RACSAM
Rev. R. Acad. Cien. Serie A. Mat.
VOL. **Falta (Falta)**, **Falta**, pp. 1–37
Ciencias de la Computaciónón / Computational Sciences
Comunicación Preliminar / Preliminary Communication

Taclets: A New Paradigm for Constructing Interactive Theorem Provers

Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle
Andreas Roth, Philipp Rümmer and Steffen Schlager

Abstract. Frameworks for interactive theorem proving give the user explicit control over the construction of proofs based on *meta languages* that contain dedicated control structures for describing proof construction. Such languages are not easy to master and thus contribute to the already long list of skills required by prospective users of interactive theorem provers. Most users, however, only need a convenient formalism that allows to introduce new rules with minimal overhead. On the the other hand, rules of calculi have not only purely logical content, but contain restrictions on the expected context of rule applications and heuristic information. We suggest a new and minimalist concept for implementing interactive theorem provers called *taclet*. Their usage can be mastered in a matter of hours, and they are efficiently compiled into the GUI of a prover. We implemented the KeY system, an interactive theorem prover for the full JAVA CARD language based on taclets.

Taclets:

Un Nuevo Paradigma para construir Demostradores Automáticos Interactivos

Resumen. Los marcos para la demostración interactiva de teoremas permiten al usuario tener control explícito de la construcción de las demostraciones sobre la base de unos metalenguajes que contienen unas estructuras de control dedicadas a la descripción de la construcción de las demostraciones. Estos lenguajes no son fáciles de dominar y se añaden así a la ya larga lista de habilidades requeridas a los potenciales usuarios de los demostradores interactivos de teoremas. Sin embargo, la mayoría de los usuarios sólo necesitan un formalismo conveniente que les permita introducir nuevas reglas con el mínimo esfuerzo. Por otra parte, las reglas de cálculo no sólo tienen un contenido puramente lógico, sino que también contienen restricciones sobre el contexto esperado de las aplicaciones de reglas y la información heurística. Se propone un concepto nuevo y minimalista para la implementación de los demostradores interactivos de teoremas que se ha denominado ”taclet”. Se puede dominar su uso en cuestión de horas, y se compilan de forma eficaz en la interfaz gráfica de usuario de un demostrador. Se ha implementado el sistema KeY, un demostrador interactivo de teoremas basado en taclets para el lenguaje completo JAVA CARD.

1. Introduction

Mechanical Theorem Proving is a field in Computer Science, where one tries to find mathematical proofs with the assistance of computer programs. Being one of the earliest contributions of *Artificial Intelligence*

Presentado por **Falta**.
Recibido: **Falta**. *Aceptado*: **Falta**.
Palabras clave / Keywords: formal verification, theorem proving, JAVA CARD, program logics
Mathematics Subject Classifications: 68T15, 68N30, 68Q60, 03B35, 03B45

(AI), the original motivation was the automation of theorem proving as a task for which human intelligence and creativity seemed indispensable. Like many other AI problems, theorem proving is first and foremost a *search* problem. Typically, one formalises the theorem to be proven in a suitable logic that comes with a sound and complete calculus so that one can systematically search for a proof that employs the rules of the logic calculus at hand.

Despite some early successes, progress in emulating human mathematicians was limited, though. On the other hand, mechanical theorem proving techniques turned out to be very effective in *technical applications* (mainly in computer science itself), where human provers are hampered by lack of intuitiveness or the sheer size of problems. Hardware and software systems of considerable size have been mechanically proven to be correct with the help of computer programs.

Today, theorem proving is a flourishing field of computer science, with new applications surfacing constantly. Most work is concerned with *fully automatic* theorem proving. Although nice when it works, for many complex applications it is simply not feasible. A typical example is software verification. Consider a partial correctness assertion of a program α: whenever precondition ϕ holds and α terminates, then in the final state of α postcondition ψ holds. Formalised in a suitable program logic with a relative complete calculus (for example, Hoare logic) proving a partial correctness assertion is a problem that can be tackled with theorem proving methods. Unfortunately, one does not get very far with fully automated proof search. It is instructive to look at the reasons for this:

- The rules of Hoare calculus (and any other non-trivial program logic) contain a number of rules that have an infinite local search space. For example, if α contains loops or is recursive one must use induction for proving correctness. This requires to find a suitable induction hypothesis. [1]

- A partial correctness proof of α, in general, makes use of properties of the datatypes that are occur in α. Even standard datatypes (e.g. lists, arrays) have theories of considerable size, not to speak of more realistic datatypes such as JAVA's `int` or `String` types. In consequence, at any time during a proof hundreds or even thousands of logical rules are applicable.

- The previous point is compounded by the necessity to formulate frequently needed properties of datatypes as explicit *lemmas* in order to keep the proof size manageable at all. It is a longstanding open problem in automated proof search to devise filters that keep only the "useful" of all found lemmas. In practice, lemma finding mechanisms often cause more harm than they help, because they increase the local search space even further.

From the late 1970s onwards frameworks for *interactive* theorem proving appeared that gave the user explicit control over the construction of proofs. They feature *meta languages* that contain dedicated control structures for the description of proof construction and for combination of new proof rules (lemmas). One of the earliest and still the most widely used meta language is the functional programming language ML. It introduced two important concepts: *tactics* are ML programs that add new rule applications to a given formal proof (this may involve search for sequences of rule applications that finish part of the current proof); *tacticals* are higher-order combinators that allow to compose new tactics from given ones.

Many interactive theorem provers (e.g. HOL, Nuprl, Coq, Isabelle) take a *foundational* approach: from a small set of *primitive* rules (for example, the axioms of ZF set theory) all other rules have to be proven. In some cases the system even enforces that only tactics and proofs can be derived which are justifiable relative to this small set of primitive rules.

Other provers (e.g. KIV, PVS, KeY) take a *pragmatic* approach. Here, the set of primitive rules is considerably larger; it is not fixed a priori and depends on the target programming language or even on the datatypes occurring in the program to be verified. Validity of rules typically is not enforced by the system,

[1] This situation is inevitable in software verification, because statements about non-trivial programming languages go beyond the expressivity of first-order logic. In program logics that directly represent the target programs one has typically to find suitable induction hypotheses while in program logics that encode programs as higher-order formulas, the problem reappears as higher-order quantifier elimination.

but can be optionally proven. A closer look reveals a number of quite different purposes that tactics are used for:

Proof search: for example, a tactic called "Blast_tac" [Paulson, 1998] is the main tool for automated proof search in Isabelle.

Derived Rules: these can be lemmas that capture properties of datatypes or optimised derivable rules only applicable in specialized contexts. They are introduced by special purpose tactics. In foundational systems, a justification must or at least should be included.

Heuristics: many tactics contain implicit heuristic information about which rule should be preferably applied in what order, etc.

While meta languages are a powerful concept, their practical application is prone to a number of criticisms.

Languages like ML are non-trivial to master. Moreover, they introduce an additional language level (on top of the target language and the logic framework). One has tried to achieve more uniformity by giving an operational interpretation to the underlying logical framework which is then used as its own meta language [Miller et al., 1991], but this severely restricts the choice of logical framework and introduces certain inefficiencies. In the present work we suggest a more radical solution, namely, to get rid of meta languages altogether.

This step is motivated by additional observations from many years of using verification systems in practice: powerful and effective tactics involving *proof search* are only written by experts that are very familiar with the underlying prover. It seems more efficient then, to realise proof search directly in the *implementation language* of the prover that is suitable for this task. On the other hand, by far most tactics serve to introduce lemmas or to provide efficient specialisations of general purpose rules.[2] In order to achieve this, the overhead of learning to encode target objects in higher-order logic or to learn a meta language seems too high. In other respects, meta languages do not offer *enough* concepts: side conditions and heuristic information (when to apply rules automatically, precedence) are represented implicitly (and are often "buried") in the code of tactics.

In this paper we suggest a new concept for implementing interactive theorem provers that we call *taclet*.[3] As the name suggests, taclets can be considered as lightweight, stand-alone tactics. They have a simple syntax and semantics which can be mastered in a matter of hours. Taclets have means to represent explicitly (i) the pure logical content of a rule; (ii) restrictions or *guards* on the expected context and position of a rule application; (iii) heuristic information on whether and when a rule is applied automatically/interactively.

An important consequence of the simplicity of taclets is that they can be schematically compiled into the GUI of the prover: then, for example, a mouse click over a term displays all interactively applicable rules at the position of the pointer. Only rules whose guard is satisfied and that can be successfully matched with the current position are displayed. This reduces drastically the cognitive burden on the user.

This paper is organised as follows: in the next section we give an informal introduction into the design philosophy of taclets, and we explain the basic concepts. In the following two sections we define formally syntax and semantics of taclets. In Sect. 5. we describe how the semantics of taclets affects a taclets-based prover from the user's point of view. As explained above, the taclets approach does not subscribe to a foundational design philosophy, but still one can (and should) address correctness. This is done in Sect. 6. Taclets are not merely a theoretical concept but are implemented and used successfully in the KeY [Ahrendt et al., 2004, 2002] system: Sect. 7. describes the implementation and Sect. 8. documents major case studies realised with the help of taclets. We close by giving related and future work.

[2]We are aware that the original intent of meta languages was the ability to "script" proofs (with intermittent search), but at least in our experience this is not the way in which meta languages are actually used.

[3]Taclets were first introduced under the name of *schematic theory specific rules (STSR)* by Habermalz [2000a,b].

2. Design and Concepts of the Taclet Language

2.1. Design Philosophy and Basic Definitions

Interactive proof systems are usually based on sequent-style calculi. We follow that approach, i.e., *taclets are a language for describing and implementing sequent calculi*.

Definition 1 *A* sequent *is of the form* $\Gamma \vdash \Delta$*, where* Γ, Δ *are duplicate-free lists of formulas. The left-hand side* Γ *is called* antecedent *and the right-hand side* Δ *is called* succedent *of the sequent.*

Usually (exceptions are rare), the semantics of a sequent $\Gamma \vdash \Delta$ is the same as that of the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$. Thus, in a sound and complete calculus, $\Gamma \vdash \Delta$ is derivable iff $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. Intuitively, the formulas in the antecendent are conjunctively connected; and the formulas in the succedent are disjunctively connected.

A further basic design philosophy of our taclet language is to give up the total generality of logical frameworks. Taclets support the large but restricted class of calculi for *first-order (multi-)modal logic* (with constant domains). The logics we consider must have vocabularies satisfying the following definition.

Definition 2 *A* vocabulary Σ *contains at least (a) a set of constant and function symbols and (b) a set of predicate symbols. It may also contain a finite set of sorts, in which case the constant, function, and predicate symbols may be sorted. In addition, there is a set of object variables.*

The formal syntax and semantics in Sections 3**.** and 4**.** is based on such vocabularies.

We trade generality for the advantages of a well-defined application domain. Before describing these advantages, we first note that the class of first-order modal logic is a good choice. It encompasses most logics used in the important and successful application areas of Logic in Computer Science and Logic in Artificial Intelligence, including: classical first-order predicate logic, Hoare logic and (first-order) dynamic logic, description logic, (first-order) temporal logic, propositional modal logic, etc.

The main type of logic not in the supported class is higher-order logic. The taclet language itself, however, *does* have higher-order concepts such that, for example, induction rules are easy to express.

The main advantage of only considering a restricted class of logics, is that a large part of the techniques and algorithms for proof search and construction can be implemented once and for all as part of the taclet system—even parts that in logical frameworks have to be implemented as tactics.

Most of the rules in calculi for first-order modal logic follow certain recurring patterns. They can be expressed schematically with a small number of concepts and mechanisms in our taclet language.

Definition 3 *A* rule *R is a binary relation between (a) the set of all tuples of sequents and (b) the set of all sequents. If $R(\langle P_1, \ldots, P_k \rangle,\ C)$ $(k \geq 0)$, then the* conclusion *C is* derivable *from the* premisses *$P_1, \ldots, P_k$ using rule R.*

A calculus *is a set of rules.*

Definition 4 *The set of sequents that are derivable in a calculus C is the smallest set such that: If there is a rule in C that allows to derive a sequent S from premisses that are all derivable in C, then S is derivable in C.*

A *proof* for a sequent S is a derivation of S written as an upside-down tree with root S.

The following definition makes use of the notion of *schema variables*. They represent concrete syntactical elements (e.g. terms or formulas). Every schema variable is assigned a type that determines which kind of concrete elements are represented by such a schema variable. Sect. 3**.** will define this concept in detail.

Definition 5 *A* rule schema *is of the form*

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_k}{C} \qquad (k \geq 0)$$

where $P_1, \ldots, P_k$ and C are schematic sequents, i.e., sequents containing schema variables.

A rule schema $P_1 \cdots P_k \,/\, C$ represents a rule R if the following equivalence holds: a sequent C^* is derivable from premisses $P_1^*, \ldots, P_k^*$ iff $P_1^* \cdots P_k^* \,/\, C^*$ is an instance of the rule schema. Schema instances are constructed by instantiating the schema variables with syntactical constructs (terms, formulas, etc.) which are compliant to the types of the schema variables.

