# Chapter 4
# Proof Search with Taclets

**Philipp Rümmer and Mattias Ulbrich**

## 4.1 Introduction

The primary means of reasoning in a logic are *calculi*, collections of purely syntactic operations that allow us to determine whether a given formula is valid. Two such calculi are defined in Chapter 2 and 3 for first-order predicate logic and for dynamic logic (DL). Having such calculi at hand enables us, in principle, to create proofs of complex conjectures, using pen and paper, but it is obvious that we need computer support for realistic applications. Such a mechanized *proof assistant* primarily helps us in two respects: 1. The assistant ensures that rules are applied correctly, e.g., that rules can only be applied if their side-conditions are not violated, and 2. the assistant can provide guidance for selecting the right rules. Whereas the first point is a necessity for making calculi and proofs meaningful, the second item covers a whole spectrum from simple analyses to determine which rules are applicable in a certain situation to the complete automation that is possible for many first-order problems.

Creating a proof assistant requires formalizing the rules that the implemented calculus consists of. In our setting—in particular looking at calculi for dynamic logic—such a formalization is subject to a number of requirements:

- JavaDL has a complex syntax (subsuming the actual Java language) and a large number of rules: first-order rules, rules for the reduction of programs and rules that belong to theories like integer arithmetic. Besides that, in many situations it is necessary to introduce derived rules (*lemmas*) that are more convenient or that are tailored to a particular complex proof. This motivates the need for a language in which new rules can easily be written, rather than hard-coding rules as it is done in high-performance automated provers (for first-order logic). It is also necessary to ensure the soundness of lemmas, i.e., we need a mechanized way to reason about the soundness of rules.
- Because complete automation is impossible for most aspects of program verification, the formalization has to support interactive theorem proving. KeY provides a graphical user interface (GUI) that makes most rules applicable only using mouse clicks and drag and drop. This puts a limit on the complexity that a single

rule should have for keeping the required user interaction clear and simple, and it requires that rules also contain "pragmatic" information that describes how the rules are supposed to be applied. Accounts on the user interface in KeY are Chapter 15 and [Giese, 2004].

- The formalization also has to enable the automation of as many proof tasks as possible. This covers the simplification of formulas and proof goals, the symbolic execution of programs (which usually does not require user interaction) as well as automated proof or decision procedures for simpler fragments of the logic and for theories. The approach followed in KeY is to have global *strategies* that give priorities to the different applicable rules and automatically apply the rule that is considered most suitable. This concept is powerful enough to implement proof procedures for first-order logic and to handle theories like linear integer arithmetic or polynomial rings mostly automatically.

This chapter is devoted to the formalism called *taclets* that is used in KeY to meet these requirements. The concept of taclets provides a notation for rules of sequent calculi, which has an expressiveness comparable to the "textbook-notation" that is used in Chapters 2 and 3, while being more formal. Compared to textbook-notation, taclets inherently limit the degrees of freedom (nondeterminism) that a rule can have, which is important to clarify user interaction. Furthermore, an *application mechanism*—the semantics of taclets—is provided that describes when taclets can be applied and what the effect of an application is.

Historically, taclets have first been devised by Habermalz [2000b,a] under the name "Schematic Theory Specific Rules," with the main purpose of capturing the axioms of theories and algebraic specifications as rules. The language is general enough, however, to also cover all rules of a first-order sequent calculus and most rules of calculi for dynamic logic. The development of taclets as a way to build interactive provers was influenced to a large degree by the theorem prover InterACT [Geisler et al., 1996], but also has strong roots in more traditional methods like tactics and derived rules that are commonly used for higher-order logics (examples for such systems are Isabelle/HOL, see [Nipkow et al., 2002], Coq, see [Dowek et al., 1993], or PVS, see [Owre et al., 1996]). Compared to tactics, the expressiveness of taclets is very limited, for the reasons mentioned above. A further difference is that taclets do not (explicitly) build on a small and fixed set of primitive rules, as tactics do in (foundational) higher-order frameworks like Isabelle. It nevertheless is a good idea to add comments in files containing taclets that signal which are meant to be axioms and which are derived rules that require a proof from the axioms. This has, e.g., been consistently done for the data type of finite sequences, see Section 5.2.

### *4.1.1 Purpose and Organization of this Chapter*

The purpose of this chapter is twofold: on the one hand, it provides new KeY users an introduction to the way calculus rules are implemented in the KeY system; on

the other hand, it is a reference manual of the taclet formalism, targeting more experienced users as well as developers. The main sections of the chapter are:

- *A taclet tutorial* (Section 4.2): a high-level overview of the most important features provided by the taclet language, and the methodology how taclets are used for introducing new theories.
- *The taclet reference manual* (Section 4.3): a detailed description of the taclet language, and its semantics.
- *Reasoning about the soundness of taclets* (Section 4.4): techniques to mechanically prove the soundness of taclets, by deriving a formula representation of the logical content of a taclet.

## 4.2 A Taclet Tutorial

The next pages give a tour through the taclet language and illustrate the most important taclet features by means of a case study. Taclets are used in the KeY system for multiple purposes: for the definition of first-order calculus rules, for the rules of the JavaDL calculus, to introduce data types and decision procedures, and to give users the possibility to define and reason about new logical theories. Users typically encounter taclets in the context of the last scenario, which is why our tutorial will describe the introduction of a new theory in KeY: we consider a simplified version of a theory of *lists,* and refer the reader to a more complete and practical version of *finite sequences* in Chapter 5 .

Theories are introduced by declaring a vocabulary of *types* and (interpreted) *functions,* a set of basic *axioms* defining the semantics of the theory, as well as a set of *derived rules* that are suitable for the construction of actual proofs. Both axioms and derived rules are formulated as taclets in KeY, with the difference that axioms are assumed and cannot be proven to be correct, while derived rules logically follow from the axioms.

### *4.2.1 A Basic Theory of Lists*

We will work with a simple data structure of lists resembling the data type found in Lisp and functional programming languages. The core theory, in the following sections denoted by $T_{List}$, is defined through a type *List*; elements of the data type are generated by two *constructor* symbols, *nil* and *cons*, representing the empty list and extension of a list by adding a new head element respectively. For simplicity we consider only elements of type *int* here. The theory of lists is in no way affected by the type of its elements.

$$nil : List$$
$$cons : int \times List \to List$$

For instance, the sequence $\langle 3,5,-2\rangle$ of integers will be represented through the term $cons(3,cons(5,cons(-2,nil)))$.

The symbols *nil* and *cons* are the only constructors of lists, and lists furthermore represent a *free algebraic data type,* which implies that:

- every list can be represented as a term only consisting of *nil* and *cons*, and possibly functions needed to construct the list elements (the first argument of *cons*);
- the representation of a list using *nil* and *cons* is *unique* assuming a unique representation of the integers.

Those properties are typically expressed with the help of *axioms*, which eliminate all interpretations of the constructor symbols that are inconsistent with the two properties. Axioms are formulas that are *assumed* to hold in all considered interpretations of the theory symbols; there is no way to prove that axioms are correct, since they are independent assumptions and cannot be derived from any other rules or axioms of the logic. The *consistency* of the axioms can be shown by defining a model in which all axioms are true: one such model is obtained by considering the set of ground terms over the constructors *nil* and *cons* and a unique representation of all integers as universe, and interpreting *nil* and *cons* and all integer ground terms as themselves. More details are given in Chapter 5 on theories. For our core theory of lists, we need three axioms:

$$\big(\phi[l/nil] \land \forall List\ l;\ \forall int\ a;\ \big(\phi \to \phi[l/cons(a,l)]\big)\big) \to \forall List\ l;\ \phi \tag{4.1}$$
$$\forall List\ l;\ \forall int\ a;\ \big(nil \mathrel{\dot{\neq}} cons(a,l)\big) \tag{4.2}$$
$$\forall List\ l_1,l_2;\ \forall int\ a_1,a_2;\ \big(cons(a_1,l_1) \doteq cons(a_2,l_2) \to a_1 \doteq a_2 \land l_1 \doteq l_2\big) \tag{4.3}$$

The axiom (4.1) reflects the assumption that any element of the list data type can be constructed using the symbols *nil* and *cons*. It can be shown that this assumption cannot precisely be captured using an axiom in first-order logic, it can only be approximated using weaker formulas, for instance using the induction axiom (4.1) shown here. The formula represents an *axiom schema,* since it is formulated with the help of a schematic variable $\phi$ that stands for an *arbitrary* formula that has to be chosen when using the axiom in a proof; this formula $\phi$ will usually contain the free variable $l$ of type *List*.

In other words, (4.1) should be read as an infinite set of first-order formulas, one for each possible choice of the symbol $\phi$. The axiom schema introduces an induction principle for lists, resembling the one for natural numbers (nonnegative integers) defined in Section 2.4.2: if it is possible to show that some formula $\phi$ holds for the empty list $l = nil$ (denoted by the substitution $[l/nil]$ replacing every occurrence of $l$ with *nil*), and that $\phi$ implies that also $\phi[l/cons(a,l)]$ holds for any $a$, then it can be concluded that $\phi$ holds *for all* lists $l$.

Taclet

```
\sorts {
   List;
}

\functions {
   \unique List nil;
   \unique List cons(any, List);
}

\axioms {
   list_induction {
      \schemaVar \formula phi;
      \schemaVar \variable List lv;
      \schemaVar \variable any  av;

      \find( ==> \forall lv; phi )
      \varcond(\notFreeIn(av, phi))

      \replacewith( ==> {\subst lv; nil} phi
                     & \forall lv; \forall av;
                           (phi -> {\subst lv; cons(av, lv)}phi) )
   };
}
```

Taclet

**Figure 4.1** Vocabulary and induction axiom for a simplified theory of lists

Induction axioms are relevant for all theories that are assumed to be generated by some set of function symbols, in the sense that all elements can be written as terms over this set of functions. In particular, every algebraic data type (an example of which are lists) comes with a predefined induction axiom similar to (4.1).

The axioms (4.2) and (4.3) represent uniqueness of the representation of a list using *nil* and *cons*. (4.2) expresses that the ranges of *nil* and *cons* do not overlap, whereas (4.3) states that *cons* is an injective function; in combination, the axioms imply that two lists are equal only if they contain the same number of elements, and the elements coincide.

### *4.2.2 The List Theory in Concrete Syntax*

We now explain how the theory $T_{List}$ of lists (as introduced so far) can be modeled in the concrete syntax of the KeY system. We first use declarations and taclets in order to model the vocabulary and axioms of the theory in a direct way, and then describe how further rules can be derived to make the theory more convenient to work with in practice. Derived rules are also essential for automating the construction of proofs.

The taclet syntax is explained in Section 4.3.1, and a complete description of the KeY syntax is given in Appendix B.

Figure 4.1 shows the *List* type, the function symbols *nil* and *cons*, as well as the induction axiom (4.1) in taclet syntax. The `\sorts` block is used to declare the types of the theory, whereas `\functions` contains the declaration of available function symbols and their signature (the result type and the type of arguments), in syntax inspired by Java. The declarations and definitions would normally be placed in the beginning of a KeY problem file, and can then be used for formulating and proving formulas involving $T_{List}$; the concrete steps to do this are described in Chapter 15, and later chapters of the book.

Figure 4.1 also captures the two axioms (4.2)–(4.3) of lists. KeY provides a built-in keyword for specifying the uniqueness of functions, so that the axioms (4.2) and (4.3) do not have to be written by hand; it suffices to add the flag `\unique` in the function declarations. A function declared to be `\unique` is injective, and the values of two distinct `\unique` functions are never equal. The `\unique` flag implies that KeY will internally generate (and automatically apply) rules that capture those assumptions.

The `\rules` block contains the taclet list_induction representing the induction axiom (4.1). Operationally, the rule list_induction is applied to an existing formula $\forall List\ l;\ \phi$, and replaces this formula with $\phi[l/nil]\ \wedge\ \forall List\ l;\ \forall int\ a;\ \big(\phi \rightarrow \phi[l/cons(a,l)]\big)$:

$$\text{list\_induction}\ \frac{\Gamma \Longrightarrow \phi[l/nil] \wedge \forall List\ l;\ \forall int\ a;\ \big(\phi \rightarrow \phi[l/cons(a,l)]\big), \Delta}{\Gamma \Longrightarrow \forall List\ l;\ \phi, \Delta}$$

In order to specify this transformation, the taclet uses a number of features of the taclet language, which are explained in the following paragraphs.

