

———————————————————————————————— Java + JML ——

The special Java comments `/*@ ... @*/` mark JML annotations in the Java code. The keyword `normal_behavior` states that the method `postInc()` terminates without throwing an exception. The method contract consists of a pre- and a postcondition. The meaning of the contract is that if the caller of the method fulfills the precondition, the callee guarantees the postcondition to hold after termination. In this example the precondition is `true` and the postcondition says that after the successful termination of the method the field `rec.x` is equal to the value of the field

rec.y before the method call (indicated by the keyword \old) increased by one. Similarly, the field rec.y is equal to the value of the field rec.y before the method call increased by 1. For a more detailed description of JML we point the reader to Chapter 7. The reader is encouraged to determine what the method postInc() performs.

When loading this file, the **Proof Management** dialogue will open. In its **Contract Targets** pane, we make sure that the folder **PostIncMod** (*not* **PostInc**) is expanded, and therein select the method **postInc()** we want to verify. We are asked to select a contract (in this case, there is only one), and press the **Start Proof** button. The reader may make sure that the One Step Simplifier is turned on, and start the automated proof search strategy. The prover will stop with one open goal where the modality is already removed.

The reader may now search for the node where the empty modality is about to be removed from the sequent (the last node on the open branch which is highlighted in light blue and labeled with **{ }**) and select that node. In the case at hand, the automated strategy searched a little too far, so we undo some automated rule applications in order to understand the case that could not be proved. For that we left-click on the *next* node in the proof tree (where the empty modality is removed), and select the context menu item **Prune Proof**. The open goal should now look similar to the following:

```
── KeY Output ─────────────────────────────────────────────────
 wellFormed(heap),
 self.<created> = TRUE,
 PostIncMod::exactInstance(self) = TRUE,
 measuredByEmpty,
 self.rec.x >= 0,
 self.rec.y >= 0
==>
 self.rec = null,
 self = null,
 {heapAtPre:=heap || exc:=null ||
  heap:=
     heap[self.rec.y:= 1 + self.rec.y][self.rec.x:=self.rec.y]}
   (self.rec.y = self.rec.x
   & self.rec.y@heapAtPre = -1 + self.rec.y
   & self.<inv>
   & exc = null)
                                          ── KeY Output ──
```

This is the point in the proof process where the prover has processed the entire Java method postInc(). The effects of the method execution are accumulated in the (parallel) update, which precedes the properties that must hold after postInc() (the formulas connected with &). To determine which of the parts of the postcondition does not hold (if any), we highlight the last formula of the sequent (by focusing the leading { of the update), and apply the rule **andRight**, which splits one of the

conjuncts. We repeat this step for as long as the is more than one conjuncts left.[19]
Now we have a closer look at the different sequents.

We start with the node whose last formula is:

─── KeY Output ──
```
{heapAtPre:=heap || exc:=null ||
 heap:=heap[self.rec.y:=1+self.rec.y][self.rec.x:=self.rec.y]}
   (self.rec.y@heapAtPre = -1 + self.rec.y)
```
── KeY Output ───

Focusing on (the leading { of) this formula, we apply the rule **One step Simplification**.
This will basically apply, and thereby resolve, the parallel update as a *substitution* on
the equation `self.rec.y@heapAtPre =-1 + self.rec.y(@heap)`. (Recall that
the field access `self.rec.y`, without @, abbreviates `self.rec.y@heap`). Therefore,
the last formula of the new sequent is

─── KeY Output ──
```
self.rec.y = -1 + self.rec.y@heap[self.rec.y:=1+self.rec.y]
                                  [self.rec.x:=self.rec.y]
```
── KeY Output ───

This formula states that the value of `self.rec.y(@heap)` is equal to `-1` plus the
value `self.rec.y` on a heap that is constructed from `heap` through the two given
store operations. It can be instructive for the reader to try to understand whether,
and why, this formula is true. One way to do that is to, *mentally*, apply the axiom
(15.3) (page 527), which removes the `[self.rec.x:=self.rec.y]`. Then apply
the axiom (15.2), which turns `self.rec.y@heap[self.rec.y:=1+self.rec.y]`
into `1+self.rec.y`. To prove this branch the reader may now left-click on the
sequent arrow and select the context menu entry **Apply rules automatically here**.

We now switch to the open goal with the following last formula:

─── KeY Output ──
```
{heapAtPre:=heap || exc:=null ||
 heap:=heap[self.rec.y:=1+self.rec.y][self.rec.x:=self.rec.y]}
   (self.rec.y = self.rec.x)
```
── KeY Output ───

We again apply the rule **One step Simplification** onto the shown formula. The new last
formula is

─── KeY Output ──
```
self.rec.y@heap[self.rec.y:=1+self.rec.y]
                [self.rec.x:=self.rec.y]
 =
```
────────────────────────────────────

---

[19] For postconditions with a lot of conjunctions this task can be tedious. Therefore, the KeY system
offers a proof macro called **Propositional Expansions w/ splits** which the user may apply
instead.

```
self.rec.x@heap[self.rec.y:=1+self.rec.y]
              [self.rec.x:=self.rec.y]
```
—————————————————————————————————— KeY Output ——

This formula says that, in a heap constructed from `heap` with the two given *stores*, the values of `self.rec.y` and `self.rec.x` are the same. This is not true, however. The user can see that by, again *mentally*, applying the axioms (15.3) and (15.2) to the left side of the equation, resulting in `1 + self.rec.y`, and axiom (15.2) to the right side of the equation, resulting in `self.rec.y`.