As usual, one rule schema represents infinitely many rules, namely, its instances.

The basic actions in proof construction that are used in our taclet language (and turn out to be sufficient to implement most rules for first-order modal logic) are:

- A sequent is recognised as an axiom, and the corresponding proof branch is closed.
- A formula in a sequent that is a rule premiss is modified. A single formula (in the conclusion of the rule) is chosen to be in focus. It can be modified or deleted from the sequent. Note, that we do not allow more than one formula to be modified by a rule application.
- Formulas are added to a sequent. The number of formulas that are added is finite and is the same for all possible applications of the same rule schema.
- The proof branches. The number of new branches is the same for all possible applications of the same rule schema.
- Whether the rule schema is applicable and what the result of the application is, may depend on the presence of certain formulas in the conclusion.

The core of the taclet language, i.e., the constructs for using the above schematic concepts, are described in Sect. 2.2. They are sufficient to implement a sound and complete sequent calculus for propositional logic, as well as rules for theory reasoning and equality rewriting.

There are often cases, however, where the basic concepts listed above are not sufficient for describing a rule. Even if its general form adheres to the above patterns, there may be details in a rule that cannot be expressed schematically. For example, in rules for handling first-order quantifiers, there is usually a restriction that variables or (Skolem) constants introduced by a rule application must not already occur in the sequent. When a rule is described schematically, such constraints are usually added as a note to the schema.

To express constraints and other rule features that are not expressible in a schematic way, additional concepts are introduced in Sect. 2.3. With these, the taclet language can be used to describe calculi for first-order and non-classical (modal) logic. The feasibility of describing calculi for quite complex logics is shown by our implementation of a dynamic logic for JAVA CARD, as well as other case studies (see Sect. 8.)

Due to the restriction that a rule application can only modify a single formula, labelled deduction has to be used for modal logic. Calculi that implement state transitions by deleting all formulas not true in a successor state cannot be implemented. This is not a principal problem as there are labelled deduction calculi for all important modal logics [Gabbay, 1996].

Furthermore, we introduce constructs into the taclet language that increase the efficiency of automation and improve user interaction. They are explained in Sect. 2.3.2. and 2.3.3.

2.2. Basic Concepts

A very basic information contained in a derivation rule is which term or formula is modified by applying the rule. The taclet language offers the keyword `find` to express this information. Using this keyword it is already possible to write axioms in taclet notation. Axioms are rules without premisses and allow closing branches in a proof tree. The fact that a taclet is an axiom is expressed using the keyword `close goal`.

The first example is a simplified[4] taclet representing an axiom that allows to close a branch in a proof provided that its leaf is labelled with a sequent containing a conjunction of an arbitrary formula ϕ and its negation $\neg\phi$ in the antecedent.

$$\texttt{find } (\phi \wedge \neg\phi \vdash\ ) \texttt{ close goal}$$

The expression within parentheses after the keyword `find` defines a pattern that must be matched (in this taclet ϕ matches formulas) in the actual proof problem in order for the taclet being applicable. This means, the find-clause defines where the taclet and, therefore, its logical information is applicable.

The expression in the find-clause contains *schema variables*. A schema variable has a type that defines which expression the variable can stand for (a precise definition is given in Sect. 3.1.) In our example, the schema variable ϕ can stand for an arbitrary formula. The taclet language has a set of built-in types of schema variables that are necessary for implementing any kind of logic, e.g. types for matching variables, terms, and formulas.

For being able to express derivation rules that modify a term or a formula (specified in the find-clause) the taclet language offers the keyword `replacewith`.

As an example for a taclet using the replacewith-clause we consider a simple derivation rule for propositional logic. If one wants to prove the validity of an implication $\phi \to \psi$, i.e. derivability of the sequent $\vdash \phi \to \psi$, one has to show that ϕ is false or ψ is true. In taclet notation this corresponds to

$$\texttt{find } (\ \vdash \phi \to \psi) \texttt{ replacewith}(\phi \vdash \psi). \tag{1}$$

When this taclet is applied a new proof goal is created from the previous one by replacing the expression matched in the find-part with the accordingly instantiated expression in the replacewith-part.

Fig. 1 shows how the information expressed in the find-clause is automatically compiled into the graphical user interface: The user points with the mouse on the term or formula he or she wants to modify and the focused term or formula is highlighted by the system automatically (see Sect. 7.1.). A mouse-click offers the set of taclets that are applicable to the focused term or formula. The figure also shows that the actual taclet is displayed as a tool-tip when the user points on a taclet name with the mouse cursor.[5] See [Giese, 2004] for a discussion of the graphical user interface of the KeY system.

Besides rules that modify a term or a formula there are rules that add formulas (but not terms) to a sequent. In the taclet language this can be expressed using the keyword `add`. An example for a taclet that requires the add-clause is the so-called *cut-rule*. This rule has two premisses and allows to eliminate a formula that is contained in the antecedent of one premiss and in the succedent of the other. If applied from bottom to top this rule allows to add arbitrary formulas to the proof (hence, the keyword "add").

$$\texttt{add } (\phi \vdash\ )\texttt{; add } (\ \vdash \phi)$$

Note that the cut-rule taclet does not contain a find-clause, i.e. the taclet is always applicable.

As already mentioned, the cut-rule has two premisses which are reflected by the two add-clauses in the taclet. The consequence of applying a taclet with several replacewith- and/or add-clauses is that the proof is split into several subproofs, i.e., the proof branches.

The above examples contained exclusively either add- or replacewith-clauses, however, it is legal to use both in one and the same taclet.

For the soundness of some rules the context formulas in the sequent, i.e. the formulas except for the one in focus, are crucial. Consider as an example the rule which states, that in order to show that $\phi \to \psi$ holds it is sufficient to prove ψ *if* we know that ϕ holds or, speaking in terms of a sequent calculus, that ϕ is contained in the antecedent. This informal description of the rule has already introduced the keyword `if`

[4]In this section we omit the names of taclets and use usual mathematical notation for denoting their the logical content.

[5]The taclet shown in the tool-tip in Fig. 1 is written in the concrete syntax used in the KeY system. E.g., the sequent symbol is denoted by ==> instead of $\vdash$.

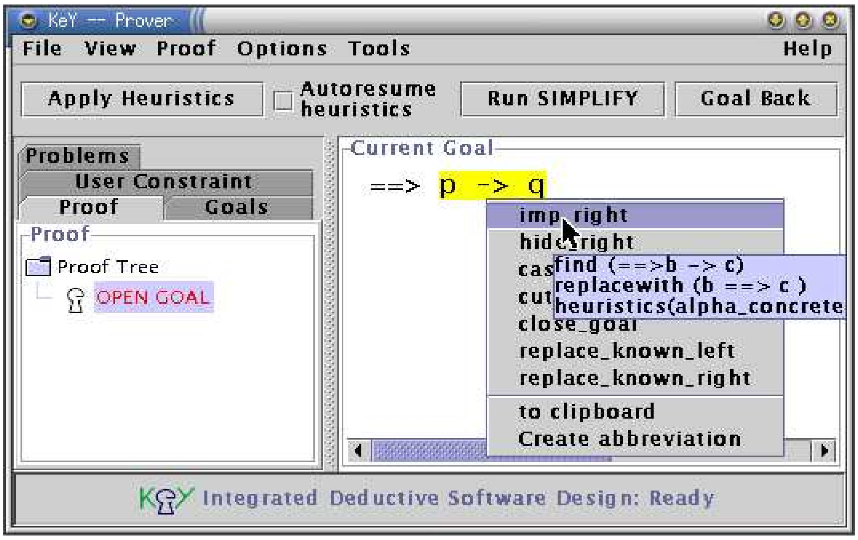

Figure 1. Applicable Taclets for the Formula in Focus

that allows us to express a condition on the applicability of a taclet. Thus, in taclet notation this rule looks like

$$\texttt{if } (\phi \vdash\ ) \texttt{ find } (\ \vdash \phi \to \psi) \texttt{ replacewith } (\ \vdash \psi).$$

If ϕ is contained in the antecedent and $\phi \to \psi$ in the succedent of a sequent then this taclet is applicable and $\phi \to \psi$ can be replaced with ψ. If ϕ is not contained in the antecedent the user can first apply the cut-rule for ϕ. As a result the proof is split into two branches: In the one branch it has to be shown that ϕ can be deduced from the formulas contained in the antecedent and in the second branch ϕ is added to the antecedent which then allows to apply the above taclet. Such a situation occurs quite often and it is tedious to first apply the cut-rule in order to satisfy the condition in the if-clause. Taclets offer a neat way to overcome this inconvenience by allowing the application of taclets even if the if-clause is not satisfied. To ensure soundness the derivability of the formula in the if-clause has to be established in a subproof. Thus, the application of a taclet with a non-matched if-clause can be seen as an implicit application of the cut-rule.

The concepts we have presented in this section are sufficient to implement a complete sequent calculus for propositional logic within a few minutes.

2.3. Advanced Concepts

In this section we present advanced concepts of the taclet language. First, we show how quantifiers and substitution in first-order logic can be handled. Then we explain concepts for increasing efficiency and improving user interaction. Finally, we show an example of a dynamic logic calculus implemented with taclets.

2.3.1. First-order Concepts

As an example we consider the rule for handling universal quantifiers in first-order logic. In usual textbook notation the rule looks as follows:[6]

$$\frac{\forall x.\phi,\ \phi_x^t\ \vdash}{\forall x.\phi\ \vdash}$$

[6]For simplicity we omitted the schema variables denoting the side formulas of the sequent, i.e. the formulas that are not affected by the rule.

For the soundness of this rule it is crucial that term t does not contain free variables that become bound in ϕ after substituting t for x. To avoid those problems a sufficient restriction is to forbid free variables at all (see Sect. 3.2.). Then, in taclet notation this restriction does not have to be specified explicitly and the quantifier rule can be written as

$$\texttt{find (}\forall x.\phi \vdash \texttt{) add (}\phi_x^t \vdash \texttt{).} \tag{2}$$

When this taclet is applied an instantiation dialog pops up asking the user for the desired instance of term t. Either the user fills in the term manually or, if the desired term is already contained in the sequent, he or she can also use drag-and-drop instead (we will mention more features improving user interaction in Sect. 2.3.3.)

Sometimes a substitution requires a condition on the variables involved, e.g. that a variable introduced must be "new" or that a variable may not occur free in some term. For this purpose, the taclet language offers the keyword `varcond` followed by the required condition, e.g. `varcond (`x `not free in` ϕ`)`.

2.3.2. Concepts for Automation

So far we have presented taclets that can only be applied interactively, however, for increased efficiency automation is required.

Taclets can be marked for automatic application using the keyword `heuristics` followed by a list of names denoting heuristics this taclet should belong to. If one of those heuristics is activated by the user this taclet can be applied automatically. Consider the following example, showing a taclet that simplifies arithmetic expressions by replacing $x + 0$ by x.

$$\texttt{find (}x+0\texttt{) replacewith (}x\texttt{) heuristics (simplify, arithmetic)}$$

This taclet can be applied automatically if the user activates the heuristic "simplify" or "arithmetic". Which taclet is applied next is determined by a feature vector which is evaluated by the taclet interpreter. The feature vector is a weighted sum of priorities which can be assigned to heuristics and a parameter which guarantees fairness, i.e. every applicable rule is eventually applied.

The concept of automatic taclet application allows the definition of domain-specific heuristics, e.g. for integer arithmetic, which can be turned on and off by the user if required.

Note, that the previous taclet is the first example for a taclet where the find- and replacewith-clause do not contain the sequent symbol $\vdash$ as it was the case in all the examples before. The crucial difference is that a find-clause without sequent symbol can match arbitrarily *nested* terms or formulas on either sides of the sequent, whereas a find-clause containing a sequent symbol merely matches *top level* formulas on the side of the sequent defined in the find-clause.

2.3.3. Concepts for Improving User interaction

In the previous sections we already mentioned some of the concepts that are designed for improving user interaction:

- Selection of the term or formula in focus is done by pointing with the mouse cursor on it.
- The term or formula currently in focus is highlighted automatically.
- Only taclets that are applicable to the term or formula in focus are offered to the user.
- Drag-and-drop can be used for specifying the arguments in instantiation dialogues.