- `\find` defines a pattern that must occur in the sequent to which the taclet is supposed to be applied. In this taclet, the pattern `==> \forall lv; phi` matches on quantified list formulas in the succedent of a sequent; accordingly, list_induction can be applied whenever such a quantified formula turns up in a proof goal. The expression matched by `\find` is called the *focus* of a taclet application.
- `\replacewith` tells how the focus of the taclet application will be altered: a new proof goal is created from the previous one by replacing the expression matched in the `\find` part with the expression in the `\replacewith` part.

For list_induction, the quantified list formula in the succedent will be replaced by the somewhat complicated expression after the arrow `==>`; upon closer inspection, it can be seen that the expression indeed represents the conjunction $\phi[l/nil]\ \wedge\ \forall List\ l;\ \forall int\ a;\ \big(\phi \rightarrow \phi[l/cons(a,l)]\big)$. The operator `{\subst x; t}` expresses substitution of a variable $x$ with a term $t$.

*Note 4.1.* The keywords of the taclet language reflect the direction in which sequent calculus proofs are constructed: we start with a formula that is supposed to be proven and create a tree upwards by *analyzing* the formula and taking it apart. Taclets describe expansion steps (or, as a border case, closure steps), and by the *application*

of a taclet we mean the process of adding new nodes to a leaf of a proof tree following this description.

The taclet illustrates a further important feature of the taclet language, namely the use of *schema variables* in order to create flexible rules that can be instantiated in many concrete ways. The taclet list_induction contains three such schema variables, `phi`, `lv`, and `av`. Every schema variable is of a certain *kind,* defining which expressions the variable can stand for (a precise definition is given in Section 4.3.2). In our example, `phi` represents an arbitrary formula, while `lv` represents bound variables of type *List*, and `av` bound variables of type *int*. The possible valuations of schema variables are controlled with the help of *variable conditions,* and the `\varcond` clause in list_induction:

- `\varcond` specifies conditions that have to hold for admissible instantiations of the schema variables of a taclet. The condition `\notFreeIn` in list_induction, in particular, expresses that the bound variable `av` must not occur as a free variable in the formula `phi`.

Note that some, but not all occurrences of the schema variable `phi` in the rule list_induction are in the scope of a quantifier binding `av`. Without the variable condition `\notFreeIn(av, phi)` it would be ambiguous whether `av` is allowed to occur in `phi` or not.

*Example 4.2.* We illustrate how the rule list_induction can be used to prove a theorem in our theory $T_{List}$, the fact that every list is constructed using either *nil* or *cons*:

$$\forall List\ l;\ \big(l \doteq nil \vee \exists List\ m, int\ b;\ l \doteq cons(b,m)\big) \tag{4.4}$$

For this, we apply the sequent calculus notation introduced in Section 2.2.2. This sentence already has the shape of the formula `\forall lv; phi` in the `\find` part of the taclet list_induction, so that the taclet can directly be applied; this has been done in step $(*)$ in the proof in Figure 4.2. It should be noted, however, that inductive proofs often require appropriate *strengthening* of the formula to be proven: in order to show that $\forall x; \phi$ is a theorem, first a formula that implies $\forall x; \psi$ is introduced using the cut rule, and proven by means of induction. Luckily, no such strengthening is necessary in the example at hand.

When applying list_induction at $(*)$, all schema variables occurring in the taclet have to be instantiated with concrete syntactic objects: the variable `lv` is mapped to the bound variable $l$, the variable `av` to the (fresh) variable $a$, and the formula variable `phi` to the body $l \doteq nil \vee \exists List\ m, int\ b;\ l \doteq cons(b,m)$. When constructing the proof in the KeY system, the tool is able to determine those instantiations automatically, it is only necessary to tell KeY to apply list_induction to the formula (4.4) in the antecedent of the proof goal.

The rest of the proof can be constructed in a comparatively straightforward way (and can in fact be found automatically by KeY). At $(**)$, it can be observed that $nil \doteq nil$ holds, so that the whole conjunct $nil \doteq nil \vee \exists List\ m, int\ a;\ nil \doteq cons(a,m)$ can be reduced to *true* and eliminated. Finally, at $(***)$, we can observe that the

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\cdots \Longrightarrow cons(d,c) \doteq nil,\ cons(d,c) \doteq cons(d,c)}\,(*)
}{\cdots \Longrightarrow cons(d,c) \doteq nil,\ \exists List\ m, int\ b;\ cons(d,c) \doteq cons(b,m)}\,(***)
}{c \doteq nil \vee \exists List\ m, int\ b;\ c \doteq cons(b,m) \Longrightarrow cons(d,c) \doteq nil \vee \exists List\ m, int\ b;\ cons(d,c) \doteq cons(b,m)}
}{\Longrightarrow \begin{array}{l}(c \doteq nil \vee \exists List\ m, int\ b;\ c \doteq cons(b,m)) \\ \quad \rightarrow (cons(d,c) \doteq nil \vee \exists List\ m, int\ b;\ cons(d,c) \doteq cons(b,m))\end{array}}
}{\Longrightarrow \begin{array}{l}\forall List\ l;\ \forall int\ a;\ \big( \\ \quad (l \doteq nil \vee \exists List\ m, int\ b;\ l \doteq cons(b,m)) \\ \quad \rightarrow (cons(a,l) \doteq nil \vee \exists List\ m, int\ b;\ cons(a,l) \doteq cons(b,m))\big)\end{array}}\,(**)
}{\begin{array}{l}\big(nil \doteq nil \vee \exists List\ m, int\ a;\ nil \doteq cons(a,m)\big) \\ \wedge \forall List\ l;\ \forall int\ a;\ \big( \\ \quad (l \doteq nil \vee \exists List\ m, int\ b;\ l \doteq cons(b,m)) \\ \quad \rightarrow (cons(a,l) \doteq nil \vee \exists List\ m, int\ b;\ cons(a,l) \doteq cons(b,m))\big)\end{array} \Longrightarrow}\,(*)
}{\Longrightarrow \forall List\ l;\ \big(l \doteq nil \vee \exists List\ m, int\ b;\ l \doteq cons(b,m)\big)}
$$

**Figure 4.2** Inductive example proof

existentially quantified variables $m, b$ can be instantiated with the terms $c$ and $d$, respectively, concluding the proof.

### 4.2.3 Definitional Extension of the List Theory

At this point we have a fully defined, albeit very minimalist theory $T_{List}$ of lists available, which could in principle be used to state and prove conjectures about lists in KeY, or to reason about programs operating on lists or sequences. Most practical applications require a richer set of operations on lists, however; a general strategy to introduce such operations, without putting the consistency of the theory at risk, is known as *definitional extension,* and proceeds by introducing further functions or predicates over lists, and defining their intended meaning through recursive axioms according to the list constructors. Again, for more details we refer to the dedicated Chapter 5 on theories.

In the scope of our taclet tutorial, we consider two defined functions for computing *length* and *concatenation* of lists; the resulting extension of $T_{List}$ will be denoted by $T^{LA}_{List}$, and include the following additional function symbols:

$$length : List \rightarrow Int$$
$$append : List \times List \rightarrow List$$

The semantics of the functions can be formulated by simple recursion over one of the *List* arguments of each function, in mathematical notation leading to equations as follows:

Taclet

```
\functions {
   int length(List);
   List append(List, List);
}

\axioms {
   length_nil {
      length(nil) = 0
   };

   length_cons {
      \forall List l; \forall any a; length(cons(a, l)) = 1 + length(l)
   };

   append_nil  {
      \schemaVar \term List l;
      \find( append(nil, l) )
      \replacewith( l )
   };

   append_cons {
      \schemaVar \term any a;
      \schemaVar \term List l1;
      \schemaVar \term List l2;
      \find( append(cons(a, l1), l2) )
      \replacewith( cons(a, append(l1, l2)) )
   };
}
```

Taclet

**Figure 4.3** Vocabulary and axioms for defined list functions

$$length(l) = \begin{cases} 0 & \text{if } l = nil \\ length(l') + 1 & \text{if } l = cons(a, l') \end{cases} \tag{4.5}$$

$$append(l_1, l_2) = \begin{cases} l_2 & \text{if } l_1 = nil \\ cons(a, append(l_1', l_2)) & \text{if } l_1 = cons(a, l_1') \end{cases} \tag{4.6}$$

The corresponding declarations and axioms in KeY syntax are shown in Figure 4.3. The definitions can again be put in a KeY problem file, normally right after the definitions from Figure 4.1, and extend the basic list theory with the two additional functions *length* and *append* (as a technical detail, it is indeed necessary that the new axioms appear textually *after* the declarations of the constructors *nil* and *cons* in the KeY file, since KeY adopts a single-pass parsing approach).

Figure 4.3 illustrates that axioms can be written in two different styles. The first two axioms length_nil and length_cons are formulated as (quantified) formulas, and closely capture the recursive equation (4.5). When applying either rule in a proof, the

KeY prover will add the given formula to the antecedent of a proof goal; afterwards, quantifiers in the formula can be eliminated by instantiating with ground terms occurring in the goal, and the resulting equation can be used for equational rewriting. An example showing the rules is the following proof; after application of length_nil and length_cons, the proof can be closed using equational and arithmetic reasoning (not shown here):

$$\dfrac{\dfrac{\dfrac{*}{length(cons(1,nil)) \doteq 1 + length(nil), length(nil) = 0 \Longrightarrow length(cons(1,nil)) \doteq 1}\,\text{length\_nil}}{length(cons(1,nil)) \doteq 1 + length(nil) \Longrightarrow length(cons(1,nil)) \doteq 1}}{\forall List\ l.\ \forall int\ a.\ length(cons(a,l)) \doteq 1 + length(l) \Longrightarrow length(cons(1,nil)) \doteq 1}\,\text{length\_cons}$$

In contrast, the axioms append_nil and append_cons are formulated in a similar operational style as the induction axiom in Figure 4.1; the main difference to the induction axiom is the fact that `\find` expressions in Figure 4.3 are no longer *sequents* but *terms* (they do not contain an arrow `==>`). Rules of this form are called *rewriting taclets* in the KeY terminology, and represent transformations that modify subexpressions (either a formula or a term) of arbitrary formulas in a proof, both in the antecedent and succedent, leaving the surrounding formula unchanged. For instance, the rule append_nil can be used to rewrite any term $append(nil,l)$ to the simpler expression $l$, and rule append_cons is applicable to any expression of the form $append(cons(a,l_1),l_2)$. An example proof is:

$$\dfrac{\dfrac{\dfrac{*}{\Longrightarrow cons(a,l) \doteq cons(a,l)}\,\text{append\_nil}}{\Longrightarrow cons(a,append(nil,l)) \doteq cons(a,l)}\,\text{append\_cons}}{\Longrightarrow append(cons(a,nil),l) \doteq cons(a,l)}$$

Compared to the declarative style of length_nil and length_cons, the taclets append_nil and append_cons have both advantages and disadvantages: in particular, rewriting taclets are usually a lot more convenient to apply when constructing proofs interactively, since expressions can be simplified (or "evaluated") with only a few mouse clicks, in contrast to the multiple rule applications needed when using axiom length_nil. In addition, the rewriting taclet append_nil also captures the direction in which the corresponding equation `append(nil, l) = l` should be applied, namely rewriting the more complicated left-hand side to the simpler right-hand side, and can therefore also be applied automatically by the KeY system (see Section 4.3.1.10).

On the other hand, since the rules length_nil and length_cons are closer to the recursive mathematical formulation, the introduction of axioms in this style tends

to be less error-prone. In the exceedingly rare case that a user wants to rewrite `l` to `append(nil, l)` when constructing a proof, append_nil is actually less practical than the axiom length_nil, since the simple equation introduced by the latter rule can be applied in both directions. Rewriting from right to left is still possible even with append_nil, however, by means of introducing a *cut* in the proof.