With this technique we have encountered a mistake in our postcondition. We should have stated `rec.x==\old(rec.y)` instead of `rec.x==\old(rec.y)+1` in the JML specification. The reason is that the postincrement expression (in the Java implementation) returns the old value. A corrected version of the problem is included in file `PostIncCorrected.java`. The reader is encouraged to load this file and use the automated strategy to prove the problem. For further examples on using the KeY system we point the reader to the tutorial chapter (Chapter 16).

15.4 Further Features

Besides the introduced features and mechanisms in this chapter, the KeY systems employs a variety of different features. In the following we will give a glimpse into some other useful features of KeY.

Employing External Decision Procedures

Apart from strategies, which apply taclets automatically, KeY also employs external decision procedure tools for increasing the automation of proofs. If formulas contain a lot of equations and inequations over terms that represent structures from different theories it can be a good idea to use SMT solvers instead of a full theorem prover. SMT solvers implement highly-efficient algorithms for deciding the satisfiability of formulas over specific theories, in contrast to full theorem provers, which are designed to work on many different domains. We refer to [Bradley and Manna, 2007] and [Kroening and Strichman, 2008] for a more detailed introduction and description of decision procedures and their applications.

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 by selecting the main menu item **Options → SMT Solvers**. We first load `generalProjection.key` and then choose **SMT solvers Options** via the main menu item **Options**. This opens the **Settings for Decision Procedure** dialogue. The user can now adjust general SMT options as well as settings for individual solvers.

In the **General SMT Options** pane, we can choose for instance the timeout for the SMT solvers. Timeouts are important when working with SMT solvers, as the

search process can last very long, without necessarily leading anywhere. Here we suggest using as a first step the default timeout settings. However, for more complex problems, it can be useful to increase the timeout, to give the solver a better chance to find a proof. For now we select the external decision procedure tool Z3[20] in the menu on the left-hand side in the dialogue. Now we are able to adjust some settings for the solver if needed, but for this example we leave the default settings and click **Okay**. In the tool bar the **Run Z3** button now appears and we can press it. This opens a dialogue which shows the application of Z3 and whether it was successful. In this case the dialogue says valid and the reader is now able to press the button **Apply**. 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 originally go beyond the scope of decision procedures.

Counterexample Generator

A feature that comes in handy when deciding whether a proof obligation is invalid is the counter example generator in KeY. This feature is accessible by pressing the toolbar button ❓ when a proof state is loaded. The mechanism translates the negation of the given proof obligation to an SMT specification and uses an SMT solver to decide the validity of this formula. To use this feature, the SMT solver Z3_CE has to be configured in the SMT solver options dialogue.

Model Search

If a sequent contains a lot of (in)equations, the KeY system offers the possibility to adjust the proof search strategy to systematically look for a model. This strategy is accessible via the **Proof Search Strategy** tab. It is a support for nonlinear inequations and model search. In addition, this strategy performs multiplication of inequations with each other and systematic case distinctions (cuts).

The method is guaranteed to find counterexamples for invalid goals that only contain polynomial (in)equations. Such counterexamples turn up as trivially unprovable goals. It is also able to prove many more valid goals involving (in)equations, but will in general not terminate on such goals.

---

[20] To use an external SMT solver it has to be installed beforehand and the path to the executable of the solver has to be set in the **Settings for Decision Procedure** dialogue.

Test Case Generation

Another feature of KeY is the automated generation of test cases, achieving high code coverage criteria by construction. This feature is called KeYTestGen. It constructs and analyses a (partial) proof tree for a method under test, extracts path conditions, generates test data, and synthesizes test code. This includes the generation of test oracles, or alternatively the usage of the OpenJML runtime checker. Test case generation is accessible by pressing the button ✿ right after starting a proof for the method under test. The usage and underlying principles of test generation with KeY are described in detail in Chapter 12. In particular, the 'Quick Tutorial' (Section 12.2) offers a quick introduction into the usage of KeYTestGen to a new user.

15.5 What Next?

In this chapter, we introduced the usage of the KeY prover, in parallel to explaining the basic artifacts used by KeY, the logic, the calculus, the reasoning principles, and so on. As we did not assume the reader to be familiar with any of these concepts prior to reading this text, we hope we have achieved a self contained exposition. Naturally, this imposed limits on how far we could go. The examples were rather basic, and discussed in depth. Demonstrating the usage of KeY in more realistic scenarios is not within the scope of this chapter. However, this book contains the tutorial 'Formal Verification with KeY' (Chapter 16), which lifts the usage of KeY to the next level. It discusses more realistic examples, more involved usage of the tool, and solutions to archetypal problems of verification. We therefore encourage the reader to not stop here, but continue to learn more about how KeY can be used for program verification.