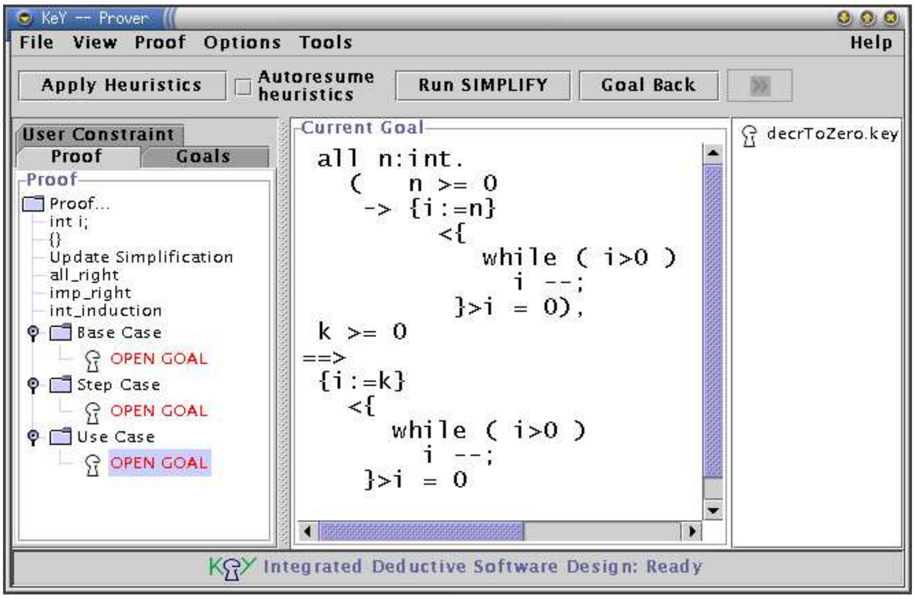

Figure 2. Annotated Proof Tree after Applying Taclet for Induction

In the following we introduce additional concepts that have not been mentioned yet.

In Sect. 2.3.1. we presented a taclet for handling universal quantifiers in the antecedent. Another point of view on the quantifier rule is to focus on the term to instantiate—provided it is contained in the sequent—and not on the quantified formula, i.e. the user points and clicks on the term he or she wants to instantiate and not on the quantified formula. Then, the system offers possible choices of quantified formulas[7] that can be instantiated with the selected term. If there is only one possibility no user interaction is required. A taclet expressing this point of view is

$$\texttt{if}\ (\forall x.\phi \vdash\ )\ \texttt{find}\ (t)\ \texttt{add}\ (\phi^t_x \vdash\ ).$$

Instead of clicking on the desired term and then choosing the quantifier taclet, the application of this taclet can also be triggered by drag-and-drop: the user drags the desired term and drops it onto the quantified formula he or she wants to instantiate. This feature demonstrates how taclets are "compiled" into the graphical user interface of the prover improving the user interaction significantly.

Often proofs are branching heavily, in particular proofs in program verification, which makes it difficult to keep track of the whole proof. To diminish this problem the implementation of the taclet language in the KeY system allows to annotate add- and replacewith-clauses with labels that are displayed in the nodes of the proof tree after the taclet is applied. Fig. 2 shows a proof tree after applying the following taclet for induction where the three cases are labelled with "Base Case", "Step Case", and "Use Case".

$$\begin{array}{ll}
\texttt{"Base Case":} & \texttt{add}\ (\ \vdash \phi^0_n)\texttt{;} \\
\texttt{"Step Case":} & \texttt{add}\ (\ \vdash \forall n.(n \geq 0 \land \phi \rightarrow \phi^{n+1}_n))\texttt{;} \\
\texttt{"Use Case":} & \texttt{add}\ (\forall n.(n \geq 0 \rightarrow \phi) \vdash\ )
\end{array}$$

Finally, we compare the notation of sequent-style derivation rules in usual textbook notation (see Def. 5) and in taclet notation. As one can see from the example in Fig. 3, taclets describe merely the formulas or terms that are affected by the rule whereas the textbook-style notation also provides schema variables Γ, Δ

[7] The system only offers formulas whose quantified variable is sort-compatible with the selected term.

$$\texttt{find (} \forall x.\phi \vdash \texttt{) add (} \phi^t_x \vdash \texttt{)}$$

$$\frac{\Gamma, \forall x.\phi, \phi^t_x \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}$$

Figure 3. Textbook Notation vs. Taclet Notation of a Sequent-style Derivation Rule

for the remaining formulas in the sequent. Another difference is that rules in textbook-style notation are written from top to bottom but are applied from bottom to top in practice. In contrast, taclets are written in the same way as they are applied. It is simple to translate automatically from taclet notation into textbook notation. This allows to generate the documentation of the calculus rules in the usual textbook notation automatically from taclet files. Moreover, this ensures that the documentation and the actual set of taclets are always in sync.

2.3.4. Concepts beyond First-order

An advanced feature of the taclet language allows the introduction of additional taclets through the application of a taclet containing the keyword `addrules` followed by a template for the new taclet. Consider the taclet

$$\texttt{find (} s \doteq t \vdash \texttt{) addrules (}\{\texttt{find (} s \texttt{) replacewith (} t \texttt{)}\}\texttt{)} \tag{3}$$

which is an example for how equality can be handled. If the antecedent contains an equality that can be matched by $s \doteq t$ then applying this taclet results in a new taclet which allows to replace a term matched by s with a term matched by t. For example, applying the above taclet to the formula $x \doteq 0$ in the antecedent of some sequent results in the additional taclet

$$\texttt{find (} x \texttt{) replacewith (} 0 \texttt{)}.$$ [8]

Due to the addrules-clause the set of taclets is not fixed but can grow dynamically in the course of a proof. Note, that the generated taclets are not sound in general but only in the context where the taclet containing the addrules-clause was applied (soundness of taclets is discussed in Sect.6.)

So far we have shown examples for taclets representing calculus rules for propositional and first-order logic. However, taclets are not restricted to these two logics and in the remainder of this section we will present an example of a taclet from a calculus for dynamic logic for JAVA CARD [Beckert, 2001]. This program logic, for short JAVA CARD DL, is the logical basis of the KeY system and allows reasoning about JAVA CARD programs.

The example we present is a taclet for handling the if-then-else statement in JAVA CARD.

$$\begin{array}{l} \texttt{find (}\langle\texttt{.. if (\#se) \#s0 else \#s1...}\rangle\phi\texttt{)} \\ \texttt{replacewith (} \;\; \texttt{\#se} \doteq \mathit{true} \rightarrow \langle\texttt{..\#s0...}\rangle\phi \\ \qquad\qquad\qquad\quad \wedge\, \texttt{\#se} \doteq \mathit{false} \rightarrow \langle\texttt{..\#s1...}\rangle\phi\texttt{)} \end{array} \tag{4}$$

The basic idea of the taclet is to split the if-then-else statement into two statements representing the possible branches (see the replacewith-clause). As you can see, JAVA CARD DL formulas may contain actual program code within its *diamond* operators $\langle\cdot\rangle$[9], in our case an extension of JAVA CARD called *schematic* JAVA CARD. This extension of JAVA CARD which is defined in Sect. 3.3. allows a JAVA CARD program to

[8] As will be explained in Def. 19, in fact the taclet `find (`s`) replacewith (`t`)` is added to the set of available taclets and the instantiations x for s and 0 for t are attached to the new taclet. However, the user does not notice the difference.

[9] Informally speaking, the formula $\langle p\rangle\phi$ states that program p terminates and formula ϕ holds in the program's final state(s).

contain schema variables (in the example, `#se`, `#s0`, and `#s1` are schema variables) which match certain JAVA CARD expressions. In the example, `#se` stands for expressions without side-effects and `#s0`, `#s1` stand for arbitrary JAVA CARD statements.

As already mentioned, the taclet language contains a set of pre-defined schema variable types for basic constituents like terms and formulas. The schema variables for JAVA CARD expressions can be seen as a user-defined extension of the type system of schema variables to cope with advanced features of a specific logic. This shows the flexibility of the taclet language and, in fact, replacing JAVA CARD with another language would be relatively easy.

Another feature of taclets for JAVA CARD DL, that is explained in greater detail in Sect. 4.3., is the use of `..` and `...` in (4). Roughly speaking, these elements determine that taclet (4) is applicable at a formula that possibly has, e.g., opening braces or opening `try`-blocks *before* an `if` statement of the appropriate form, and that possibly has arbitrary statements, closing braces, or `catch`-blocks *after* it. This corresponds to the fact that the JAVA CARD DL calculus is always working on the first *active* statement of the program contained in a modal operator [Beckert, 2001].

3. Formal Taclet Syntax

As already pointed out, the concept of taclets is applicable to every sorted first-order (modal) logic. The syntax of the logic of interest (for which taclets are defined) is at least required to have a sorted first-order vocabulary Σ containing function, predicate, and sort symbols. For the time being, sorts are assumed to be flat (i.e. unordered), a restriction which is (concerning the taclet concept) easily changeable. Furthermore, the logic provides junctors $\mathcal{J}$, quantifiers $\mathcal{Q}$, and modal operators $\mathcal{M}$. A sorted first order logic can be regarded as a simple instance of this characterisation.

In the next subsection a basic version of taclets for such a logic is introduced. Minor modifications of the taclet definition turn out to be necessary if more specific logics are considered. By example of the dynamic logic JAVA CARD DL we describe these subsequently.

3.1. A Basic Version of Taclets

Crucial to the formal definition of taclets is the notion of *schematic terms*, *schematic formulas*, and *schematic sequents*. Essentially, they are just normal terms, formulas, or sequents (resp.) of a logic of interest which contain *schema variables* as additional elements. Schema variables are distinct from the elements of the vocabulary. Schematic terms, formulas, and sequents are defined in the same way as it would have been done for the respective elements of the original logic. A *type* is assigned to each schema variable which is needed to determine how schematic elements are mapped to concrete elements, when a taclet is applied. We allow schema variables to occur as basic terms and, if they have the special type Formula, as basic formulas. Furthermore we disallow the occurrence of object variables and allow only special schema variables (of type Variable) to be bound by quantifiers and substitution operators.

Definition 6 *A* schema variable *is a symbol distinct from any other symbol of the first-order vocabulary. A* schema variable type *is assigned to each schema variable which is either* Term*,* Variable *or* Formula*. If the type is* Term *or* Variable *the schema variable is additionally assigned a sort of the vocabulary.*

Given a set SV of schema variables and the vocabulary Σ, the set $STerm_{SV}$ of schematic terms *is defined to be the smallest set such that*

- *for all $sv \in SV$ of type* Term *or* Variable*: $sv \in STerm_{SV}$;*
- *for all function symbols $f \in \Sigma$ of arity n and argument sorts $S_1, \ldots, S_n$ and for all $t_1, \ldots, t_n \in STerm_{SV}$ with sorts $S_1, \ldots, S_n$: $f(t_1, \ldots, t_n) \in STerm_{SV}$.*

Given SV and Σ, the set $SFor_{SV}$ of schematic formulas is the smallest set such that:

- *for all* $sv \in SV$ *of type* Formula*:* $sv \in SFor_{SV}$*,*
- *for all predicate symbols* $p \in \Sigma$ *of arity* n *and argument sorts* $S_1, \ldots, S_n$ *and for all* $t_1, \ldots, t_n \in STerm_{SV}$ *with sorts* $S_1, \ldots, S_n$*:* $p(t_1, \ldots, t_n) \in SFor_{SV}$*,*
- *for all* $Q \in \mathcal{Q}$*, all* $\phi \in SFor_{SV}$*, and all* $sv \in SV$ *of type* Variable*:* $Qsv.\phi \in SFor_{SV}$*,*
- *for all* $sv \in SV$ *of type* Variable*,* $t \in STerm_{SV}$ *(with the same sort as* sv*), and* $\phi \in SFor_{SV}$*:* $\phi^t_{sv} \in SFor_{SV}$ *(the substitution operator),*
- *for all junctors* $\circ \in \mathcal{J}$ *of arity* n *and all* $\phi_1, \ldots, \phi_n \in SFor_{SV}$*:* $\circ(\phi_1, \ldots, \phi_n) \in SFor_{SV}$*,*
- *for all modal operators* $m \in \mathcal{M}$ *of arity* n *and all* $\phi_1, \ldots, \phi_n \in SFor_{SV}$*:* $m(\phi_1, \ldots, \phi_n) \in SFor_{SV}$*.*

A schematic sequent *for schema variables* SV *and a signature* Σ *is of the form* $\Gamma \vdash \Delta$ *of duplicate-free finite sequences* Γ *(the antecedent) and* Δ *(the succedent) over the closed formulas of* $SFor_{SV}$*. All schematic sequents for* SV *and* Σ *form the set* $SSeq_{SV}$*.*

As has been shown in Sect. 2.3.4., taclets can syntactically contain other taclets. Hence, definitions 7 to 9 that formally define taclets are simultaneous.