### 4.2.4 Derivation of Lemmas

An important feature of the taclet language, and of the KeY prover, is the ability to easily add further *derived* rules to a theory. Such rules represent lemmas that logically follow from the theory axioms, and can help structure proofs because the lemmas can be proven once and for all, and later be applied repeatedly for proving theorems. The number of derived rules often exceeds the number of axioms of a theory by far: to reduce the risk of inconsistencies, the set of axioms is usually kept minimalist, whereas any number of derived rules can be added for reasons of convenience. The soundness of derived rules can be verified using the same calculus as for proving theorems, by translating taclets to *meaning formulas* that capture the logical content of a rule (see Section 4.4).

A small selection of derived rules for the theory $T^{LA}_{List}$ is shown in Figure 4.4; many more relevant lemmas exist. The first difference to earlier taclets is the fact that rules are now formulated within a `\rules` block, and no longer as `\axioms`, to indicate that the rules are lemmas.[1] The definitions from Figure 4.4 can again be put in a KeY problem file, either after the contents of Figures 4.1 and 4.3, or in a separate file that KeY users can load on demand. In the latter case, the KeY system will request that the soundness of the newly introduced rules is immediately justified by showing that their meaning formula is valid (Section 4.4).

The rewriting rules length_nil_rw and length_cons_rw in Figure 4.4 are operational versions of the axioms length_nil and length_cons, and correspond to the way the axioms append_nil and append_cons are written. Since rewriting rules are usually more convenient than axioms in the form of formulas, as illustrated by the examples in the previous section, length_nil_rw and length_cons_rw are useful derived rules; their correctness is directly implied by the theory axioms, of course.

The rule append_nil_right captures the fact that *nil* is also a right-neutral element for concatenation *append*, and complements the axiom append_nil. The soundness of append_nil_right has to be shown by induction over the first argument of *append*, with the help of the axiom list_induction. Similarly, length_append expresses that *length* distributes over concatenation, and can be proven correct by induction over the first argument of *append*. Both append_nil_right and length_append are rules that are frequently needed when proving theorems over lists, and present in every self-respecting list theory.

[1] In built-in rules of the KeY system, moreover the annotation `\lemma` can be added in front of a rule to indicate that a correctness proof has been conducted; the proof will then be checked automatically during regression testing.

Taclet

```
\rules {
   length_nil_rw {
      \find( length(nil) )
      \replacewith( 0 )
   };

   length_cons_rw {
      \schemaVar \term any a;
      \schemaVar \term List l;
      \find( length(cons(a, l)) )
      \replacewith( 1 + length(l) )
   };

   append_nil_right {
      \schemaVar \term List l;
      \find( append(l, nil) )
      \replacewith( l )
   };

   length_append {
      \schemaVar \term List l1, l2;
      \find( length(append(l1, l2)) )
      \replacewith( length(l1) + length(l2) )
   };

   length_cons_assume {
      \schemaVar \term List l, l1;
      \schemaVar \term any  a;
      \assumes( l = cons(a, l1) ==> )
      \find( length(l) ) \sameUpdateLevel
      \replacewith( 1 + length(l1) )
   };

   list_ctor_split {
      \schemaVar \term       List l;
      \schemaVar \skolemTerm List skl;
      \schemaVar \skolemTerm any  ska;
      \find( l ) \sameUpdateLevel
      \varcond( \new(ska, \dependingOn(l)), \new(skl, \dependingOn(l)) )
      \replacewith( nil )            \add( l = nil ==> );
      \replacewith( cons(ska, skl) ) \add( l = cons(ska, skl) ==> )
   };
}
```

Taclet

**Figure 4.4** Derived taclets for the list theory $T^{LA}_{List}$

The rules length_cons_assume and list_ctor_split are more sophisticated, and show several further features of the taclet language. The rule length_cons_assume is similar to length_cons_rw, but is (also) applicable to list terms that are not of the

form $cons(a,l)$: replacing $length(l)$ with $1+length(l_1)$ is admissible provided that the equality $l \doteq cons(a,l_1)$ holds for some element $a$. This can be expressed using the keyword `\assumes`:

- `\assumes` imposes a condition on the applicability of a taclet, and has a sequent as parameter. In the case of length_cons_assume, the `\assumes` clause states that the taclet must only be applied if an equation $l \doteq cons(a,l_1)$ appears in the antecedent of a goal (the sequent may contain further formulas).
- `\sameUpdateLevel` is a *state condition* that can be added to rewriting taclets, and is relevant in the case of taclets in JavaDL proofs. The flag ensures that the focus of the taclet application (the term that is represented by $length(l)$ in rule length_cons_assume) does not occur in the scope of modal operators apart from updates. Updates are allowed above the focus, but in this case the equation $l \doteq cons(a,l_1)$—or, more generally, all formulas referred to using `\assumes`, `\replacewith` and `\add`—have to be in the scope of the same update.

This flag `\sameUpdateLevel` is necessary to ensure the soundness of the rule length_cons_assume in dynamic logic, and in fact required for most rewriting rules that contain any `\assumes` or `\add` clauses. In order to illustrate the effect of `\sameUpdateLevel`, we consider two potential applications of length_cons_assume:

Illegal:

$$\frac{v \doteq cons(a,w) \Longrightarrow \{w := nil\}p(1+length(w))}{v \doteq cons(a,w) \Longrightarrow \{w := nil\}p(length(v))}$$

Legal:

$$\frac{\{w := nil\}(v \doteq cons(a,w)) \Longrightarrow \{w := nil\}p(1+length(w))}{\{w := nil\}(v \doteq cons(a,w)) \Longrightarrow \{w := nil\}p(length(v))}$$

The first application of length_cons_assume has to be ruled out, and is prevented by KeY in the presence of the `\sameUpdateLevel` flag, since the application would incorrectly move the term $w$ into the scope of the update $w := nil$ redefining the meaning of $w$: in the equation $v \doteq cons(a,w)$, the term $w$ represents arbitrary lists, whereas the update defines $w$ to denote the empty list. The application illustrates the case of a symbol changing its meaning due to the presence of modal operators.[2] The second application of length_cons_assume is correct and possible in KeY, because all formulas involved are in the scope of the same update.

The rule list_ctor_split enables users to introduce case splits for arbitrary list expressions $l$ in a proof: either such an expression has to denote an empty list ($l = nil$), or the list must have length at least one and can be represented in the

[2] It should be noted, however, that KeY will usually apply updates immediately and thus simplify the formula $\{w := nil\}p(length(v))$ to $p(length(v))$; the illegal situation shown here therefore requires some mischievous energy to construct in an actual proof. Rule length_cons_assume without flag `\sameUpdateLevel` would be unsound nevertheless.

form $l = cons(a, l_1)$. In sequent notation, such case splits can be described using the following rule, in which $c, d$ are required to be fresh constants, and $\phi[l]$ expresses that the list expression $l$ occurs anywhere in the conclusion:

$$\frac{\Gamma, l \doteq nil \Longrightarrow \phi[nil], \Delta \qquad \Gamma, l \doteq cons(c,d) \Longrightarrow \phi[cons(c,d)], \Delta}{\Gamma \Longrightarrow \phi[l], \Delta}$$

In contrast to all taclets shown up to this point, list_ctor_split contains two *goal templates* separated by a semicolon `;`, each with one `\replacewith` and one `\add` clause, corresponding to the two cases (or premises) to be generated when applying the rule. The `\add` clauses take care of adding the equations $l = nil$ and $l = cons(a, l_1)$ in the antecedent:

- `\add` specifies formulas that are added to a sequent when the taclet is applied. The argument of `\add` is a sequent with the formulas to be added to the antecedent and the succedent, respectively.

The taclet also states, by means of the variable condition `\new`, that `ska` and `skl` have to be instantiated with fresh *Skolem symbols* each time the taclet is applied. The correctness of list_ctor_split can again be proven by means of induction: the meaning formula of the taclet is essentially the formula discussed in Example 4.2.

Derived taclets can be used not only to augment user-defined theories, but also for all built-in data types and logics of the KeY system: for each proof to be constructed, a set of tailor-made taclets can be loaded into the system. The soundness of the derived taclets has to be shown as outlined before, by first producing a proof of the meaning formula of the taclets. For instance, a user might choose to introduce the following rule for *modus ponens* of antecedent formulas:

—— Taclet ——

```
\rules {
  mpLeft {
    \formula phi, psi;
    \assumes( phi ==> ) \find( phi -> psi ==> )
                        \replacewith( psi ==> )
  };
}
```

—————————————————————————— Taclet ——

This rule is subsumed by propositional rules that already exist in KeY (since the KeY calculus is complete for propositional logic), but might sometimes be more natural to use in proofs than the built-in rules. The soundness of the rule can easily be shown automatically be KeY.

## 4.3 A Reference Manual of Taclets

This section introduces the concrete syntax of the taclet language and explains its semantics. It is written in the style of a reference manual for the different taclet constructs and provides most of the information that is necessary for writing one's own taclets to implement a new theory.

### *4.3.1 The Taclet Language*

Taclets formalize sequent calculus rule schemata (see Section 3.5.1) within the KeY system. They define elementary proof goal expansion steps and describe

1. to which parts of a sequent and
2. under which conditions the taclet can be applied, and
3. in which way the sequent is modified yielding new proof goals.

This information is declared in the different parts of the body of a taclet. Figure 4.5 shows the syntax of the taclet language, which is explained in more detail on the following pages. The taclet language is part of the language for KeY input files whose grammar is described in Appendix B. The nonterminal symbols of the grammar that are not further expanded in Figure 4.5 (in particular ⟨*schematicSequent*⟩, ⟨*schematicFormula*⟩, and ⟨*schematicTerm*⟩) can be found in the appendix.

#### 4.3.1.1 A Taclet Section

⟨*taclets*⟩ ::= `\rules {` ( ⟨*taclet*⟩ )∗ `}`
| `\axioms {` ( ⟨*taclet*⟩ | ⟨*axiom*⟩ )∗ `}`

KeY input files are divided into different section that define various parts of the syntactical language that can be used (functions, predicates, sorts, ...). Taclets are declared in their own sections headed by either `rules` or `axioms`. The header should be used to differentiate between rules which define the semantics of a newly introduced logical theory and theorems and lemma rules which follow from the axioms. Rules consisting only of a single formula (see Section 4.3.1.3) are only allowed in sections headed `axioms` .

KeY Syntax

⟨*taclets*⟩ ::= `\rules {` ( ⟨*taclet*⟩ )∗ `}`
| `\axioms {` ( ⟨*taclet*⟩ | ⟨*axiom*⟩ )∗ `}`

⟨*taclet*⟩ ::=
⟨*identifier*⟩ `{`
⟨*localSchemaVarDecl*⟩∗
⟨*contextAssumptions*⟩? ⟨*findPattern*⟩?
⟨*applicationRestriction*⟩? ⟨*variableConditions*⟩?
( ⟨*goalTemplateList*⟩ | `\closegoal` )
⟨*ruleSetMemberships*⟩?
`}`

⟨*axiom*⟩ ::= ⟨*identifier*⟩ `{` ⟨*formula*⟩ `}`

⟨*localSchemaVarDecl*⟩ ::= `\schemaVar` ⟨*schemaVarDecl*⟩
⟨*schemaVarDecl*⟩ ::= ⟨*schemaVarType*⟩ ⟨*identifier*⟩ ( `,` ⟨*identifier*⟩ )∗ `;`

⟨*contextAssumptions*⟩ ::= `\assumes (` ⟨*schematicSequent*⟩ `)`

⟨*findPattern*⟩ ::= `\find (` ⟨*schematicExpression*⟩ `)`
⟨*schematicExpression*⟩ ::=
⟨*schematicSequent*⟩ | ⟨*schematicFormula*⟩ | ⟨*schematicTerm*⟩

⟨*applicationRestriction*⟩ ::= `\inSequentState` | `\sameUpdateLevel`
| `\antecedentPolarity` | `\succedentPolarity`

⟨*variableConditions*⟩ ::= `\varcond (` ⟨*variableConditionList*⟩ `)`
⟨*variableConditionList*⟩ ::= ⟨*variableCondition*⟩ ( `,` ⟨*variableCondition*⟩ )∗
⟨*variableCondition*⟩ ::= `\notFreeIn(` ⟨*identifier*⟩ `,` ⟨*identifier*⟩ `)`
| `\new(` ⟨*identifier*⟩ `, \dependingOn(` ⟨*identifier*⟩ `))`

⟨*goalTemplateList*⟩ ::= ⟨*goalTemplate*⟩ ( `;` ⟨*goalTemplate*⟩ )∗
⟨*goalTemplate*⟩ ::=
⟨*branchName*⟩?
( `\replacewith (` ⟨*schematicExpression*⟩ `)` )?
( `\add (` ⟨*schematicSequent*⟩ `)` )?
( `\addrules (` ⟨*taclet*⟩ ( `,` ⟨*taclet*⟩ )∗ `)` )?
⟨*branchName*⟩ ::= ⟨*string*⟩ `:`

⟨*ruleSetMemberships*⟩ ::= `\heuristics (` ⟨*identifierList*⟩ `)`
⟨*identifierList*⟩ ::= ⟨*identifier*⟩ ( `,` ⟨*identifier*⟩ )∗

KeY Syntax

**Figure 4.5** The taclet syntax

#### 4.3.1.2 A Taclet Declaration

⟨*taclet*⟩ ::=
  ⟨*identifier*⟩ `{`
    ⟨*localSchemaVarDecl*⟩∗
    ⟨*contextAssumptions*⟩? ⟨*findPattern*⟩?
    ⟨*applicationRestriction*⟩? ⟨*variableConditions*⟩?
    ( ⟨*goalTemplateList*⟩ | `\closegoal` )
    ⟨*ruleSetMemberships*⟩?
  `}`

Every taclet has got a unique name and a body containing elements describing how the taclet is to be matched against a sequent followed by a description of what action will then take place. The order of elements matters in taclet definitions, the system will not accept taclet definitions that disobey this order of declaration.

#### 4.3.1.3 Special Case: Axiom Declarations

⟨*axiom*⟩ ::=
  ⟨*identifier*⟩ `{`
    ⟨*formula*⟩
  `}`

When defining a logical theory, it is often clearer to state the axiomatic basis as a set of individual formulas rather than as inference rules that are matched against the current proof state. The semantics of axiom rules is very similar to rules that consist of a single `\add` clause. The first axiom `length_nil` from Figure 4.3, for instance, is semantically equivalent to the rule

— Taclet —

```
length_nil {
  \add( length(nil) = 0 ==> )
}
```

— Taclet —

A special situation arises if quantified axioms are to be defined as taclets. The formula patterns in nonaxiomatic rule definitions are schematic formulas in which only schema variables can be quantified. The second example from the same figure, would hence have to be reformulated more lengthily when composed as a usual rule:

— Taclet —

```
length_cons {
  \schemaVar \variable List l;
  \schemaVar \variable any a;
  \add( \forall l; \forall a;
           length(cons(a, l)) = 1 + length(l) ==> )
```

```
}
```

Taclet

Not all axioms can be stated as individual first-order formulas. The induction rule `list_induction` from Figure 4.1, for instance, *is* a schematic rule (standing for the infinite set of all possible instantiations of the schema variable `phi`) that cannot be formulated using this notation.

#### 4.3.1.4 Schema Variables: Declaring Matching Placeholders

⟨*localSchemaVarDecl*⟩ ::= **`\schemaVar`** ⟨*schemaVarDecl*⟩
⟨*schemaVarDecl*⟩ ::= ⟨*schemaVarType*⟩ ⟨*identifier*⟩ ( **`,`** ⟨*identifier*⟩ )∗ **`;`**

The patterns within the clauses of taclet definitions are templates which can be applied to many concrete instantiations. They may, hence, contain placeholder symbols called *schema variables* which are instantiated during rule application either by matching the template description containing schematic entities to a part of the current proof sequent or through user input.

Schema variables can be declared locally at the beginning of a taclet or globally in a separate section before the taclet definitions. The available types of schema variables are listed and explained in Section 4.3.2.

#### 4.3.1.5 Context Assumptions: What Has to Be Present in a Sequent

⟨*contextAssumptions*⟩ ::= **`\assumes`** **`(`** ⟨*schematicSequent*⟩ **`)`**

Context assumptions are—together with the `\find` part of a taclet—the means of expressing that a goal modification can only be performed if certain formulas are present in the goal. If a taclet contains an `\assumes` clause, then the taclet may only be applied if the specified formulas are part of the goal that is supposed to be modified. Assumptions specify side conditions for the application of taclets. The formulas specified as assumptions are not modified[3] by the taclet application.

#### 4.3.1.6 Find Pattern: To Which Expressions a Taclet Can Be Applied

⟨*findPattern*⟩ ::= **`\find`** **`(`** ⟨*schematicExpression*⟩ **`)`**
⟨*schematicExpression*⟩ ::=
⟨*schematicSequent*⟩ | ⟨*schematicFormula*⟩ | ⟨*schematicTerm*⟩

More specifically than just to a goal of a proof, taclets are usually applied to an occurrence of either a formula or a term within this goal. This occurrence is called

[3] It is possible, however, that an assumption is also matched by the `\find` pattern of the taclet. In this situation a taclet application can modify or remove an assumption.

the *focus* of the taclet application and is the only place in the goal where the taclet can modify an already existing formula.

There are three different kinds of patterns a taclet can match on:

- A schematic sequent that contains a formula: this either specifies that the taclet can be applied if the given formula is an element of the antecedent, or if it is an element of the succedent, with the formula being the focus of the application. It is allowed, however, that the occurrence of the formula is preceded by updates (see the section on "State Conditions" and Section 3.5.1).
  The question how many formulas may appear in a schematic sequent is not settled by the grammar. The KeY implementation insists that there is exactly one formula in schematic sequents in find patterns while in assumes patterns multiple occurrences are possible, e.g., `\assumes (phi1, phi2 ==>)`.
- A formula: the focus of the application can be an arbitrary occurrence of the given formula (also as subformula) within a goal.
- A term: the focus of the application can be any occurrence of the given term within a goal.

Taclets with the last two kinds of `\find` patterns are commonly referred to as *rewriting taclets*.

The find pattern is an optional part of a taclet definition. However, most taclets possess a find pattern which acts as a hook for the strategy during automatic proof search by which it finds applicable rules. There are only few taclets without find clause with the cut rule that allows for case distinction being the most prominent example. Axioms (in Figure 4.3, e.g.) are also taclets without find clause since they add knowledge unconditionally onto the sequent.

#### 4.3.1.7 State Conditions: Where a Taclet Can Be Applied

⟨*applicationRestriction*⟩ ::= `\inSequentState` | `\sameUpdateLevel` | `\antecedentPolarity` | `\succedentPolarity`

In JavaDL—like in any modal logic—, the same expression may evaluate differently depending on the modalities in whose context it occurs. A finer control over where the focus of a taclet application may be located is needed. For rewriting rules it is, for instance, often necessary to forbid taclet applications within the scope of modal operators in order to ensure soundness. Likewise, some rewrite rules are only sound if the matched focus lies within a context of a certain polarity.

There are three different "modes" that a taclet can have and that restrict its applicability:

- `\inSequentState`: the most restrictive mode, in which the focus of a taclet application must not be located within the scope of *any* modal operator.Likewise, the assumptions that match the `\assumes` pattern must not be under the influence of any modality.
  There are two submodes for this mode that restrict under which logical connectives a formula may appear. These modes anticipate on which side of the

**Table 4.1** Matrix of the different taclet modes and the different `\find` patterns

| | `\find` pattern is sequent | `\find` pattern is term or formula | No `\find` |
|---|---|---|---|
| *Operators that are allowed above focus* | | | |
| `\inSequentState` | None | All nonmodal operators | *Forbidden combination* |
| `\sameUpdateLevel` | *Forbidden combination* | All nonmodal operators, updates | *Forbidden combination* |
| Default | Updates | All operators | — |
| *Which updates occur above* `\assumes` *and* `\add` *formulas* | | | |
| `\inSequentState` | None | None | *Forbidden combination* |
| `\sameUpdateLevel` | *Forbidden combination* | Same updates as above focus | *Forbidden combination* |
| Default | Same updates as above focus | None | None |
| *Which updates occur above* `\replacewith` *formulas* | | | |
| `\inSequentState` | None | None | *Forbidden combination* |
| `\sameUpdateLevel` | *Forbidden combination* | Same updates as above focus | *Forbidden combination* |
| Default | Same updates as above focus | Same updates as above focus | None |

For each combination, it is shown (1) where the focus of the taclet application can be located, and (2) which updates consequently have to occur above the formulas that are matched or added by `\assumes`, `\add` or `\replacewith`.

sequent a subformula would end up if the top-level formula were fully expanded using the basic propositional sequent calculus rules. For example, in the sequent $\neg a \Longrightarrow b \wedge \neg c$, the formula $c$ has "antecedent polarity" while $a$ and $b$ have "succedent polarity" since the fully expanded equivalent sequent reads $c \Longrightarrow b, a$. The mode flags `\antecedentPolarity` or `\succedentPolarity` can be added to constrain the application of a taclet to the one polarity or the other.

- `\sameUpdateLevel`: this mode is only allowed for rewriting taclets and allows the application focus of a taclet to lie within the scope of updates, but not in the scope of other modal operators. The same updates that occur in front of the application focus must also occur before the formulas referred to using `\assumes`. The same update context is used when the `\replacewith` and `\add` patterns are expanded.
- Default: the most liberal mode. For rewriting taclets, this means that the focus can occur arbitrarily deeply nested and in the scope of any modal operator. If the `\find` pattern of the taclet is a sequent, then the application focus may occur below updates, but not in the scope of any other operator.

While there are no restrictions on the location of the focus, for rewriting taclets in default mode, formulas that are described by `\assumes` or `\add` must *not* be in the scope of updates.

An important representative for rules which require the mode `\sameUpdateLevel` is the rule `applyEq`:

Taclet

```
applyEq {
  \schemaVar \variable \term int t1, t2
  \assumes( t1 = t2 ==> )
  \find( t1 ) \sameUpdateLevel
  \replacewith( t2 )
}
```

Taclet

The mode flag `\sameUpdateLevel` is mandatory for the soundness of the rule as it prevents the rule from illegally replacing terms which are influenced by an update. In the sequent $c \doteq 3 \Longrightarrow \{c := 0\}(c > 0)$, the term $c > 0$ cannot soundly be replaced with $3 > 0$ since the equality $c \doteq 3$ does not hold in the scope of the update. see also the examples on page 119.

As an example for a taclet that must be declared using the state condition `\antecedentPolarity`, consider the taclet

Taclet

```
weaken {
  \schemaVar \formula phi;
  \find( phi )
  \antecedentPolarity
  \replacewith( true )
}
```

Taclet

that allows replacing of any subformula $\phi$ within the goal by true. In general, replacing a subformula by true is not a sound proof step. The taclet becomes, however, a sound rule if the polarity restriction is added: Replacing a formula in the antecedent by true strengthens the proof obligation and is thus a valid proof step. Using the modifier `\antecedentPolarity`, one can strengthen the obligation without having to fully expand its propositional structure. State conditions also affect the formulas that are required or added by `\assumes`, `\add` or `\replacewith` clauses. The relation between the positions of the different formulas is also shown in Table 4.1.

#### 4.3.1.8 Variable Conditions: How Schema Variables May Be Instantiated

⟨*variableConditions*⟩ ::= `\varcond` `(` ⟨*variableConditionList*⟩ `)`
⟨*variableConditionList*⟩ ::= ⟨*variableCondition*⟩ ( `,` ⟨*variableCondition*⟩ )*
⟨*variableCondition*⟩ ::= `\notFreeIn(` ⟨*identifier*⟩ `,` ⟨*identifier*⟩ `)`
| `\new(` ⟨*identifier*⟩ `,` `\dependingOn(` ⟨*identifier*⟩ `))`

Schema variables are declared with a *kind* restricting how they can be instantiated. Many kinds additionally support sorts limiting instantiation even further (see Section 4.3.2). In many cases, one has to impose further restrictions on the instantiations of schema variables, for instance, state that certain logical variables must not occur free in certain terms. The taclet formalism is hence equipped with a simple language for expressing such conditions, *variable conditions*. To each taclet, a list of variable conditions can be attached which will be checked when the taclet is about to be applied.