Schematic formulas and terms occur in taclets to describe both, the applicability of the taclet as well as the effects that its application will have. The elements of this "effect part" of a taclet are called *goal templates* and are defined as follows:

Definition 7 *Let* SV *be a set of schema variables,* $rw \in SSeq_{SV} \cup STerm_{SV} \cup \{\bot\}$*,* $add \in SSeq_{SV}$*, and be* $addTaclets$ *a set of taclets (see Def. 9). Then* $(rw, add, addTaclets)$ *is a* goal template *over* SV*. At least one of* rw*,* add*, or* $addTaclets$ *has to be non-empty however, i.e. the tuple* $(\bot, \vdash, \emptyset)$ *is disallowed. In concrete syntax we write*

`replacewith (`$[rw]$`) add (`$[add]$`) addrules (`$[addTaclets]$`)`

if $[rw]$*,* $[add]$ *are the representations of* rw*,* add *(resp.) in concrete syntax and* $[addTaclets]$ *is a comma-separated list of taclets in concrete syntax representing* $addTaclets$ *(each taclet either attached with a name or enclosed by curly brackets). If* $rw = \bot$*,* $add = \ \vdash$*, or* $addTaclets = \emptyset$ *the respective part is empty in concrete syntax.*

Goal templates in taclets comprise knowledge about how new goals are supposed to be built when a taclet is applied. The basis of constructing these goals is always one given goal which is mainly characterised by the *find-part* of a taclet.

Note, that the $addTaclets$ set of a goal template may contain taclets with goal templates which themselves have a non-empty $addTaclets$ set, and so on. It is not clear whether such taclets are useful, but they are well-defined.

To avoid invalid instantiations there is another small but vital part of taclets called *variable conditions*.

Definition 8 *Suppose* SV *is a set of schema variables. If* $var \in SV$ *is of type* Variable*,* $sv_0 \in SV$ *of type* Term *or* Formula *and* $sv_1 \in SV$ *of type* Term *then* variable conditions *over* SV *are:*

- var `not free in` sv_0
- sv_1 `new depending on` sv_0

Another main constituent of taclets (apart from goal templates, find-part and variable conditions), the *if-part*, captures conditions on the original sequent which must additionally be valid. The *heuristics-part* contains information on how taclets are to be applied in automated mode (see Sect. 5.2.). From the syntax perspective it is sufficient to require that there is a fixed set of names for heuristics available.

Taclets are designed to be used in an *interactive* prover. Thus, in the KeY system a *name* is required to help users to understand the effect of a taclet. In this formal description however, names are omitted. All parts are optional, which is represented by the symbol $\perp$, the empty set $\emptyset$, or the empty sequent $\vdash$.

Definition 9 *Suppose SV is a set of schema variables, GT is a set of goal templates over SV, the find-part $f \in STerm_{SV} \cup SFor_{SV} \cup SSeq_{SV} \cup \{\perp\}$, the if-part $ifseq \in SSeq_{SV}$, VC is a set of variable conditions over SV, and H is a set of heuristics names. Then $(f, ifseq, VC, GT, H)$ is a* taclet *over SV if the following conditions hold:*

- *If $f \in SSeq_{SV}$ then f contains exactly one top level formula.*
- *f and GT are compatible, i.e. for all $(rw, add, addTaclets) \in GT$:*
  If $f \in SSeq_{SV}$ then $rw \in SSeq_{SV} \cup \{\perp\}$,
  if $f \in STerm_{SV}$ then $rw \in STerm_{SV} \cup \{\perp\}$ and (if $rw \neq \perp$) the sorts of f and rw are the same,
  if $f \in SFor_{SV}$ then $rw \in SFor_{SV} \cup \{\perp\}$,
  if $f = \perp$ then $rw = \perp$.
- *and the additional syntactical conditions as described in Sect. 3.2.*

A taclet is written in concrete syntax as follows:

`if (`$[ifseq]$`) find (`$[f]$`) varcond (`$[VC]$`)` $[GT]$ `heuristics (`$[H]$`)`

Here, $[GT]$ is a semicolon-separated list of goal templates in the syntax of Def. 7; $[H]$ and $[VC]$ are comma-separated lists of heuristics and variable conditions; $[ifseq]$ and $[f]$ are the representations for $ifseq$ and f (resp.). As with goal templates, if a part equals $\perp$ or $\emptyset$ the whole part is skipped in concrete syntax, if $ifseq = \vdash$ the if-part is skipped. If $GT = \emptyset$, we write `close goal` *in concrete syntax.*
The name n for a taclet is made explicit in concrete syntax as follows: n `{ ... }`.

Apart from these constituent parts, taclets can contain additional information to ease the interaction with a user, such as a more extensive display name, labels for each goal template, etc., as pointed out in Sect. 2.3.3.

3.2. Additional Syntactical Conditions on Taclets

This subsection imposes additional conditions on taclets which basically exist to exclude certain taclets which would, if applied, destroy well-formedness properties of the resulting formulas, or that would produce other undesirable effects. The situation is comparable to static type checking of programs: Problematic programs (or taclets) are detected earlier than at run time. If realisations of taclets choose to allow the existence of taclets that are never applicable and opt to to take care of the occurring problems at run time, then ignoring the conditions of this section is permitted. Since even users with little experience with formal logic should be enabled to define new taclets (which are then verified with methods explained in Sect. 6.), the check of these conditions is recommended, however.

Moreover, the conditions presented here depend on the desired properties of the resulting sequents: If free logic variables in top level formulas are not allowed (as in the KeY system), the last two conditions can be required, otherwise they do not make sense to be imposed.

Condition 1 (Avoid ambiguous variable binding) *Let v be a schema variable occurring in a taclet t with type* Variable*, $ifseq$ the if-part of t and f its find-part. There can be at most one quantifier that binds an occurrence of v in $ifseq$ or f.*

With this condition it is taken into account that the effect of binding variables is well-defined. A taclet `find(`$\forall v.b \wedge \neg\forall v.b \;\vdash\;$`) close goal` is certainly expected to be applicable at the antecedent of a sequent $\forall x.p(x) \wedge \neg\forall y.p(y) \;\vdash$ though no matching instantiation of v exists; v would have to be mapped to both x and y. Such ambiguous situations are simply avoided by prohibiting the above taclet by applying the condition from above.

Condition 2 (Prevent introduction of free variables) *Let v be a schema variable occurring in a taclet t with type* Variable. *All occurrences of v in t must be bound.*

Together with the fact that only schema variables of type Variable are bound in schematic formulas and the next condition, this prevents variables to occur freely in a result of a taclet application.

Condition 3 (Prevent introduction of free variables) *Let t be a taclet and VC its variable conditions. The* prefix $\Pi(\underline{sv})$ *of an occurrence $\underline{sv}$ of a schema variable sv with type* Formula *or* Term *in t is defined as*

$$\begin{array}{rcl} \Pi(\underline{sv}) & = & \{v \mid v \textit{ is of type } \mathsf{Variable}, sv \textit{ is in the scope of } \underline{v}\} \\ & \setminus & \{v \mid (v \;\texttt{not free in}\; sv) \in VC\}\,. \end{array}$$

Suppose $\underline{V}$ are all occurrences of a schema variable v in a t with type Formula *or* Term. *Then, for all $\underline{v}_1, \underline{v}_2 \in \underline{V}$, the prefixes of $\underline{v}_1$ and $\underline{v}_2$ must be the same, i.e. $\Pi(\underline{v}_1) = \Pi(\underline{v}_2)$, and we can define the prefix of v as $\Pi(v) := \Pi(\underline{v}_1)$.*

This condition prevents the introduction of a taclet like `find(`$\forall sv.b$`) replacewith(`b`)`, which possibly introduces free logic variables into a top level formula. Note, that this taclet can easily be made valid by adding the variable condition sv `not free in` b. With this extension, it would remove "unused" all-quantifiers.

3.3. Taclets for JAVA CARD DL

If taclets are equipped with further types of schema variables and special variable conditions they can easily be adapted to cope with logics that have special modal operators and calculi that have rules specific to these modal operators. Here, we describe some issues of the syntax of taclets for JAVA CARD DL [Beckert, 2001].

JAVA CARD DL is captured by the characterisation of logics for which taclets are defined in Sect. 2.2. The speciality of this logic is that the modalities $\mathcal{M}$ consist of *base modal operators* $\mathcal{M}^0$ which are parameterised with sequences of JAVA CARD statements. For example, for the base modal operator $\langle\cdot\rangle \in \mathcal{M}^0$ and a (legal) sequence α of JAVA CARD DL statements, $\langle\alpha\rangle \in \mathcal{M}$. Schematic transformations within these programs α have to be covered by taclets as well in order to provide means to implement a JAVA CARD DL calculus [Beckert, 2001] with taclets. Moreover, like taclets must introduce new symbols as in Def. 8, it is also necessary that taclet applications introduce new, so far unused program variables. Therefore additional variable conditions are provided, which also make a new program variable to have a certain JAVA type.

We essentially refine the definitions 6 to 9. If it is not clear from the context the suffix "…for JAVA CARD DL" is used to refer to the re-definitions.

Definition 10 *Schema variables are defined as in Def. 6 but schema variables may have (in addition to those defined there) one of these additional schema variable types*[10]*:* ProgramVariable, SimpleExpression, NonSimpleExpression, Expression, Statement.

Possible variable conditions are those defined in Def. 8 and the following ($pv_0, pv_1 \in SV$ of type ProgramVariable*, t a primitive* JAVA CARD *type):*

- `typeof`*(*pv_1*)* pv_0 `new`

[10] We only give an extract of the types existing in the KeY system. There, a few more special-purpose schema variable types have been defined.

- t pv_0 `new`.

Schematic terms are defined as in Def. 6.

JAVA CARD DL contains JAVA CARD programs as parameters of its modal operators. In order to process such programs by taclets we introduce *SchemaJava* programs in the definition of schematic formulas which provide schematic elements in JAVA CARD programs as it was done for the basic version of taclets. As will be pointed out later (in Sect. 4.3.), there is a need for another extension, which is called *meta construct*. Such constructs resemble functions in having a fixed number of parameters. There are only a few predefined meta constructs available. Both, meta constructs and schema variables that can occur in SchemaJava are denoted in such a way that they can be distinguished from other regular JAVA program elements, e.g. by making use of special non-JAVA identifiers that all start with a #-sign.

For the purpose of the subsequent definition we abstract from the concrete JAVA CARD syntax and assume JAVA CARD program elements to be given as abstract syntax trees.

Definition 11 *Suppose JavaProgElem is the set of all* JAVA *program elements.*
The set $SchemaJava^0_{SV}$ over a set SV of schema variables is defined to be the smallest set such that

- *if $\alpha \in JavaProgElem$ and α' is obtained by replacing in α arbitrary (or no) subtrees by an arbitrary $sv \in SV$, then $\alpha' \in SchemaJava^0_{SV}$,*
- *if mc is a meta construct of arity n, $\alpha_1, \dots, \alpha_n \in SchemaJava^0_{SV}$ then $mc(\alpha_1, \dots, \alpha_n) \in SchemaJava^0_{SV}$.*

Then, the set of SchemaJava programs *over SV is defined as follows*

$$SchemaJava_{SV} = SchemaJava^0_{SV} \cup \{\, \texttt{..}\,\alpha\,\texttt{...} \mid \alpha \in SchemaJava^0_{SV}\}$$

The construct $\texttt{..}\,\alpha\,\texttt{...}$ is called schematic program context *(see Sect. 4.3.).*

For the conciseness of the SchemaJava definition, conditions on where schema variables of certain types can be placed have been omitted here: Schema variables of type Statement should, for instance, be required to occur only where JAVA statements are expected.

Finally, the definition of schematic formulas and taclets can be concluded in the natural way. The definition depends of course on the modal operators of the underlying JAVA CARD DL.

Definition 12 *Schema formulas for* JAVA CARD DL *are defined as in Def. 6 with this modification: The modal operators $\mathcal{M}$ consist of the base modal operators $\mathcal{M}^0$ parameterised with sequences of SchemaJava statements. Schematic sequents, goal templates and taclets are defined as in the rest of definitions 6 to 9.*

4. Semantics

The semantics of taclets is to a high degree related to the user interaction model of a taclet-based interactive prover. Since it should be the responsibility of a concrete prover realisation to determine how interaction is designed, the semantics of a proof system based on taclets should not be described in all details. However, there are some fixed obligatory rules to such a system, which are described in this and the next section. First, some basic definitions are given that introduce the terms which enable us to describe how taclets are applied. The definitions are almost general enough to handle taclets for JAVA CARD DL. However, they have to be slightly adapted to capture the fact that instantiations to JAVA CARD programs are possible and to reflect the semantics of the schema variables and variable conditions for JAVA CARD DL. There are also two more substantial extensions to handle a complex program logic like JAVA CARD DL, which we will investigate in Sect. 4.3.

The basic notions to describe the behaviour of taclets are *instantiation* and *application*. They denote the two basic phases of working with taclets: first, it is determined if a taclet is applicable and under which circumstances, i.e. how the schematic terms are mapped to concrete ones occurring in the goal being worked on, and second, a taclet is applied producing a number of new goals (with possibly more taclets attached).