Many variable conditions are available in KeY, but only two are of importance when defining new theories. See Appendix B.2.3.3 for a list of all available variable conditions.

**notFreeIn** The variable condition `\notFreeIn(`*lv*`,` *te*`)` is satisfied if the logical variable which is the instantiation of the schema variable *lv* does *not* occur (freely) in the instantiation of *te* (which is a term or a formula). The following rule, for instance, removes a universal quantifier if the quantified variable *x* does not occur in the matrix *b*.

— Taclet —

```
deleteForall {
   \schemaVar \formula b;
   \schemaVar \variable int x;
   \find( \forall x; b )
   \varcond( \notFreeIn(x, b) )
   \replacewith( b )
}
```

— Taclet —

**new** The variable condition `\new(sk, \dependingOn(t))` is used to indicate that the schema variable `sk` is to be instantiated with a fresh symbol which has not yet been used anywhere else within the proof. A fresh symbol not yet present is surely not constrained by a formula on the sequent and can thus stand in for an arbitrary value. After naming the schema variable `sk` which is to be instantiated, one has to include a `\dependingOn()` clause listing all schema variables on which the value of `sk` may depend. This variable condition used to be mandatory in older versions of KeY, but is optional now. It is still valuable for documentation purposes.

As an example consider the following taclet `pullOut` which allows the user to replace a concrete integer expression *t* by a fresh constant *sk*. The equality

between the two is added as assumption to the antecedent of the sequent:

—— Taclet ——

```
pullOut {
  \schemaVar \term int t;
  \schemaVar \skolemTerm int sk;
  \find( t )
  \sameUpdateLevel
  \varcond( \new(sk, \dependingOn(t)) )
  \replacewith( sk )
  \add( t = sk ==> )
}
```

—— Taclet ——

#### 4.3.1.9 Goal Templates: The Effect of the Taclet Application

⟨*goalTemplateList*⟩ ::= ⟨*goalTemplate*⟩ ( **;** ⟨*goalTemplate*⟩ )∗
⟨*goalTemplate*⟩ ::=
⟨*branchName*⟩?
( **\replacewith (** ⟨*schematicExpression*⟩ **)** )?
( **\add (** ⟨*schematicSequent*⟩ **)** )?
( **\addrules (** ⟨*taclet*⟩ ( **,** ⟨*taclet*⟩ )∗ **)** )?
⟨*branchName*⟩ ::= ⟨*string*⟩ **:**

If the application of a taclet on a certain goal and a certain focus is permitted and is carried out, the *goal templates* of the taclet describe in which way the goal is altered. Generally, the taclet application will first create a number of new proof goals (split the existing proof goal into a number of new goals) and then modify each of the goals according to one of the goal templates. A taclet without goal templates will close a proof goal. In this case the keyword `\closegoal` is written instead of a list of goal templates to clarify this behavior syntactically.

Goal templates are made up of three kinds of operations:

- `\replacewith`: if a taclet contains a `\find` clause, then the focus of the taclet application can be replaced with new formulas or terms. `\replacewith` has to be used in accordance with the kind of the `\find` pattern: if the pattern is a sequent, then also the argument of the keyword `\replacewith` has to be a sequent, etc. In contrast to `\find` patterns, there is no restriction concerning the number of formulas that may turn up in a sequent being argument of `\replacewith`. It is possible to remove a formula from a sequent by replacing it with an empty sequent, or to replace it with multiple new formulas.
- `\add`: independently of the kind of the `\find` pattern, the taclet application can add new formulas to a goal.
- `\addrules`: a taclet can also create new taclets when being applied. We will not go into this subject any deeper in this chapter.

Apart from that, each of the new goals (or branches) can be given a name using a `<branchName>` rule in order to improve readability of proof trees. Observe that this rule has to be terminated by `:`.

Note that a semicolon separates goal templates. The action of the taclet whose goal template is defined as `\find(A) \replacewith(B) \add(C==>)` has a single goal template: it replaces `A` with `B` and adds `C` to the antecedent of the sequent. In contrast to this, the taclet defined as `\find(A) \replacewith(B) ; \add(C==>)` has got two goal templates such that this rules spawns two child sequences, one replacing `A` with `B` and one with adding `C` to the sequent.

#### 4.3.1.10 Rule Sets: Control How Taclets are Applied Automatically

⟨*ruleSetMemberships*⟩ ::= **`\heuristics`** `(` ⟨*identifierList*⟩ `)`
⟨*identifierList*⟩ ::= ⟨*identifier*⟩ ( `,` ⟨*identifier*⟩ )∗

Each taclet can be declared to be element of one or more rule sets, which are used by the *proof strategies* in KeY to choose the taclets which are applied automatically. Rule sets describe collections of taclets that should be treated in the same way by the strategies. The strategies work by assigning weights (called "costs") to all possible rule application and by choosing that applicable taclet for a sequent that has the lowest cost. The cost of an applicable rule application decreases over time (i.e., while other taclets take precedence), thus guaranteeing that every possible rule application will eventually be taken (fairness).

There exists a number of rule sets in KeY, of which only a few are relevant for creating new data types definitions. Most rule sets are special-purpose indicators used by the strategies. Table 4.2 lists those rule sets interesting for the design of data types and theories. Of particular interest are the rule sets 'userTaclets1', 'userTaclets2' and 'userTaclets3' whose priority can be chosen by the user and even modified at runtime during an interactive KeY proof.

There is one strategy optimization implemented to increase the performance of KeY's JavaDL calculus: the *One-Step-Simplifier*. This built-in aggregator rule accumulates taclet applications of the 'concrete' and 'simplify' rule sets and applies them as one modification to a formula within the sequent. The rules applied by the one-step-simplifier are the same as are applied by the strategies; however, proofs with and without activated One-Step-Simplifier may sometimes differ due to the order in which the individual rules are applied.

### *4.3.2 Schema Variables*

Schema variables are placeholders for different kinds of syntactic entities that can be used in taclets. Despite their name *variable*, schema variables are a very broad concept in KeY. Schema variables can stand in for different kinds of variables (like

**Table 4.2** Most important rule sets in KeY

| | |
|---|---|
| concrete | Rules that simplify expressions containing concrete *constant* values are subsumed into this rule set. This includes, for instance, the rules that simplify $x \wedge \mathit{true}$ to $x$ or $2+4$ to 6. Taclets for computations with concrete values have the highest priority and are applied eagerly. |
| simplify | Rules that simplify expressions locally, without making additional assumptions, are collected into this rule set. This includes a large number of taclets, for instance, the one that simplifies the expression $\mathit{elementOf}(o, f, \mathit{union}(A, B))$ into $\mathit{elementOf}(o, f, A) \vee \mathit{elementOf}(o, f, B)$ for location sets $A$ and $B$. Taclets in set 'simplify' are applied eagerly, but with less priority than taclets in 'concrete'. |
| simplify_enlarging | Simplification taclets that expand a definition such that the resulting expression is considerably longer than the original one go into this rule set. It is applied with more reluctance than the above rule sets since it makes sequents grow. The rules which expand the predicate *wellFormed* (modeling the well-formedness of reachable Java heap models) belong to this set, for instance. |
| inReachableState-Implication | Taclets that add new formulas onto the sequent go into this set. The strategies make sure that the same formula is not added twice onto the same branch which could make the prover run round in circles. Rules in this rule set are applied more reluctantly and only if no rule of the above rule sets can be applied. The name of this rule set is historic, a more appropriate name would be 'adding'. |
| userTaclets1<br>userTaclets2<br>userTaclets3 | These three rule sets are empty by default and are meant to be inhabited by user-defined taclets implemented for new theories and data types. Their priority can be controlled interactively by the user in the user interface of KeY. |

logical variables or program variables), terms, formulas, programs or more abstract things like types or modal operators.

Schema variables are used in taclet definitions. When a taclet is applied, the contained schema variables will be replaced by *concrete* syntactic entities. This process is called *instantiation* and ensures that schema variables never occur in proof sequents. Some schema variables are instantiated by *matching* schematic expressions against concrete expressions on the goal sequent, other instantiations come up only during taclet application (through user interaction or by the automatic proof strategies).

In order to ensure that no ill-formed expressions occur while instantiating schema variables with concrete expressions, e.g., that no formula is inserted at a place where only terms are allowed, the *kind* of a schema variable defines which entities the schema variable can represent and may be replaced with. Schema variables can be declared locally at the beginning of a taclet definition or globally at the beginning of a file.

*Example 4.3.* In KeY syntax, we globally declare `phi` to be a schema variable representing formulas and `n` a variable for terms of type *int*. The taclet definition for

**Table 4.3** Kinds of schema variables in the context of a type hierarchy $(\text{TSym}, \sqsubseteq)$

| | |
|---|---|
| `\variable` $A$ | Logical variables of type $A \in \text{TSym}$ |
| `\term` $A$ | Terms of type $B \sqsubseteq A$ (with $A \in \text{TSym}$) |
| `\formula` | Formulas |
| `\skolemTerm` $A$ | Skolem constants/functions of type $A \in \text{TSym}$ |
| `\program` $t$ | Program entities of type $t$ |

`impRight` locally declares another schema variables `psi` for formulas and makes use of it and the global `phi`.

KeY

```
\schemaVariables {
  \formula phi;
  \term int n;
}

\rules {
  impRight {
    \schemaVar \formula psi;
    \find( ==> phi -> psi )
    \replacewith( phi ==> psi )
  };
}
```

KeY

#### 4.3.2.1 Schema Variable Kinds

The most important kinds of schema variables in the KeY system are given in Table 4.3. A more detailed explanation of each of the different categories is given on the following pages. Out of the kinds of schema variables in the table, the first four are relevant if you want to introduce user-defined logical theories and calculus rules. The last one is needed only when taclets are introduced that deal with JavaDL program modalities. Many subkinds of program schema variables exist and the kind is listed here only for completeness' sake and will not be explained in detail.

**Variables:** `\variable` $A$

Schema variables for variables can be instantiated with logical variables (*not* with program variables) that have static type $A$. In contrast to schema variables for terms, logical variables of subtypes of $A$ are not allowed for instantiation.[4] Schema variables

[4] Such a semantics is hardly ever desired and would make development of sound taclets difficult.

of this kind can also be bound by quantifiers or variable-binding function symbols (see Section 2.3.1). Bound occurrences of such schema variables will also be replaced with concrete logical variables when instantiations are applied.

**Terms:** `\term` *A*

Schema variables for terms can be instantiated with arbitrary terms that have the static type *A* or a subtype of *A*. Subtypes are allowed because this behavior is most useful in practice: there are only very few rules for which the static type of involved terms has to match some given type *exactly*.[5] In general, there are no conditions on the logical variables that may occur (free) in terms substituted for such schema variables. When a term schema variable is in the scope of a quantifier, logical variables can be "captured" when applying the instantiation, which needs to be considered when writing taclets. The occurrence of variables within the instantiation of a term can be restricted using the variable condition `notFreeIn` (see Section 4.3.1.8).

**Formulas:** `\formula`

Schema variables for formulas can be instantiated with arbitrary JavaDL formulas. As for schema variables for terms, the substituted concrete formulas may contain free variables, and during instantiation variable capture can occur.

**Skolem Terms:** `\skolemTerm` *A*

A schema variable for Skolem terms is instantiated with a fresh constant $c_{\text{sk}}$ of type *A* that has not occurred anywhere in the proof, yet.

The taclet application mechanism in KeY creates a fresh constant symbols every time a taclet with such a schema variable is applied. This ensures that the inserted symbols are always new, and, hence, can be used as Skolem constants. Compare the remarks on page 30 in Chapter 2 and at the end of Section 3.5.1.1 on page 62 in Chapter 3.

There are only few rules that require schema variables for Skolem terms. Sometimes it is helpful to be able to talk about a witnessing object which has some property. One can realize that using a Skolem schema variable. An alternative would be to state a corresponding quantified formula.

Schema variables of this kind always require a corresponding variable condition `\new` (see Section 4.3.1.8).

[5] In case the reader needs to implement a schema variable with exact type *A*, they may use the modifier `strict` after `\term`.