$type_{sv}$	condition on an instantiation $\iota(sv)$
Variable	$\iota(sv) \in V$ is of sort $sort_{sv}$
Term	$\iota(sv) \in Term_V$ is of sort $sort_{sv}$
Formula	$\iota(sv) \in For_V$
ProgramVariable	$\iota(sv) \in JavaProgElem$ is a program variable or a static field reference with side effect free prefix
SimpleExpression	$\iota(sv) \in JavaProgElem$ satisfies the ProgramVariable condition or is a (negated) literal or is an `instanceof` expression with a program variable or `null` as first argument
NonSimpleExpression	$\iota(sv) \in JavaProgElem$ is an expression and does not satisfy the SimpleExpression condition
Expression	$\iota(sv) \in JavaProgElem$ is an expression
Statement	$\iota(sv) \in JavaProgElem$ is a statement

Table 1. Conditions on the instantiations of schema variables

4.1. Instantiations

An instantiation is first of all a (partial) map from schema variables to concrete terms and formulas for which certain conditions depending on the type and the sort of the schema variable have to hold. Given the first-order signature and a set V of object variables that can be bound by the quantifiers, the concrete terms, formulas and sequents are called $Term_V$, For_V, and Seq_V. When taclets are applied, we are interested in such instantiations that map *all* schema variables of the taclet because otherwise schema variables would possibly intrude into concrete sequents.

Definition 13 *An* instantiation *of a set of schema variables SV is a partial map*

$$\iota : SV \longrightarrow Term_V \cup For_V$$

if, for all $sv \in SV$ and their types $type_{sv}$ (and sorts $sort_{sv}$), $\iota(sv)$ satisfies the conditions described in Table 1 for $type_{sv}$ (and $sort_{sv}$).
The instantiation is complete *if the map is total. An instantiation of a schema term (a schema formula, a schema sequent) sc is an instantiation of all the schema variables that occur in sc. An instantiation of a goal template is an instantiation of all the schema variables that occur in the rw- and the add-part. An instantiation of a taclet t is an instantiation of all the schema variables that occur in t without those that occur only in the $addTaclets$ part of a goal template of t.*
ι is canonically continued on $STerm_{SV}$, $SFor_{SV}$, and $SSeq_{SV}$:

$$\iota : STerm_{SV} \cup SFor_{SV} \cup SSeq_{SV} \longrightarrow Term_V \cup For_V \cup Seq_V$$
$$\iota(op(t_1, \ldots, t_n)) = \begin{cases} \iota(op) & \text{if } n = 0 \text{ and } op \in SV \\ op(\iota(t_1), \ldots, \iota(t_n)) & \text{otherwise}^{11} \end{cases}$$

Not all instantiations are meaningful, especially conditions imposed on taclets by their variable conditions should be met. It can be required that the instantiation of a schema variable must be a *skolem* term using a new, so far unused, function symbol. Another condition restricts the free variables occurring in an instantiation. This enables to strictly check that no free variables can intrude into a formula through a taclet application. When defining this property the *variable prefix* already defined in Cond. 3 is reused. For

[11] We assume in this case that substitutions (that are syntactic elements of $SFor_{SV}$) are applied here in a collision resolving way on formulas.

taclets with a sequent as find-part, the variable prefix describes exactly those schema variables (that match on bound variables) whose instantiation may occur in the instantiation of this variable. The check for the variable condition is thus merely a lookup in the prefix. For taclets with a term as find-part, additional free logic variables can be allowed in the instantiations, namely those bound above the position where the taclet is applied.

In all of the following definitions we assume a fixed set of schema variables SV to be given which taclets and their elements are built over.

Definition 14 *The variable conditions VC of a taclet $t = (f, \mathit{ifseq}, VC, GT, H)$ are* satisfied *by the instantiation ι at the occurrence $\underline{\mathit{focus}}$ of a term, formula, or sequent focus, if*

- *for every $sv \in SV$ of type* Formula *or* Term *and all free variables x in $\iota(sv)$:*
  - $x \in \{\iota(\mathit{var}) \mid \mathit{var} \in \Pi(sv)\}$ *(as in Cond. 3), or*
  - *there is only one goal template $(rw, add, addTaclets) \in GT$ with $rw \neq \perp$; further: for this $rw \in STerm_{SV} \cup SFor_{SV}$, sv occurs only in rw and f, and x is bound above $\underline{\mathit{focus}}$;*
- *whenever there is a variable condition $(sv_0$ `new depending on` $sv_1) \in VC$ and $\iota(sv_0), \iota(sv_1)$ are defined then $\iota(sv_0)$ is a new skolem constant.*[12]

4.2. Taclet Application

Instantiations allow us to speak of *applicability* of a taclet. If there are instantiations that unify a concrete occurrence and a find-part of a taclet then the taclet is called applicable at that occurrence. Furthermore we are interested in such instantiations that make taclets applicable and call them *matching*.

Definition 15 *A taclet $t = (f, \mathit{ifseq}, VC, GT, H)$ is* applicable *at an occurrence $\underline{\mathit{focus}}$ of a term, formula, or sequent focus, if there is an instantiation ι of f such that,*

- *if focus is a term or formula and $f \in STerm_{SV} \cup SFor_{SV}$: $\iota(f) = \mathit{focus}$;*
- *if $\underline{\mathit{focus}}$ occurs on top level, $f \in SSeq_{SV}$, f_{for} the single formula in f, further $\underline{\mathit{focus}}$ and f_{for} are either both in the antecedent or both in the succedent: $\iota(f_{\text{for}}) = \mathit{focus}$;*
- *if focus is a sequent and $f = \perp$: ι is empty;*
- *the variable conditions VC of t are satisfied by ι at $\underline{\mathit{focus}}$.*

Such an instantiation ι is then called a matching instantiation *for t and $\underline{\mathit{focus}}$.*

Note, that *occurrences* of terms, formulas, or sequents are considered. This is justified because the find-part of a taclet considers occurrences as well, i.e. when it is a schematic sequent and thus requires an occurrence of a formula in either the antecedent or the succedent.

Example 1 *The taclet (1) in Sect. 2. is applicable at the top level occurrence of the formula $p(c) \rightarrow p(d)$ in the sequent $p(d) \vdash p(c) \rightarrow p(d)$ because*

- *there is an instantiation ι which maps ϕ to $p(c)$ and ψ to $p(d)$,*
- *both occurrences, the one in the find-part and $p(c) \rightarrow p(d)$, are in the succedent, and*

[12] In this article we do not describe how skolemisation is performed. A skolem term usually depends on some meta-variables (i.e. free variables for the purpose of the automated proof search) occurring in another term. In order to pass this other term to the prover the second argument sv_1 is needed. In its simplest form a skolem function (instead of a skolem constant) will be used with the meta-variables of $\iota(sv_1)$ as arguments.

- *no variable condition is violated.*

The following straight-forward definition helps to cleanly define the subsequent notion of a *result of a taclet application*.

Definition 16 *A sequent seq* includes *a sequent seq' if, for all top level formulas ϕ of the antecedent of seq', ϕ is also in the antecedent of seq and, for all top level formulas ψ of the succedent of seq', ψ is also in the succedent of seq.*

Basically, taclets get applied by instantiating the goal templates of a taclet using complete and matching instantiations. Thus, first the effect of instantiating a goal template is defined. There, special care has to be taken of the different kinds of rewrite-parts of a goal template, i.e. whether the focus occurrence is on top level in either the antecedent or the succedent, or if a term is rewritten.

Taclets add those taclets that are defined in their $addTaclets$ set dynamically to the rule base of subsequent goals. Instantiations made to the "mother taclet" should of course have an effect on the "child taclet". The latter are however not instantiated directly, since this would violate their well-formedness. Instead the instantiation of the taclet application they stem from is "memorised" by creating a pair of the taclet to be added and the instantiation restricted to the schema variables occurring in the new taclet. This has the consequence that implementing provers have to manage taclets with partial instantiations instead of taclets only (see Def. 20).

Definition 17 *Suppose seq is a sequent and $\underline{focus}$ is an occurrence of a term, formula, or sequent focus in seq. Suppose further $gt = (rw, add, addTaclets)$ is a goal template, ι is a complete instantiation for gt. The sequent $seq' \in SSeq_{SV}$ and the set T are defined as follows*

- *seq' equals seq except for the fact that*
  - *if $rw \in STerm_{SV} \cup SFor_{SV}$: $\underline{focus}$ is replaced with $\iota(rw)$,*
  - *if $rw \in SSeq_{SV}$: $\underline{focus}$ is removed and seq' includes $\iota(rw)$,*
  - *seq' includes $\iota(add)$.*
- *$T = \{(t, \iota|_{SV_t}) \mid t \in addTaclets\}$. With SV_t, we denoted the schema variables occurring in a taclet t.*

The result of a gt-application *on $\underline{focus}$ and seq with ι is the tuple (seq', T) if seq' is a well-formed sequent (otherwise the result is not defined).*

The if-part of a taclet requires special treatment. Here, two cases have to be distinguished: Either the required if-part is already contained in the sequent, or it is not. In the first case, the taclet simply gets applied. In the latter, it must be proven that the facts required by the if-part are actually fulfilled; this is done by creating an additional goal, the *if-cut-application*, which "negates" the if-part, i.e. adds the (instantiated) antecedent of the if-part as conjunction to the succedent and handles the succedent of the if-part analogously. This can easily be simulated by performing the corresponding cuts manually.

Definition 18 *Suppose $ifseq \in SSeq_{SV}$, ι is a complete instantiation for $ifseq$. The* result of an if-cut-application *of $ifseq = \phi_1, \ldots, \phi_n \vdash \psi_1, \ldots, \psi_m$ on a sequent seq with ι is a sequent seq' such that seq equals seq' with the exception that,*

$$\vdash \bigwedge_{i=1,\ldots,n} \iota(\phi_i) \wedge \bigwedge_{j=1,\ldots,m} \neg\iota(\psi_j)$$

is included in seq'.

The actual taclet *application* is now simple to define: it is just the list of the results of its goal template applications plus (possibly) the if-cut-application. Furthermore, it is possible to add the sequent of the if-part to the result of every goal template application.

Definition 19 *Suppose again seq to be a sequent and $\underline{focus}$ to be an occurrence of a term, formula, or sequent focus in seq. Let $t = (f, \mathit{ifseq}, VC, GT, H)$ be a taclet and (i) ι a matching instantiation for t and $\underline{focus}$. Suppose (ii) ι is complete (for t).*

The result of a t-application on $\underline{focus}$ and seq with ι *is the tuple $((seq_1, T_1), \ldots, (seq_n, T_n))$ with these properties:*

- *if $\iota(\mathit{ifseq})$ is included in seq then $n = \#(GT)$ otherwise: $n = \#(GT) + 1$, $T_n = \emptyset$, and seq_n is the if-cut-application of ifseq on seq with ι.*
- *for all $gt_i \in GT$ ($i = 1, \ldots, \#(GT)$): (seq'_i, T_i) is the result of the gt_i-application on $\underline{focus}$ and seq' with ι. For all $i = 1, \ldots, \#(GT)$, seq_i equals seq'_i but includes $\iota(\mathit{ifseq})$.*

If there is a $gt \in GT$ for which the result of the gt-application on $\underline{focus}$ and seq with ι is not defined then the result of the t-application is not defined either.

Note again, that two basic conditions are required in the above definition of a taclet application: An instantiation must be *(i)* complete and *(ii)* matching, otherwise an application is not defined. A taclet-based system must disallow applications with incomplete or not matching instantiations.

Now, some properties of taclet-based provers can be defined. We specify the system's state to include at least a set of open goals. In reality, it might be preferable to regard a proof tree as the state of the prover, in order to inspect older goals and to undo proof steps manually. Of course, the latter view subsumes the former one because the leaves of the proof tree would be seen as the open goals of the state defined here. With this notion of state, a proof step, i.e. an application of a taclet, can be regarded as a state transition between the states. Simply put, the semantics of a taclet is defined to be a relation between sets of goals.

Definition 20 *A* state *of a taclet-based system consists (at least) of a set of* goals*. Each of the goals contains a sequent and a set of pairs (t, ι), the taclet set, where t is a taclet and ι an instantiation of t.*

If ϵ_t denotes the empty instantiation of a taclet t, and there is a fixed set of base taclets*, i.e. those taclets a proof is started with, then an* initial state *consists (at least) of a goal with a sequent and a taclet set $T = \{(t, \epsilon_t) \mid t \text{ is a base taclet}\}$.*

A proof step *is a pair (s_1, s_2) of states if there are g, $\underline{focus}$, t, ι', and ι with*

1. *g is a goal with the sequent seq and the taclet set T from s_1,*
2. *$\underline{focus}$ is an occurrence of a term (formula, or sequent) focus in seq,*
3. *$(t, \iota') \in T$,*
4. *ι is an instantiation of t with the properties:*
   - *for all schema variables sv of t, $\iota'(sv)$ defined implies $\iota'(sv) = \iota(sv)$,*
   - *ι is complete,*
   - *ι is matching for t and $\underline{focus}$,*
   - *the result of the t-application on $\underline{focus}$ and seq with ι is defined.*
5. *s_2 equals s_1 except that g is replaced with n new goals $g_1, \ldots, g_n$. A goal g_i contains the sequent seq_i and the taclet set T_i such that $((seq_1, T_1), \ldots, (seq_n, T_n))$ is the result of the t-application on $\underline{focus}$ and seq with ι.*

A lot is deliberately left open in this definition, especially how a goal, a focus term or formula, a taclet, and an instantiation is chosen in a proof step. These decisions depend on the actual realisation of the taclet-based prover and, moreover, on the mode a user is working in. We look at the two basic modes, the interactive and the automated mode in more detail in Sect. 5.

Example 2 *Consider the taclet (3) from Sect. 2. Let us call it t_1 and the taclet in its set $addTaclets$ is called t_2. Furthermore, we take a state s_1 that contains a goal g_1. Goal g_1 consists of the taclet set T (with $(t_1, \emptyset) \in T$) and the following sequent seq_1:*

$$c \doteq 0 \ \vdash \ f(c) \doteq f(0)$$

There is a complete instantiation ι_1 of t_1 which maps s to c and t to 0. Taclet t_1 is applicable at the antecedent formula of seq_1 with ι_1. Thus, if there is a state s_2 which equals s_1 except for the fact that g_1 is replaced with a goal g_2 that consists of a taclet set $T \cup \{(t_2, \iota_1)\}$ and of $seq_2 = seq_1$, then (s_1, s_2) is a valid proof step.

Furthermore, (s_2, s_3) is a valid proof step if s_3 equals s_2 except that g_2 is replaced by a new goal g_3 containing the same taclet set as s_2 but the sequent

$$c \doteq 0 \ \vdash \ f(0) \doteq f(0)$$

The reason is, that there is an instantiation $\iota_2 = \iota_1$ which is matching (and already complete) for t and the occurrence of c in the succedent of seq_2.

Some modal logics distinguish between rigid and non-rigid terms. The first are independent of states, i.e. it is irrelevant if they occur after a modal operator or not, while the latter can be evaluated to different values, depending on their occurrence. It is therefore impossible to have an equation handling taclet like (3) from Sect. 2. for non-rigid terms. By distinguishing between schema variables that must be instantiated to rigid terms and others, the taclet can still be used if the involved schema variables match only on rigid terms.

This situation is still unsatisfactory since equations of non-rigid terms could not be handled by taclets. As an extension of the taclet mechanism it is therefore possible to make use of an additional "flag" called *same modality level* that requires taclet application positions to respect the occurrences of modalities before this position. With this possibility a similar taclet like the one above can be established which treats non-rigid terms. The correct handling of rigid and non-rigid terms has been implemented in the KeY system.

4.3. Adaptation for JAVA CARD DL

The definitions above have already handled taclets in a general way. However, the schema variables that are specific to taclets for JAVA CARD DL are placeholders for JAVA CARD DL programs, not for terms or formulas. The definitions of instantiations have thus to be slightly adapted as follows:

Definition 21 *The partial map ι in Def. 13 is extended to:*

$$\iota : SV \longrightarrow Term_V \cup For_V \cup JavaProgElem$$

Analogously for the continuation:

$$\begin{aligned} \iota \quad : \quad & STerm_{SV} \cup SchemaJava_{SV} \cup SFor_{SV} \cup SSeq_{SV} \\ & \longrightarrow Term_V \cup JavaProgElem \cup For_V \cup Seq \end{aligned}$$

The instantiation conditions according to Def. 13 of the extra schema variables for JAVA CARD DL *are as described in Table 1*[13]*. The meaning of the additional variable conditions (Def. 8) is given by*

[13]The terminology there refers to [Gosling et al., 2000].

- *if there is a variable condition* $(t\ \ pv\ \ \texttt{new}) \in VC$ *and* $\iota(pv)$ *is defined then* $\iota(pv)$ *is a program variable that is new and is of the* JAVA *type* t*.*

- *if there is a variable condition* $(\texttt{type}(pv_1)\ \ pv_0\ \ \texttt{new}\ ) \in VC$ *and* $\iota(pv_0), \iota(pv_1)$ *are defined then* $\iota(pv_0)$ *is a program variable that is new and is of the* JAVA *type that* $\iota(pv_1)$ *is of.*

Two further concepts needed for taclets for JAVA CARD DL have been omitted so far: the schematic program context (Def. 11) and meta constructs. Their introduction will show again how flexible the taclet framework is and how easily it is extensible for new tasks. Please note, that the introduction of these concepts is not an ad-hoc solution but can be carried out in the same manner in similar situations.

The schematic program context makes allowance for the fact that JAVA CARD DL rules operate on the first active statement of the program attached to a modal operator [Beckert, 2001]. It is therefore crucial to have a schematic construct that consumes opening braces or `try`-blocks before the active statement and keeps track of the end of the active statement. This function is performed by the schematic program context which has already been provided syntactically in Def. 11. The next definition describes its meaning by a program transformation, called *context instantiation*, that puts the active statement into an appropriate context. The notion of instantiation must then be altered to contain such a specific program transformation. Note, that there is only one such program transformation per taclet instantiation because there is only one schematic program context available as its syntactical representation. This is however an arbitrary restriction as we will explain below.

The extension with *meta constructs* is necessary to increase the expressive power of JAVA CARD DL taclets. Without them it is still possible to model an impressively large, non-trivial part of JAVA CARD, e.g. the complete exception handling or the treatment of null pointer accesses. Nevertheless, some parts of the language—like dynamic method dispatching—cannot be treated in a purely schematic way.

Meta constructs are references to (meta) evaluation procedures that seamlessly transform given program elements into other program elements when a taclet is being applied. With meta constructs, the taclet language gains a very powerful instrument, which, if misused, could destroy its simplicity and elegance. Users are thus not allowed to invent new meta-constructs; a fixed small set of predefined meta constructs is provided instead. To implement the JAVA CARD DL calculus in KeY, surprisingly few were needed. They are mainly used to access type information on JAVA program elements, e.g. the subtype hierarchy of classes. We mention some of the meta constructs here:

- `#method-call`: Transforms its argument, which is a method reference into an if-else-cascade simulating dynamic binding by case distinction on the runtime type of the target object.

- `#typeof`: Delivers the static type of its argument expression.

Meta constructs and the schematic program context form the main additions to refine the definitions of instantiations and applications of taclets for JAVA CARD DL.

Definition 22 *The evaluation procedure* S_{mc} *of a* meta construct mc *with arity* n *is a function that transforms an* n*-tuple of program elements into a single program element.*

A map cti : $JavaProgElem \longrightarrow JavaProgElem$ *is a* (program) context instantiation *if, for every* $\alpha \in JavaProgElem$, $cti(\alpha)$ *is a sequence of statements of which the first one contains* α *and has only opening braces, opening try blocks, and similar "inactive" parts of* JAVA CARD *in front.*

An instantiation of a schema term (schema formula, goal template, taclet) sc for JAVA CARD DL *is a pair* (ι_0, cti) *of an instantiation* ι_0 *of sc in the sense of Def. 13/21 and a context instantiation* cti*. The map* ι *on* $SchemaJava_{SV}$ *is then modified:*

$$\iota(op(\alpha_1, \ldots, \alpha_n)) = \iota_{\iota_0,cti}(op(\alpha_1, \ldots, \alpha_n)) = \begin{cases} \iota_0(op) & \text{if } n = 0 \text{ and } op \in SV \\ cti(\iota(\alpha_1)) & \text{if } n = 1,\ op(\alpha_1) = \ \textbf{..}\,\alpha_1\,\textbf{...} \\ S_{mc}(\iota(\alpha_1), \ldots \iota(\alpha_n)) & \text{if } op = mc \text{ is a meta construct} \\ op(\iota(\alpha_1), \ldots, \iota(\alpha_n)) & \text{otherwise} \end{cases}$$

An instantiation (ι_0, cti) *of a taclet is complete if* ι_0 *is complete. The definitions 15 to 20 are literally the same for* JAVA CARD DL *taclets.*

Example 3 *The taclet (4) from Sect. 2. is applicable at an occurrence of a formula*

$$\langle\ \texttt{try \{ \{ if (true) i=0; else i++; i--;\} \}}\ \texttt{catch (Exception e) \{\}}\ \texttt{while (i>0);}\rangle\ \texttt{i} \doteq 0$$

with a complete and matching instantiation $\iota = (\iota_0, cti)$ *defined as:*

sv	$\iota_0(sv)$
#se	true
#s0	i=0;
#s1	i++;
ϕ	$\texttt{i} \doteq 0$

$$cti(\alpha) \;=\; \texttt{try \{ \{ } \alpha \texttt{ i--;\} \} catch (Exception e) \{\} while (i>0);}$$

In the result of a taclet application the above occurrence is replaced by a formula:

$$\begin{array}{lll} & \texttt{true} \doteq \texttt{true} & \rightarrow \langle\ \texttt{try \{ \{ i=0; i--;\} \} catch (Exception e) \{\}} \\ & & \quad \texttt{while (i>0);}\rangle\ \texttt{i} \doteq 0 \\ \wedge & \texttt{true} \doteq \texttt{false} & \rightarrow \langle\ \texttt{try \{ \{ i++; i--;\} \} catch (Exception e) \{\}} \\ & & \quad \texttt{while (i>0);}\rangle\ \texttt{i} \doteq 0 \end{array}$$

This extension to an advanced notion of *instantiation* could of course be made more general by allowing more than one schematic program context. We would need to keep track of the different contexts by labelling them with names. This indicates that contexts are nothing else than schema variables that depend on a parameter. Adding the schematic program context to taclets is thus a seamless extension of the schema variable concept.

Finally, we give an example for a meta construct.

Example 4 *We consider a taclet that handles a complex* `throw` *statement, like*

`throw new NullPointerException();`.

First, the complex argument of the throw statement has to be evaluated by assigning it to a freshly introduced program variable. We observe that this program variable has to be declared locally, as required in JAVA. *For this declaration the type of the complex expression is needed and, to obtain it, the meta construct* `#typeof` *is utilised. The definition of the taclet is*

```
eval_throw {  find(⟨..throw #nse;...⟩φ)
              varcond(typeof(#nse) #v0 new)
              replacewith(⟨.. #typeof(#nse) #v0=#nse;
                                throw #v0;...⟩φ)}.
```

`#nse` *is a schema variable of type* NonSimpleExpression *which matches expressions possibly having side effects.* `eval_throw` *is applicable at an occurrence of*

$$\langle\ \texttt{throw new NullPointerException();}\rangle\ \mathit{true}.$$

The result of the evaluation of $S_{\texttt{\#typeof}}(\iota(\texttt{\#nse})) = S_{\texttt{\#typeof}}(\texttt{new NullPointerException()})$ *is a* JAVA *type reference to* `NullPointerException`. *In the result of the taclet application, the occurrence of the above formula is thus replaced with*

$$\langle\ \texttt{NullPointerException e = new NullPointerException();}\ \texttt{throw e;}\ \rangle\ \mathit{true}.$$

5. Pragmatics

This section describes how the semantics of taclets affects a taclet-based prover from the user's point of view. Though we abstract away from an actual implementation, the solutions are illustrated at examples of the implementation in the KeY system [Ahrendt et al., 2004, 2002]. A first implementation following similar guidelines as pointed out here was the prover IBiJa [Supp, 1998; Habermalz, 2000a]. Basically, there are two modes in which taclets are applied, the *interactive* and the *automated* mode. There may, however, be variations, e.g. in interactive mode users may opt for minimising their interaction with the prover by only offering taclets which will not require an if-cut, etc.

Let us assume a system that implements the abstract state transition system sketched in Def. 20. Let us further assume that the system is in state s_0. We describe how, after a possible sequence of intermediate states, the next visible state is achieved.

5.1. Interactive Mode

The interactive mode is characterised by the fact that the user is working on a single (*the current*) goal g (i.e. a sequent with a list of partially instantiated taclets) at a time by applying a taclet chosen by himself. Of course the user can choose another goal to become the *current* goal to be worked on. Fig. 1 shows how a user is working on the current goal.

In general, when a proof step is performed, Def. 20 requires us to provide (1) a goal, (2) a focused occurrence of a term, formula, or sequent, (3) a taclet and (4) an appropriate instantiation. Because of the agreement that a user is working only on a single goal at a time, requirement (1) is achieved quite naturally: The *current* goal is the goal for the taclet application.

A way to initiate the application of a taclet in interactive mode is to perform three interaction steps:

1. The user can *select* either the whole sequent or a part of it, such as a formula in the sequent or a subformula or subterm. By such a selection the user chooses a *focus occurrence* of a term (formula, sequent). Hereby prerequisite (2) is determined.
2. It is, after the selection, possible to request a list of taclets that are applicable (Def. 15) at the selected focus occurrence. Prerequisite (4) is now at least partially met. However the found instantiation might not be complete yet.
3. Then, from this list a taclet can be chosen for *application* by the user. By this decision, prerequisite (3) is satisfied.

The realisation of these initial steps in the KeY system has already been demonstrated in Sect. 2.2.