**Other schema variable types**

Three schema variable kinds are concerned with matching program constructs and modalities. They are usually not required to define new data types and theories. Schema variables of type `\program` match against syntactical entities[6] within Java programs, and they can be used to compose new rules for symbolic execution (see Section 3.5.6) of Java modalities in JavaDL formulas.

Moreover, there exist a few special purpose schema variable types to match other syntactical entities like updates or term labels, but we will not discuss them here since they are not relevant for data type definitions.

#### 4.3.2.2 Schema Variable Instantiation

Schema variables are replaced with concrete entities when a taclet is applied. This replacement can be considered as a generalization of the notion of *ground substitutions* from Section 2.2.1 in Chapter 2, and like substitutions the replacement is carried out in a purely syntactic manner. A mapping from schema variables to concrete expressions is canonically extended to terms and formulas.

**Definition 4.4 (Instantiation of Schema Variables).** Let $(\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym})$ be a signature for a type hierarchy $\mathscr{T} = (\mathrm{TSym}, \sqsubseteq)$ and $SV$ a set of schema variables over $\mathscr{T}$. An *instantiation* of $SV$ is a partial mapping[7]

$$\iota : SV \nrightarrow \left(\mathrm{DLFml} \cup \bigcup_{A \in \mathrm{TSym}} \mathrm{DLTrm}_A\right)$$

that maps schema variables to syntactic entities without schema variables in accordance with Table 4.3. An instantiation is called *complete* for $SV$ if it is a total mapping on $SV$.

For sake of brevity, we also talk about instantiations of schematic terms or formulas, which really are instantiations of the set of schema variables that occur in the expression. Given a complete instantiation of a schematic expression, we can turn it into a concrete one by replacing all schema variables `sv` with the expression $\iota(\mathtt{sv})$. To this end we can extend $\iota$ to expressions which may also contain schema variables. In such expressions, a schema variable of type `\formula` can be used in places where a formula is admissible, for instance.

*Example 4.5.* Table 4.4 illustrates the instantiation of the different kinds of schema variables for first-order logic. We assume that $f, g : A \to A$ are function symbols, $a, c : A$ are constants, $p : A$ and $q$, $r$ are predicates and $x$:$A$ is a logical variable.

---

[6] like Java expressions, local variables, method or field references, types, switch labels, ...

[7] This is for the schema variables presented here. The domain of $\iota$ must be extended if schema variables for program elements, or modalities are considered.

**Table 4.4** Examples of schematic expressions and their instantiations

| Expression $t$ | Instantiation $\iota$ | Instance $\iota(t)$ |
|---|---|---|
| $f(\texttt{te})$ | $\{\texttt{te} \mapsto g(a)\}$ | $f(g(a))$ |
| $f(\texttt{va})$ | $\{\texttt{va} \mapsto x\}$ | $f(x)$ |
| $\forall \texttt{va}; p(\texttt{va})$ | $\{\texttt{va} \mapsto x\}$ | $\forall x; p(x)$ |
| $\forall \texttt{va}; p(\texttt{te})$ | $\{\texttt{va} \mapsto x, \texttt{te} \mapsto x\}$ | $\forall x; p(x)$ |
| $\forall \texttt{va}; \texttt{phi}$ | $\{\texttt{va} \mapsto x,\ \texttt{phi} \mapsto p(x)\}$ | $\forall x; p(x)$ |
| $\texttt{phi} \wedge p(\texttt{te})$ | $\{\texttt{phi} \mapsto q \vee r,\ \texttt{te} \mapsto f(a)\}$ | $(q \vee r) \wedge p(f(a))$ |
| $p(\texttt{sk}) \rightarrow \exists \texttt{va}; p(\texttt{va})$ | $\{\texttt{sk} \mapsto c,\ \texttt{va} \mapsto x\}$ | $p(c) \rightarrow \exists x; p(x)$ |

Schema variables:
```
\variables A va; \term A te;
\formula phi; \skolemTerm A sk;
```

#### 4.3.2.3 Well-formedness Conditions

Not all taclets that can be written using the syntax of Section 4.3.1 are meaningful or desirable descriptions of rules. We want to avoid, in particular, rules whose application could destroy well-formedness of formulas or sequents.

Following Chapter 2, we do not allow sequents of our proofs to contain free logical variables. Unfortunately, this is a property that can easily be destroyed by incorrect taclets:

— Taclet —
```
illegalTac1 { \find(==> \forall va; p(va))
              \replacewith(==> p(va)) };
illegalTac2 { \find(==> \forall va; phi)
              \replacewith(==> phi) };
```
— Taclet —

In both examples, the taclets remove quantifiers and possibly inject free variables into a sequent: (1) schema variables of kind `\variable` could occur free in clauses `\add` or `\replacewith`, or (2) a logical variable $\iota(\texttt{va})$ could occur free in the concrete formula $\iota(\texttt{phi})$ that a schema variable `phi` represents, and after removing the quantifier, the variable would be free in the sequent (the same can happen with schema variables for terms). We will rule out both taclets by imposing suitable constraints.

To avoid that taclets like `illegalTac1` endanger the well-formedness of proof sequents, schema variables of kind `\variable` must not occur free in `\find`, `\assumes`, `\replacewith` and `\add` clauses. To forbid taclets like `illegalTac2`, schema variables must be used consistently: If a schema variable `t` is in the scope of a quantification over a schema variable `va`, then

1. every occurrence of `t` must also be in the scope of `va`, *or*
2. the taclet must be annotated with the variable condition `\notFreeIn(t, va)`.

Both properties can be checked statically, and the KeY implementation rejects ill-formed taclets immediately; they cannot even be loaded.

### 4.3.3 Application of Taclets

This section informally explains the process of how taclets are applied as sequent calculus rules on JavaDL sequents in KeY. A more formal introduction of the semantics of taclets can be found in Section 4.4 on reasoning about the soundness of taclets.

A taclet schematically describes a set of sequent calculus rules. By instantiating the schema variables in its clauses with concrete syntactical elements, it becomes a concrete applicable rule in the calculus (see Section 2.2.2). When applying a taclet, *all* schema variables must be instantiated according to their declaration. Many instantiations are determined by *matching* schematic expressions against concrete expressions on the goal sequent. Thus, it is determined if (and by which instantiation) the schematic and the concrete expression can be unified.

But if schema variables occur in the taclet but not in the `\find` or `\assumes` clauses, they cannot be instantiated by matching. In interactive proofs, the user is then asked to provide suitable instantiations; in automatic proofs, heuristics are invoked to come up with instantiations (e.g., for finding suitable ground instances of quantified statements).

An important role in the taclet application process is played by the `\find` clause since it determines *where* on the sequent the taclet performs rewriting actions. Both in automatic and interactive reasoning, this clause chooses the application focus (see Section 4.3.1.6) and thus triggers the rule application. We write $\underline{focus}$ to denote the located application focus of a rule application, that is, $\underline{focus}$ refers actually to a position within the sequent. We will use the notation $\underline{focus}$ also for the matched term or formula. The side conditions (variable conditions, `\assumes` clause, state conditions) are checked afterwards, and only if all of them are satisfied will the rule be applied.

Consider a well-formed taclet $t$ and let $SV$ denote the set of schema variables in $t$. An *applicable instantiation of* $t$ is a tuple $(\iota, \mathscr{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$ consisting of

- a complete instantiation $\iota$ of $SV$,
- an update $\mathscr{U}$ describing the context of the taclet application ($\mathscr{U}$ can be empty),
- a sequent $\Gamma \Longrightarrow \Delta$ to which the taclet is supposed to be applied, and
- an application focus $\underline{focus}$ within $\Gamma \Longrightarrow \Delta$ that is supposed to be modified (we write $\underline{focus} = \bot$ if $t$ does not have a `\find` clause)

that satisfies the following conditions:

1. $\iota$ is an admissible instantiation of $SV$,
2. $\iota$ satisfies all variable conditions of taclet $t$,
3. all logical variables $\iota(\texttt{va})$ represented by schema variables `va` of kind `\variable` in $t$ are distinct,
4. if $t$ has a `\find` clause, then the position of $\underline{focus}$ is consistent with the state conditions of $t$ (Table 4.1),
5. $\mathscr{U}$ is derived from $\underline{focus}$ according to the middle part "Which updates have to occur above `\assumes` and `\add` formulas" of Table 4.1 (for $\underline{focus} = \bot$ and the fields "forbidden combination" we choose the empty update `skip`),

6. for each formula $\phi$ of an `\assumes` clause of $t$, $\Gamma \Longrightarrow \Delta$ contains a corresponding formula $\mathscr{U}\iota(\phi)$ (on the correct side),
7. if $t$ has a clause `\find(`$f$`)`, where $f$ is a formula or a term, then $\iota(f) = \underline{focus}$ (the `\find` pattern has to match the focus of the application),
8. if $t$ has a clause `\find(`$f$`)`, where $f$ is a sequent containing a single formula $\phi$, then $\iota(\phi) = \underline{focus}$ and the formulas $\phi$ and $\underline{focus}$ occur on the same sequent side (both antecedent or both succedent),
9. if a state condition `\antecedentPolarity` or `\succedentPolarity` is part of the rewrite taclet $t$ (see Section 4.3.1.7), then $\underline{focus}$ must have antecedent/-succedent polarity,
10. for every schema variable `sv` of $t$ of kind `\term` or `\formula` and all free variables $x \in fv(\iota(\texttt{sv}))$,
    - `sv` is in the scope of a schema variable of type `\variable` with $\iota(\texttt{va}) = x$, or
    - $t$ contains at most one `\replacewith` clause, `sv` turns up only in `\find`, `\replacewith` or `\varcond` clauses of $t$, and $x$ is bound above $\underline{focus}$.

Once a complete taclet instantiation has been found applicable, it can be used to perform a step in the sequent calculus. Applying it onto a focus within an open proof goal spawns a set of new sequents, which are new proof goals after the application. The emerging sequents are obtained by modification of the original, carrying out the modification descriptions in the taclet's goal templates. The following informally describes the effects that the application of a taclet $t$ together with an applicable instantiation $(\iota, \mathscr{U}, \Gamma \Longrightarrow \Delta, \underline{focus})$ has on the goal.

First, the sequent is duplicated into new goals according to the number of goal templates (see Section 4.3.1.9) declared in the taclet. Every new goal corresponds to one goal template in $t$ where its effects will be carried out. The new goals become children of the original goal in the sequent calculus proof tree. The following steps are then repeated for every new goal. If there are no goal templates in the taclet (indicated by `\closegoal`) the rule application successfully closes the proof branch.

1. If the goal template has a clause `\replacewith(`$rw$`)`, where $rw$ is a formula or a term, then $\underline{focus}$ is replaced with $\iota(rw)$. If $rw$ is a term and the type $A_{new}$ of $\iota(rw)$ is not a subtype of the type $A_{old}$ of $\underline{focus}$, in symbols $A_{new} \not\sqsubseteq A_{old}$, then $\underline{focus}$ is replaced with $(A_{old})\iota(rw)$ instead of $\iota(rw)$ (a cast has to be introduced to prevent ill-formed terms).
2. If the goal template has a clause `\replacewith(`$rw$`)`, where $rw$ is a sequent, then the formula containing $\underline{focus}$ is removed from $\Gamma \Longrightarrow \Delta$, and for each formula $\phi$ in $rw$ the formula $\mathscr{U}\iota(\phi)$ is added (on the correct side).
3. If the goal template has a clause `\add(`$add$`)`, then for each formula $\phi$ in $add$ the formula $\mathscr{U}\iota(\phi)$ is added (on the correct side).

It is important to note that it is not possible to modify parts of the sequent other than through the focus. Formulas can be *added* to the sequent, but never can formulas that are not in the focus be *removed*. In terms of schematic sequent calculus rules,

this means that the *context* of the sequent ($\Gamma$ and $\Delta$) is always retained through taclet application.

## 4.4 Reflection and Reasoning about Soundness of Taclets

This section summarizes results published by Bubel et al. [2008]. See the paper for further details.