As we have already stated, the instantiation of the chosen taclet is possibly not complete yet. The matching instantiation must thus be completed before applying the taclet. In interactive mode it is most natural to ask the user to give instantiations of the schema variables that are missing in the matching instantiation. There are various possibilities on how such an interaction can be designed.

Again we just give an impression of the realisation in the KeY system. Fig. 4 shows a dialog that pops up when instantiations of some schema variable are missing and must be given by the user. In this example the application of the taclet `all_left` (see (2) in Sect. 2.3.1.) requests the input of an instantiation for the schema variable `t`. The user chooses the ground term $f(c)$ as instantiation. The instantiations (of the schema variables `b` and `u`) that are already given from matching the find-part of the taclet are also displayed but cannot be modified anymore. A status area at the bottom of the dialog gives information on the validity of the user instantiations.

By this user interaction, a complete matching instantiation ι for the sequent of the current goal and the focus occurrence is obtained. According to Def. 20, there is thus a proof step (s_0, s_1), where s_1 is defined as in that definition. The system goes to state s_1 and waits for new user interaction. s_1 contains additional goals (possibly with additional, partially instantiated taclets) of which one becomes the new current goal. Which of the new goals is displayed is of course a question of the concrete realisation.

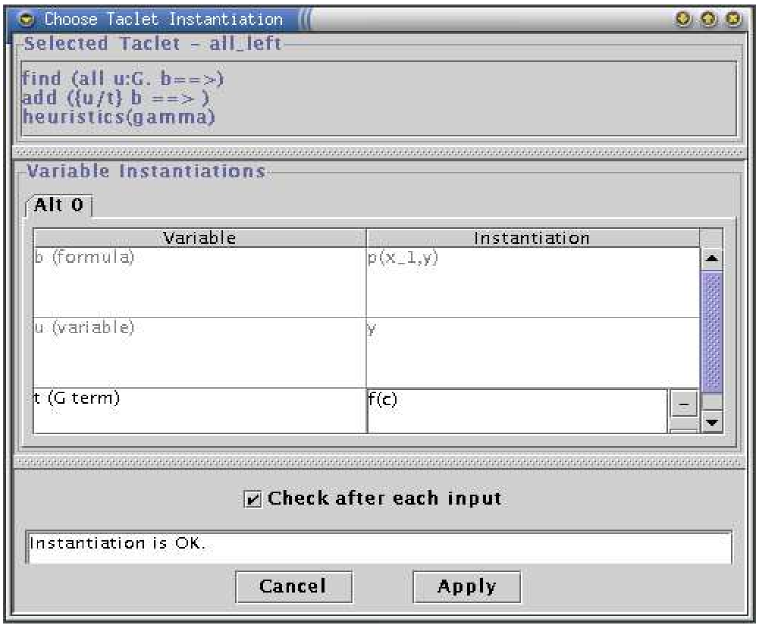

Figure 4. Instantiation Dialog in the KeY System

5.2. Automated Application

Although taclets are specifically designed to be convenient for interactive proving, they are nevertheless well suited for automated proof search. Usually the automated mode can be started right away when being in interactive mode: After one or several interactive steps, a user can initiate the automated mode, or the automated mode is entered after every interactive taclet application.

With their *heuristics-part*, taclets may declare that they belong to a certain set of taclets which can be seen as a collection of rules suited for a certain task. The prover can refer to these collections and apply the taclets belonging to them according to a user selected strategy.

When the automated proof search is initiated, the proof system iterates without further user interaction over all (or only some) goals, including those that have already been produced by automated taclet applications: All taclets of such a goal and all occurrences of all terms and formulas and the whole sequent are searched for matching instantiations. A taclet and a fitting instantiation is chosen by the automated prover and if needed, the instantiation is completed. As in interactive mode, proof steps are performed according to Def. 20 producing several new goals with possibly more (partially instantiated) taclets attached to them.

Which taclets are chosen for application, and how the matching but incomplete instantiations are completed, depends on the sophistication of the implemented automated proof search. In the simplest realisation, only instantiations may be taken that are already complete by instantiating the find-part. When certain criteria are fulfilled, the automated application of taclets stops, and, if the proof is not closed (i.e. there are open goals left), the proof system switches back to interactive mode where the user can apply one or several steps interactively and (possibly) restart the automated proof search.

In the KeY system, the automated proof search can be started by clicking on a button called *Apply Heuristics* at the top of the prover window (see Fig. 1). Then a progress bar at the bottom of the window shows the ratio between the number of already executed taclet applications and a maximal number of applications settable by the user. The automated applications stop when this maximal number is reached or a state is reached where no more rules can be applied automatically or other conditions to stop the search are satisfied (e.g. a sufficient number of quantifier-rules are applied).

6. Correctness

Apologists of logical frameworks sometimes criticize pragmatic theorem proving approaches such as KIV, PVS or KeY, because it is possible to introduce unsound rules. But there are *two different* notions of correctness at work (see Fig. 5).

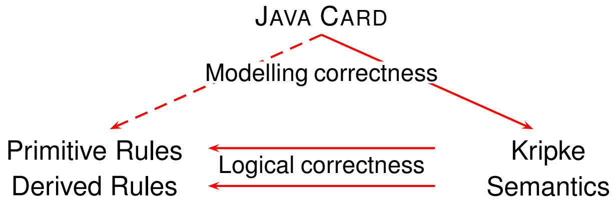

Figure 5. Different notions of correctness

Let us call *modelling correctness* (or *adequacy*) the property that the underlying informal model (programming language constructs, user requirements, etc.) has been faithfully captured in the logical formalism that is used. In the KeY framework, this means that the Kripke model semantics directly reflects the operational semantics stipulated in the JAVA language specification [Gosling et al., 2000].

A different notion is *logical correctness*, that is, formal soundness of the rules of the calculus with respect to the semantics of the target programming language.

Modelling correctness is by its very nature an informal notion. It is straightforward to see in the case of JAVA CARD DL, because the definition of a JAVA CARD DL Kripke model is very close to the informal JAVA specification. What is more, even the *primitive rules* of the JAVA CARD DL calculus directly reflect this semantics (indicated by the dashed arrow). This is due to the transparency of dynamic logic with respect to the target language (programs are first-class citizens) and to the proof paradigm of symbolic execution. In fact, JAVA CARD DL taclets are executable, so one can evaluate them, for example, with JAVA compiler compliance tests.

To ensure modelling correctness in systems where programs and their semantics are encoded as higher-order logic formulas is an altogether much hairier issue, because it is far from obvious whether the operational semantics has been correctly modelled in higher-order logic.

On the other hand, logical correctness is for free in foundational theorem proving approaches. In the taclet/JAVA CARD DL setting, this is the more difficult part: a formal correctness proof of the JAVA CARD DL primitive rules can (if desired) be obtained by formalising and verifying them e.g. in Isabelle, which contains higher-order theories of syntax and semantics of a JAVA fragment. This takes care of logical correctness of those JAVA CARD DL taclets that stay within the limitations of Isabelle's JAVA fragment, and it has been done for some of the central rules.

As to *derivable rules*, if the formula matched by the find-sequent/term and the if-sequent is first-order, as well as the formulas/terms the taclet consists of, then it is possible to schematically generate proof obligations that ensure correctness of taclets. These proof obligations are then also first-order formulas. Similar proof obligations can be generated in the DL case, but need an extension of JAVA CARD DL by elements that can serve as skolem symbols for schema variables. This leads to the notion of an *anonymous program*. In the following subsections, we describe the construction of proof obligations for first-order taclets, and outline how to extend the definition to JAVA CARD DL. For a more detailed account see [Bubel et al., 2004].

Alternatively, one could consider the correctness of certain instances of taclets and tactics instead of general correctness. In a foundational system one could justify concrete instances on-the-fly (whether that were feasible in practice is another question, though). A remedy to this kind of correctness problem in KeY is that JAVA CARD DL is expressive enough to prove relative correctness of programs. This technique allows to prove correctness of taclets that are derivable. Arguably, most lemmas and simplification rules are of this kind.

6.1. Logical Correctness for First-order Logic

In this section, we concentrate on the case of a typed first-order logic (without modal operators) and the basic taclet version of Sect. 3.1. We describe a way to reason about the logical correctness of such "first-order" taclets. For that purpose, an effective construction of a *meaning formula*, which is essentially a first-order formula, of a taclet is defined to achieve a simpler but equivalent representation of its logical content.[14] The meaning formula of a taclet is valid if (and only if) all possible applications of the taclet are correct. To show that a taclet is correct—or to derive correct lemma taclets from existing rules—it is thus sufficient to prove the validity of the corresponding meaning formulas.

Meaning formulas in general contain schema variables, i.e. are schematic formulas. The second step of the presented construction thus transforms meaning formulas of taclets into real first-order formulas. This is achieved by replacing schema variables with skolem terms or formulas.

6.1.1. Meaning Formulas

In the whole section we write $(\Gamma \vdash \Delta)^* := \bigwedge \Gamma \rightarrow \bigvee \Delta$ for the disjunction of the formulas of a sequent, which is in particular

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

Furthermore, in this section by the validity of a sequent we denote the validity of the disjunction $(\Gamma \vdash \Delta)^*$. We recall some well-known definitions, and continue Sect. 2.1. on the concept of *soundness*:

Definition 23 (Soundness) *A (sequent) calculus C is* sound*, if only valid sequents are derivable in C.*

This general definition does not refer to particular rules of a calculus C, but treats C as an abstract mechanism distinguishing a (recursively enumerable) set of derivable sequents. Using Def. 4, the usual sufficient condition can be obtained that associates the soundness of a calculus C with local properties of its rules $R \in C$. Following Def. 3, thereby a rule R is considered to be a relation between tuples of sequents (the premisses) and single sequents (the conclusions):[15]

Lemma 1 *A calculus C is sound, if for each rule $R \in C$ and all tuples $(\langle P_1, \ldots, P_k\rangle, Q) \in R$ the following implication holds:*

$$\textit{if } P_1, \ldots, P_k \textit{ are valid, then } Q \textit{ is valid.} \tag{5}$$

If condition (5) 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*.

In our case, the rules R_t of a calculus C are defined through taclets $t = (f, \mathit{ifseq}, VC, GT, H)$ over a set SV of schema variables, and within the next paragraphs we discuss how Lem. 1 can be applied considering such a rule $R_t \in C$. For the time being, we ignore the addrules-parts of t, i.e. in all goal templates $(rw, add, tac) \in GT$ the set tac of taclets is required to be empty. Furthermore, we assume that t is a taclet that is applied to top level formulas, i.e. $f \in SSeq_{SV}$. If we use the notation

$$\big(\Gamma_1 \vdash \Delta_1\big) \cup \big(\Gamma_2 \vdash \Delta_2\big) \quad := \quad \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$$

for the union of two sequents (we assume that duplicate formulas are implicitly removed), then by Def. 20 t represents the rule schema

$$\frac{rw_1 \cup add_1 \cup \mathit{ifseq} \qquad rw_2 \cup add_2 \cup \mathit{ifseq} \qquad \ldots \qquad rw_k \cup add_k \cup \mathit{ifseq}}{f \cup \mathit{ifseq}}$$

[14] This method has originally been described in [Habermalz, 2000a].

[15] The converse of the following lemma does in general only hold for complete calculi C.

where $GT = \{(rw_i, add_i, \emptyset) \mid i = 1, \ldots, k\}$ is the set of goal templates of t. To apply Lemma 1, it is then necessary to show implication (5) for the sequents

$$P_i = \iota(rw_i \cup add_i \cup \mathit{ifseq}) \quad (i = 1, \ldots, k), \qquad Q = \iota(f \cup \mathit{ifseq}). \tag{6}$$

and each instantiation ι of the schema variables SV. Provided that t does not introduce skolem functions, i.e. does not contain a variable condition "`new depending on`", implication (5)—which poses a *global* derivation regarding the interpretation of all symbols—can be replaced with a stronger *local* implication. This argument also refers to the deduction theorem:[16]

$$\left(P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^*\right) \text{ is valid.} \tag{7}$$

Inserting the sequents (6) extracted from taclet t into (7) leads to a formula whose validity is sufficient for implication (5):

$$P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^* = \bigwedge_{i=1}^{k} \iota(rw_i \cup add_i \cup \mathit{ifseq})^* \rightarrow \iota(f \cup \mathit{ifseq})^*. \tag{8}$$

We can now employ that ι treats propositional junctors as a homomorphism (Def. 13), and that the operator $(\cdot)^*$ is a morphism regarding the union of sequents up to propositional transformations:

$$(P \cup Q)^* \equiv P^* \vee Q^*.$$

Thus, formula (8) is equivalent to[17]

$$\iota\Big(\bigwedge_{i=1}^{k}(rw_i \cup add_i \cup \mathit{ifseq})^* \rightarrow (f \cup \mathit{ifseq})^*\Big)$$

$$\text{and to} \quad \iota\Big(\bigwedge_{i=1}^{k}(rw_i^* \vee add_i^*) \rightarrow (f^* \vee \mathit{ifseq}^*)\Big). \tag{9}$$

If (9) is proved for all instantiations ι, then the rule R_t represented by t will be sound.