Taclets are a general language for describing proof modification steps. In order to ensure that the rules that are implemented using taclets are correct, we can consider the definitions of the previous sections and try to derive that no incorrect proofs can be constructed using taclets. This promises to be tedious work, however, and is for a larger number of taclets virtually useless if the reasoning is performed informally: we are bound to make mistakes.

For treating the correctness of taclets in a more systematic way, we would rather like to have some *calculus* for reasoning about soundness of taclets. This is provided in this section for some of the features of taclets. To this end, a two-step translation will be presented that define *first-order soundness proof obligations* for taclets.

- We describe a translation of taclets into formulas (the *meaning formulas* of taclets), such that a taclet is sound if the formula is valid. This translation captures the semantics of the different clauses that a taclet can consist of. Meaning formulas do, however, still contain schema variables, which means that for proving their validity, (higher-order) proof methods like induction over terms or programs are necessary.
- A second transformation handles the elimination of schema variables in meaning formulas, which is achieved by replacing schema variables with Skolem terms or formulas. The result is a formula in first-order logic, such that the original formula is valid if the derived formula is valid.

The two steps can be employed to validate taclets in different theorem prover contexts:

- Only the first step can be carried out, and one can reason about the resulting formula using an appropriate proof assistant in which the semantics of schema entities can be modeled, e.g., based on higher-order logic.
- Both steps can be carried out, which opens up for a wider spectrum of provers or proof assistants with which the resulting formulas can be tackled. The formulas can in particular be treated by KeY itself.

Proving KeY taclets within KeY is an interesting feature for *lemma* rules, i.e., taclets can be proven sound referring to other—more basic—taclets. The complete translation from taclets to formulas of dynamic logic can automatically be performed by KeY and makes it possible to write and use lemmas whenever this is useful, see [Bubel et al., 2008].

Proof obligations cannot be generated for all taclets in KeY. At the time of writing this, the following artifacts within a taclet definition keep it from being verifiable within KeY:

- program modalities in any clause,
- variable conditions other than `\new` and `\notFreeIn` (see Section B.2.3.3),
- meta-functions (symbols which are evaluated at rule application time by executing Java code),
- generic sorts, or
- schema variables other `\term`, `\formula`, `\variable`.

In the following, we first give a recapitulation about when rules of a sequent calculus are sound, and then show how this notion can be applied to the taclet concept. It has to be noted, however, that although reading the following pages in detail is not necessary for defining new taclets, it might help to understand what happens when lemmas are loaded in KeY.

### *4.4.1 Soundness in Sequent Calculi*

This section continues the discussion of Sequent Calculi begun in Section 2.2.2 by introducing a concept of soundness and criteria for it. In the whole section we write $(\Gamma \Longrightarrow \Delta)^* := \bigwedge \Gamma \rightarrow \bigvee \Delta$ for the formula that expresses the meaning of the sequent $\Gamma \Longrightarrow \Delta$. This formula is, in particular:

$$(\Longrightarrow \phi)^* = \phi \quad , \qquad (\phi \Longrightarrow)^* = \neg\phi \quad .$$

By the validity of a sequent we thus mean the validity of the formula $(\Gamma \Longrightarrow \Delta)^*$.

A further notation that we are going to use is the following "union" of two sequents:

$$\big(\Gamma_1 \Longrightarrow \Delta_1\big) \cup \big(\Gamma_2 \Longrightarrow \Delta_2\big) \quad := \quad \Gamma_1 \cup \Gamma_2 \Longrightarrow \Delta_1 \cup \Delta_2 \quad .$$

Because antecedents and succedents are defined to be sets, duplicate formulas will not appear twice.

**Definition 4.6 (Soundness).** A sequent calculus $C$ is *sound* if only valid sequents are derivable in $C$, i.e., if the root $\Gamma \Longrightarrow \Delta$ of a closed proof tree is valid.

This general definition does not refer to particular rules of a calculus $C$, but treats $C$ as an abstract mechanism that determines a set of derivable sequents. For practical purposes, however, it is advantageous to formulate soundness in a more "local" fashion and to talk about the rules (or taclets implementing the rules) of $C$. Such a local criterion can already be given when considering rules in a very abstract sense: a rule $R$ can be considered as an arbitrary (but at least semi-decidable) relation between tuples of sequents (the premisses) and single sequents (the conclusions). Consequently, $(\langle P_1, \ldots, P_k \rangle, Q) \in R$ means that the rule $R$ can be applied in an expansion step

$$\frac{P_1 \quad \cdots \quad P_k}{Q}$$

The following lemma relates the notion of soundness of a calculus with rules:

**Lemma 4.7.** *A calculus $C$ is sound, if for each rule $R \in C$ and also for all tuples $(\langle P_1, \ldots, P_k \rangle, Q) \in R$ the following implication holds:*

$$\text{if } P_1, \ldots, P_k \text{ are valid, then } Q \text{ is valid.} \tag{4.7}$$

If condition (4.7) holds for all tuples $(\langle P_1, \ldots, P_k \rangle, Q) \in R$ of a rule $R$, then this rule is also called *sound*.

### 4.4.2 Meaning Formulas of Sequent Taclets

In our case, the rules of a calculus $C$ are defined through taclets $t$ over a set $SV$ of schema 15variables, and within the next paragraphs we discuss how Lemma 4.7 can be applied considering such a rule. For a start, we consider a taclet whose `\find` pattern is a sequent (without implicit update) and that has the following basic shape:

—— Taclet ——

```
t1 { \assumes(assum) \find(findSeq) \inSequentState
     \replacewith(rw1) \add(add1);
     ...
     \replacewith(rwk) \add(addk) };
```

—— Taclet ——

Using text-book notation for rules in sequent calculi (as in Chapter 2), the taclet describes the rule

$$\frac{\texttt{rw1} \cup \texttt{add1} \cup \texttt{assum} \cup (\Gamma \Longrightarrow \Delta) \qquad \cdots \qquad \texttt{rwk} \cup \texttt{addk} \cup \texttt{assum} \cup (\Gamma \Longrightarrow \Delta)}{\texttt{findSeq} \cup \texttt{assum} \cup (\Gamma \Longrightarrow \Delta)}$$

In order to apply Lemma 4.7, it is then necessary to show implication (4.7) for all possible applications of the rule, i.e., essentially for all possible ways the schema variables that now turn up in the sequents can be instantiated. If $\iota$ is an applicable schema variable instantiation, and if $\Gamma \Longrightarrow \Delta$ is an arbitrary sequent, then

$$\begin{array}{rcll} P_i & = & \iota(\texttt{rw}i \cup \texttt{add}i \cup \texttt{assum}) \cup (\Gamma \Longrightarrow \Delta) & (i = 1, \ldots, k), \\ Q & = & \iota(\texttt{findSeq} \cup \texttt{assum}) \cup (\Gamma \Longrightarrow \Delta) & \end{array} . \tag{4.8}$$

Implication (4.7) can be replaced with:

$$\left(P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^*\right) \text{ is valid.} \tag{4.9}$$

Implication (4.7) is a *global* soundness criterion since validity of the premisses implies validity of the conclusion while the implication (4.9) is *local* in the sense that the premisses implies the conclusion in any single structure.

This new condition is stronger than (4.7), however not significantly stronger because of the side formulas $\Gamma \Longrightarrow \Delta$ that can be chosen arbitrarily. Inserting the sequents (4.8) extracted from taclet `t1` into (4.9) leads to a formula whose validity is sufficient for implication (4.7):

$$\begin{aligned} P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^* \;\; &= \;\; \textstyle\bigwedge_{i=1}^{k} \big(\iota(\texttt{rw}i \cup \texttt{add}i \cup \texttt{assum}) \cup (\Gamma \Longrightarrow \Delta)\big)^* \\ &\quad \rightarrow \big(\iota(\texttt{findSeq} \cup \texttt{assum}) \cup (\Gamma \Longrightarrow \Delta)\big)^* \end{aligned} \tag{4.10}$$

In order to simplify the right hand side of Equation (4.10), we can now make use of the fact that $\iota$ distributes through all propositional connectives ($\rightarrow$, $\wedge$, $\vee$, etc.) and also through the union of sequents. Thus, the formulas of Equation (4.10) are equivalent to

$$\iota\Big(\bigwedge_{i=1}^{k} \big(\texttt{rw}i^* \vee \texttt{add}i^*\big) \rightarrow \big(\texttt{findSeq}^* \vee \texttt{assum}^*\big)\Big) \vee (\Gamma \Longrightarrow \Delta)^*.$$

Showing that this formula holds for all sequents $\Gamma \Longrightarrow \Delta$, i.e., in particular for the empty sequent, is equivalent to proving

$$\iota\Big(\bigwedge_{i=1}^{k} \big(\texttt{rw}i^* \vee \texttt{add}i^*\big) \rightarrow \big(\texttt{findSeq}^* \vee \texttt{assum}^*\big)\Big)$$

for all possible instantiations $\iota$. We call the formula

$$M(\texttt{t1}) \;\; = \;\; \bigwedge_{i=1}^{k} \big(\texttt{rw}i^* \vee \texttt{add}i^*\big) \rightarrow \big(\texttt{findSeq}^* \vee \texttt{assum}^*\big) \tag{4.11}$$

the *meaning formula* of `t1`. From the construction of $M(\texttt{t1})$, it is clear that if $M(\texttt{t1})$ is valid whatever expressions we replace its schema variables with, then the taclet `t1` will be sound. Note that the disjunctions $\vee$ in the formula stem from the union operator on sequents. Intuitively, given that the premisses of a rule application are true (the formulas on the left side of the implication), it has to be shown that at least one formula of the conclusion is true.

We can easily adapt Equation (4.11) if some of the clauses of `t1` are missing in a taclet:

- If the `\find` clause is missing: in this case, `findSeq` can simply be considered as the empty sequent, which means that we can set $\texttt{findSeq}^* = \text{false}$ in Equation (4.11).
- If `\assumes` or `\add` clauses are missing: again we can assume that the respective sequents are empty and set

$$\texttt{assum}^* = \text{false}, \qquad \texttt{add}i^* = \text{false}$$

- If a clause `\replacewith(rwi)` is not present: then we can normalize by setting $\texttt{rw}i = \texttt{findSeq}$, which means that the taclet will replace the focus of the

application with itself. If both \replacewith and \find are missing, we can simply set $\texttt{rwi}^* = \text{false}$.

*Example 4.8.* We consider the taclet impRight from Ex. 4.3 that eliminates implications within the succedent. The taclet represents the rule schema

$$\frac{\texttt{phi} \Longrightarrow \texttt{psi}}{\Longrightarrow \texttt{phi} \rightarrow \texttt{psi}}$$

and the meaning formula is the logically valid formula

$$M(\texttt{impRight}) = (\underbrace{\neg\texttt{phi} \vee \texttt{psi}}_{=\texttt{rw1}^*}) \rightarrow (\underbrace{\texttt{phi} \rightarrow \texttt{psi}}_{=\texttt{findSeq}^*}) \equiv \neg(\texttt{phi} \rightarrow \texttt{psi}) \vee (\texttt{phi} \rightarrow \texttt{psi}) \ .$$

### 4.4.3 Meaning Formulas for Rewriting Taclets

The construction given in the previous section can be carried over to rewriting taclets.

— Taclet —

```
t2 { \assumes(assum) \find(findTerm) \inSequentState
     \replacewith(rw1) \add(add1);
     ...
     \replacewith(rwk) \add(addk) };
```

— Taclet —

In this case, findTerm and rw1, ..., rwk are schematic *terms*. We can, in fact, reduce the taclet t2 to a nonrewriting taclet (note, that the union operator $\cup$ is not part of the actual taclet language).

— Taclet —

```
t2b { \assumes(assum) \inSequentState
      \add( (findTerm=rw1 ==>) ∪ add1 );
      ...
      \add( (findTerm=rwk ==>) ∪ addk ) };
```

— Taclet —

We create a taclet that adds equations findTerm=rw1, ..., findTerm=rwk to the antecedent. Using taclet t2b and a general rule for applying equations in the antecedent, the effect of t2 can be simulated. On the other hand, also taclet t2b can be simulated using t2 and standard rules (cut, reflexivity of equality), which means that it suffices to consider the soundness of t2b. Equation (4.11) and some propositional simplifications then directly give us the meaning formula

$$M(\texttt{t2b}) \;\equiv\; M(\texttt{t2}) \;=\; \bigwedge_{i=1}^{k} \left(\texttt{findTerm} \doteq \texttt{rw}i \rightarrow \texttt{add}i^*\right) \rightarrow \texttt{assum}^* \;. \tag{4.12}$$

We have looked at rewriting taclets for terms so far. In the same way, rewriting taclets for formulas can be treated, if equations in (4.12) are replaced with equivalences:

$$\bigwedge_{i=1}^{k} \left((\texttt{findFor} \leftrightarrow \texttt{rw}i) \rightarrow \texttt{add}i^*\right) \rightarrow \texttt{assum}^* \tag{4.13}$$

For a taclet like `t2` but with mode flag `\succedentPolarity` (instead of `\inSequentSate`), the taclet application is limited to occurrences with positive polarity; the meaning formula is hence weaker and has the equivalence of (4.13) replaced by an implication:

$$\bigwedge_{i=1}^{k} \left((\texttt{findFor} \rightarrow \texttt{rw}i) \rightarrow \texttt{add}i^*\right) \rightarrow \texttt{assum}^*$$

Likewise, for a taclet which is annotated with `\succedentPolarity`, the meaning formula has this implication reversed:

$$\bigwedge_{i=1}^{k} \left((\texttt{rw}i \rightarrow \texttt{findFor}) \rightarrow \texttt{add}i^*\right) \rightarrow \texttt{assum}^*$$

*Example 4.9.* Let us go back to the taclet `applyEq` introduced in Section 4.3.1.7 on page 127. According to (4.12) its meaning formula is

$$M(\texttt{applyEq}) \quad = \quad (\texttt{t1} \doteq \texttt{t2} \rightarrow \text{false}) \rightarrow \texttt{t1} \doteq \texttt{t2} \tag{4.14}$$

with `t1` and `t2` schema variables. The implication of *false* is introduced since the taclet does not specify an `\assumes` clause. The next section will elaborate how the schematic meaning formula is refined into a concrete proof obligation.

### 4.4.4 Elimination of Schema Variables

Meaning formulas of taclets in general contain schema variables, i.e., placeholders for syntactic constructs like terms, formulas or programs. In order to prove a taclet sound, it is necessary to show that its meaning formula is valid for all possible instantiations of the schema variables.

Let us once more look at taclet `applyEq` (⇒ Ex. 4.9). In order to prove the taclet sound, we would have to prove the meaning formula (4.14) valid for all possible terms $\iota(\texttt{t1})$, $\iota(\texttt{t2})$ that we can substitute for `t1`, `t2`. Note that this *syntactic* quantification ranges over terms and is completely different from a first-order formula

$\forall int\ x; p(x)$, which is *semantic* and expresses that $x$ ranges over all elements in the set of integers.

Instead of explicitly enumerating instantiations using techniques like induction over terms, it is to some degree possible, however, to replace the syntactic quantification with an implicit semantic quantification through the introduction of Skolem symbols. For $M(\texttt{applyEq})$, it is sufficient to prove validity of the formula

$$\phi \quad = \quad (c \doteq d \to \text{false}) \to c \doteq d$$

in which $c$, $d$ are fresh constant symbols. The validity of $M(\texttt{applyEq})$ for all other instantiations follows, because the symbols $c$, $d$ can take the values of arbitrary terms $\iota(\texttt{t1})$, $\iota(\texttt{t2})$. Fortunately, $\phi$ is only a first-order formula that can be tackled with a calculus as defined in Chapter 2.

We will only sketch how Skolem symbols can be introduced for some of the schema variable kinds that are described in Section 4.3.2, more details can be found in [Bubel et al., 2008]. For the rest of the section, we assume that a taclet $t$ and its meaning formula $M(t)$ are fixed. We then construct an instantiation $\iota_{\text{sk}}$ of the schema variables that turn up in $t$ with Skolem expressions. In the example above, this instantiation would be

$$\iota_{\text{sk}} = \{\texttt{t1} \mapsto c,\ \texttt{t2} \mapsto d\}$$

**Variables:** `\variable` $A$

KeY makes the names of bound variables unique by internally renaming them[8]. Thus we need only consider instantiations $\iota$ that map different schema variables `va` to distinct logical variables. Such variables only occur bound in taclets and the identity of bound variables does not matter. Therefore, this instantiation $\iota_{\text{sk}}(\texttt{va})$ of a `\variable` schema variable `va` can simply be chosen to be a fresh logical variable $\iota_{\text{sk}}(\texttt{va}) = x$ of type $A$.

**Terms:** `\term` $A$

As already shown in the example above, a schema variable `te` for terms can be eliminated by replacing it with a term. While it sufficed to choose Skolem constants in the above case, in general, also the logical variables $\Pi(\texttt{te})$ that are bound in the context of `te` have to be taken into account and have to appear as arguments of the Skolem functions symbol. The reason is that such variables can occur in the term that is represented by `te`. We choose the instantiation $\iota_{\text{sk}}(\texttt{te}) = f_{\text{sk}}(x_1, \ldots, x_l)$, where

- $x_1, \ldots, x_l$ are the instantiations of the schema variables $\texttt{va}_1, \ldots, \texttt{va}_l$, i.e., $x_i = \iota_{\text{sk}}(\texttt{va}_i)$,

[8] applying so-called $\alpha$-*conversions*

- $\texttt{va}_1,\ldots,\texttt{va}_l$ are the (distinct) context variables of the variable `te` in the taclet $t$: $\Pi(\texttt{te}) = \{\texttt{va}_1,\ldots,\texttt{va}_l\}$,
- $f_{\text{sk}} : A_1,\ldots,A_l \to A$ is a fresh function symbol,
- $A_1,\ldots,A_l$ are the types of $x_1,\ldots,x_l$ and `te` is of kind `\term` $A$.

An example motivating the more complex Skolem expression will follow after the next paragraph which first describes a very similar situation.

**Formulas:** `\formula`

The elimination of schema variables `phi` for formulas is very similar to the elimination of term schema variables. The main difference is, obviously, that instead of a program variable, which is a nonrigid function symbol, a nonrigid predicate symbol has to be introduced: $\iota_{\text{sk}}(\texttt{phi}) = p_{\text{sk}}(x_1,\ldots,x_l)$, where

- $x_1,\ldots,x_l$ are the instantiations of the schema variables $\texttt{va}_1,\ldots,\texttt{va}_l$, i.e., $x_i = \iota_{\text{sk}}(\texttt{va}_i)$,
- $\texttt{va}_1,\ldots,\texttt{va}_l$ are the (distinct) context variables of the variable `te` in the taclet $t$: $\Pi_t(\texttt{te}) = \{\texttt{va}_1,\ldots,\texttt{va}_l\}$,
- $p_{\text{sk}} : A_1,\ldots,A_l$ is a fresh predicate symbol,
- $A_1,\ldots,A_l$ are the types of $x_1,\ldots,x_l$.

As an example to demonstrate the necessity of the arguments $x_1,\ldots,x_l$, consider the following unsound[9] taclet:

Taclet

```
swapMixedQuants {
  \schemaVar \variable int x;
  \schemaVar \variable int y;
  \schemaVar \formula phi;

  \find(\exists y; \forall x; phi)
  \replacewith(\forall x; \exists y; phi)
}
```

Taclet

Its meaning formula (according to (4.12)) is

$$M(\texttt{swapMixedQuants}) = (\exists \texttt{x};\forall \texttt{y};\texttt{phi}) \leftrightarrow (\forall \texttt{y};\exists \texttt{x};\texttt{phi})$$

with `x`, `y` and `phi` schema variables. Were `phi` replaced by a Skolem (propositional) constant $\iota(\texttt{phi}) = c$ and `x`, `y` by logical variables $x$, $y$, then the instantiated meaning formula $\iota(M(\texttt{swappedMixedQuants})) = (\exists x;\forall y;c) \leftrightarrow (\forall y;\exists x;c) \equiv c \leftrightarrow c$ would be valid although the taclet is clearly unsound.

[9] This taclet can, e.g., be used to prove $\exists \mathit{int}\ x;\forall \mathit{int}\ y;x \doteq y$ which is *not* a valid formula.

If, on the over hand, the instantiation for `phi` is chosen as $\iota'(\texttt{phi}) = b(x,y)$ with $x$ and $y$ as dependencies to a fresh Skolem predicate symbol $b : int, int$, the resulting formula $\iota'(M(\texttt{swapMixedQuants})) = (\exists x; \forall y; b(x,y)) \leftrightarrow (\forall y; \exists x; b(x,y))$ is not valid.

**Skolem Terms:** `\skolemTerm` *A*

Schema variables of kind `\skolemTerm` *A* are responsible for introducing fresh constant or function symbols in a proof. Such variables could in principle be treated like schema variables for terms, but this would strengthen meaning formulas excessively (often, the formulas would no longer be valid even for sound taclets).

We can handle schema variables `sk` for Skolem terms more faithfully: if in implication (4.7) the sequents $P_1, \ldots, P_k$ contain symbols that do not occur in $Q$, then these symbols can be regarded as universally quantified. Because a negation occurs in front of the quantifiers in (4.9) (the quantifiers are on the left side of an implication), the symbols have to be considered as existentially quantified when looking at the whole meaning formula. This entails that schema variables for Skolem terms can be eliminated and replaced with existentially quantified variables: $\iota_{\text{sk}}(\texttt{sk}) = x$, where $x$ is a fresh variable of type $A$.[10] At the same time, an existential quantifier $\exists x;$ has to be added in front of the whole meaning formula.

*Example 4.10.* The taclet `pullout` allows replacing any ground term `t` with a fresh Skolem constant `sk`; equality between them is guaranteed by an added assumption.

Taclet

```
pullout {
  \schemaVar \term G t;
  \schemaVar \skolemTerm G sk;

  \find( t ) \sameUpdateLevel
  \varcond( \new(sk, \dependingOn(t)) )

  \replacewith( sk )
  \add( t = sk ==> )
};
```

Taclet

The meaning formula of the taclet `pullout` is

$$M(\texttt{pullout}) \quad = \quad (\texttt{t} \doteq sk \rightarrow \neg t \doteq sk) \rightarrow \text{false} \quad \equiv \quad t \doteq sk$$

[10] Strictly speaking, this violates Definition 4.4, because schema variables for Skolem terms must not be instantiated with variables according to this definition. The required generalization of the definition is, however, straightforward.

according to (4.12). In order to eliminate the schema variables of this taclet, we first assume that the schematic type[11] `G` of the taclet is instantiated with a concrete type $A$.

If both schema variables `t` and `sk` were replaced by Skolem constants $c$ and $d$, the resulting formula $c \doteq d$ would be far from valid—though the taclet `pullout` *is* sound.

To overcome this imprecision, the schema variable `sk` can be replaced with a fresh logical variable $\iota_{\text{sk}}(\texttt{sk}) = x$ of type $A$. The schema variable `t` is eliminated through the instantiation by a Skolem constant $\iota_{\text{sk}}(\texttt{t}) = d$. Finally, we add an existential quantifier $\exists x$. The resulting formula without schema variables is

$$\exists x; \iota_{\text{sk}}(M(\texttt{pullout})) \quad \equiv \quad \exists x; (x \doteq d)$$

which is obviously universally valid.

**The Order Matters**

To establish the soundness of taclets for a theory, validity of the meaning formulas of all taclets in the theory must be shown. To this end, it would be convenient if already proved taclets could be used in the soundness proofs of the remaining taclets.

Such taclet applications must be restricted however: If taclets could be used unconditionally in each other's soundness proofs, two unsound taclets could be abused to mutually establish their validity. The consistency of the taclet rule base could thus be compromised.

A simple heuristics guarantees that such cyclic dependencies within the set of taclets of a theory are impossible: For the verification of the soundness of taclet, only taclets which are defined *before* it in the input file may be used.

This requires that the order of taclets is well thought of for the design of a theory to simplify the proof workload. Naturally, the axioms which fix the semantics, go first; followed by taclets capturing reusable lemmas, optimized special purpose taclets follow last.

[11] Schematic types, known as *generic sorts* in KeY, are like schema variables for type references that can be instantiated by concrete types.