The next definition contains the complete formulation of meaning formulas, which is based on (9) (without the instantiation ι), but additionally treats:

- The variable condition "`new depending on`", which is responsible for the creation of skolem functions, and for which existential quantifiers are added. Namely, if in implication (5) the sequents $P_1, \ldots, P_k$ contain skolem symbols not occurring in Q, then these symbols can be regarded as universally quantified. Because the quantifiers are negated in (7) (on the left side of an implication), the whole meaning formula is existentially quantified.

- The addrules-parts of a taclet, which make an inductive definition of the meaning formula necessary. A taclet created by an addrules-statement upon application of another taclet can be seen as a family of formulas (namely all instances of its meaning formula) that is added to the antecedent of a sequent. Hence, the meaning formulas $M(s)$ of inner taclets s essentially occur as negated formulas of the add-parts.

[16] Actually, the new condition is not significantly stronger than (5) because of side formulas that can always be part of a sequent to which a rule is applied. Such formulas are not modified by the application.

[17] Recall that ifseq occurs in the sequent on which t is applied.

- Taclets for which the find-part f and the replacewith-parts are not sequents but formulas or terms (like the inner taclet in (3)), and that rewrite subformulas or subterms. Such a taclet is reduced to a rule adding an equivalence or equation to the antecedent, which leads to a formula slightly different from (9), like (if f is a term):

$$\iota\Big(\bigwedge_{i=1}^{k}(f \doteq rw_i \to add_i^*) \to \mathit{ifseq}^*\Big).$$

  Note that in this formula both f and rw_i are terms, to which the operator $(\cdot)^*$ is therefore not applied, and that the equation $f \doteq rw_i$ is negated by the implication $\to$ (as it is a formula of the antecedent).

For a shorter notation we define $\bot^* := \mathit{false}$ ($f = \bot$ is possible for certain taclets). Furthermore, we assume that goal templates of taclets always have a non-trivial replacewith-part (i.e. not $\bot$; this can always be achieved by copying the find-part), except for taclets not having a find-part either. And finally, if a taclet together with the taclets contained in addrules-parts is regarded as a tree, we require that common schema variables of two taclets t_1, t_2 in that tree also occur within a common ancestor t' (in the tree) outside of an addrules-part, and are therefore already instantiated when applying t' (see below for an example why this assumption is important).

Definition 24 (Meaning Formula) *Inductively, we define a map M of* meaning formulas*. For this, we first continue the operator $(\cdot)^*$ to taclets:*

Let $t = (f, \mathit{ifseq}, VC, GT, H)$ be a taclet over the set SV of schema variables. The unquantified meaning formula t^* is defined by

$$t^* := \bigwedge_{\substack{(rw,add,tac)\\ \in GT}} \Big(\bigwedge_{\substack{s\in\\ tac}} M(s) \to (rw^* \vee add^*)\Big) \to (f^* \vee \mathit{ifseq}^*)$$

if $f \in SSeq_{SV} \cup \{\bot\}$, and by

$$t^* := \bigwedge_{\substack{(rw,add,tac)\\ \in GT}} \Big(\bigwedge_{\substack{s\in\\ tac}} M(s) \to \big(f \doteq rw \to add^*\big)\Big) \to \mathit{ifseq}^*$$

if $f \in STerm_{SV}$ or $f \in SFor_{SV}$ (for the latter case, $\doteq$ is replaced with $\leftrightarrow$).

Let $sv_1, \ldots, sv_k \in SV$ be all schema variables such that a condition

$$(sv_i \ \texttt{new depending on} \ \ldots) \in VC$$

exists. The (quantified) meaning formula $M(t)$ of t is given by

$$M(t) := \exists x_1 \ldots \exists x_k . \phi$$

where ϕ is obtained from t^* by replacing each variable sv_i with a new schema variable x_i of type Variable *that has the same sort as sv_i.*

Example 5 *The taclet t_1 defined by (1) in Sect. 2***.** *represents the rule schema*

$$\frac{\phi \ \vdash \ \psi}{\vdash \ \phi \to \psi}$$

and the meaning formula is the tautology

$$M(t_1) = (\underbrace{\neg\phi \vee \psi}_{=rw^*}) \to (\underbrace{\phi \to \psi}_{=f^*}) \equiv \neg(\phi \to \psi) \vee (\phi \to \psi).$$

As a more interesting example, we consider the taclet t_2 defined by (3). We denote the taclet of the addrules-part by t_3 and obtain the two meaning formulas

$$M(t_3) = (s \doteq t), \qquad M(t_2) = M(t_3) \lor \neg(s \doteq t) = (s \doteq t) \lor \neg(s \doteq t).$$

Obviously, $M(t_3)$ is not a valid formula for most instantiations of the variables s and t, which reflects the observation from Sect. 2. that the taclet t_3 is not correct in general. As $M(t_2)$ is a tautology, however, t_3 is correct in situations in which t_2 can be applied, which distinguishes admissible instantiations of s and t.

A taclet that violates the assumption from above regarding common schema variables of inner taclets is the incorrect taclet

$$t\ \{\ \texttt{addrules (} t_1\ \{\ \texttt{add (} \vdash \phi\texttt{)}\ \}\ \texttt{); addrules (} t_2\ \{\ \texttt{add (} \vdash \neg\phi\texttt{)}\ \}\ \texttt{)}\ \}.$$

One might (wrongly) think that t implements the cut-rule, but in fact t can be used to add arbitrary formulas to a sequent. The "meaning formula" is the tautology $M(t) \equiv \phi \lor \neg\phi$, however, which does not reflect that the two occurrences of ϕ can be instantiated independently when applying t_1 and t_2. An adequate meaning formula, namely $M(t) \equiv \phi \lor \neg\phi'$, can be constructed by replacing one occurrence of ϕ with a new schema variable ϕ' (this does not alter the semantics of t).

6.1.2. Proof Obligations

Except for trivial taclets, the meaning formula $M(t)$ of a taclet t contains schema variables that are placeholders for formulas or terms, which is inconvenient for proving $M(t)$.[18] Variables of these types do not occur bound within the formula, however (when considering validity, they can be regarded as implicitly universally quantified), and hence the validity of a meaning formula is not altered by replacing the schema variables with suitable skolem terms and formulas.

Definition 25 (Proof Obligation) *Let $t = (f, \mathit{ifseq}, VC, GT, H)$ be a taclet and $M(t)$ the meaning formula of t. Let SV denote the set of schema variables that $M(t)$ contains. We define a complete instantiation ι of the variables SV:*

- *If $x \in SV$ is of type* Variable*, then $\iota(x) \in V$ is a new (logical) variable of the same sort as x*
- *If $sv \in SV$ is of type* Term*, then $\iota(sv) = f_{Sk}(v_1, \ldots, v_l) \in Term_V$ is a term, where*
  - *$v_1, \ldots, v_l \in V$ with $v_i = \iota(x_i)$ are the instantiations of $x_1, \ldots, x_l$, where the distinct schema variables $x_1, \ldots, x_l \in SV$ of type* Variable *are determined by the prefix of sv in t:*

    $$\Pi(sv) = \{x_1, \ldots, x_l\}$$

  - *f_{Sk} is a new function symbol with the signature $S_1 \times \ldots \times S_l \to S$*
  - *$S_1, \ldots, S_l$ are the sorts of $v_1, \ldots, v_l$*
  - *S is the sort of sv.*
- *Analogously, if $sv \in SV$ is a schema variable of type* Formula*, then $\iota(sv) = p_{Sk}(v_1, \ldots, v_l) \in For_V$ is a formula containing a new predicate symbol p_{Sk}.*

The proof obligation *for the taclet t is the formula $\phi_t^{po} := \iota(M(t))$.*[19]

[18] Besides that, the meaning formula does not contain information about all variable conditions of a taclet. Hence, it is necessary to treat a meaning formula in the context of the original taclet.

[19] To treat a calculus with meta-variables (see footnote 12), it is necessary to conduct supplementary checks regarding the variable conditions "`new depending on`", which are omitted in this article.

A proof obligation that ensures the soundness of a taclet does only make sense in the context of an application mechanism for taclets. Thus, Def. 24 and 25 rely on the definition of allowed taclet applications, which is given in Sect. 4., e.g. the restriction that the only logical variables that can occur freely within instantiations of a schema variable are the instantiations of elements of the prefix set.[20] To verify (an implementation of) an application mechanism for taclets, as well as the definition of proof obligations, it is thus necessary to show that both are compatible.

Another important aspect of this compatibility is that schema variables of type Variable are instantiated with distinct logical variables in the proof obligation, which is not necessarily true when applying a taclet. This can either be resolved by modifying the proof obligation also to cover instantiations with non-distinct logical variables, or by a taclet application mechanism that recognises and avoids collisions between those variables (which is what has been implemented in the KeY system).

The compatibility of taclet application and proof obligations is formalised in the following lemma, which can also be regarded as the definition of correct taclet applications:

Lemma 2 *If the proof obligation ϕ_t^{po} of a taclet t is valid, then implication (5) of Lemma 1 holds for all premisses/conclusion pairs $(\langle P_1, \ldots, P_k \rangle, Q) \in R_t$ of the rule R_t represented by t.*

This lemma implies, in particular, that taclets whose proof obligation has been proved correct using a sound calculus (e.g. a calculus that is defined through a set of sound taclets) represent sound rules.

Example 6 *For the taclet*

$$t \ \{ \ \texttt{find} \ (\ \vdash \forall x.\psi) \ \texttt{varcond} \ (c \ \texttt{new depending on} \ \psi) \ \texttt{replacewith} \ (\ \vdash \psi_x^c) \ \}$$

which represents a skolemisation-rule for universally quantified formulas of the succedent, the meaning formula is (v is a new schema variable of type Variable*)*

$$M(t) = \exists v. \big(\psi_x^v \rightarrow \forall x.\psi\big).$$

The prefix $\Pi(\psi)$ of the formula schema variable ψ is $\{x\}$ (x is bound by the quantifier and the substitution above both occurrences of ψ), and hence the following proof obligation is derived:

$$\phi_t^{po} = \exists y. \big(\psi_{Sk}(x)_x^y \rightarrow \forall x.\psi_{Sk}(x)\big) = \exists y. \big(\psi_{Sk}(y) \rightarrow \forall x.\psi_{Sk}(x)\big).$$

6.2. Logical Correctness for JAVA CARD DL

As for first-order logic, it is possible to define proof obligations of taclets that contain elements of the dynamic logic JAVA CARD DL,[21] or which can be applied to sequents of this kind, without moving to higher-order logic. Among others, problems that arise when leaving the first-order case are:

- JAVA CARD DL does not contain uninterpreted dynamic constructs like atomic programs (see [Harel et al., 2000]). As skolem symbols for schema variables representing program elements are needed, it is necessary to introduce such symbols by extending the logic.
- Skolem symbols that are introduced for term and formula schema variables are required to be non-rigid, i.e. their interpretation may depend on the state and may be affected by modal operators, because such a behaviour is possible for instantiations of the schema variables. This disqualifies ordinary functions and predicates as in Def. 25.

[20] Except for variables bound above $\underline{focus}$, which in certain situations are also allowed to occur freely; see Def. 14.

[21] Other first-order modal logics can be treated similarly. While in any case the definition of the meaning formula of a taclet can be retained almost without modifications, the main issue when considering other logics is to find (or to define) suitable skolem symbols for new kinds of schema variables.

- For schema variables of type ProgramVariable, which represent local program variables or class attributes, the possibility of non-distinct instantiations has to be covered. Namely, when applying a taclet, two such schema variables can either be instantiated with the same program variable or with distinct ones, which in general leads to completely different formulas. To show that a taclet is sound, both cases have to be considered. For logical variables, similar collisions are prohibited by the taclet application mechanism of the KeY system (hence it is possible to use distinct logical variables to instantiate all schema variables of type Variable in Def. 25).
- It is possible that a taclet is applied within the scope of modal operators, whereas the if-sequent matches top level formulas of the sequent, or equivalently the taclet has an add-part creating top level formulas. In this case, the states in which different parts of a single taclet are interpreted can differ.[22] To prevent taclets that are invalid in such situations, it is necessary to take modalities of the application context into account when defining proof obligations.

A more detailed description and a possible definition of proof obligations is given in [Bubel et al., 2004].

Example 7 *For the* JAVA CARD DL *taclet*

$$t_1 \ \{ \ \texttt{find} \ (\langle \#v = \#v; \rangle \psi) \ \texttt{replacewith} \ (\psi) \ \}$$

in which $\#v$ is a schema variable of type ProgramVariable *and ψ a formula schema variable, the derived proof obligation is*

$$\phi^{po}_{t_1} = \Big(\langle x_{Sk} = x_{Sk}; \rangle \psi_{Sk}(x_{Sk}) \Big) \leftrightarrow \psi_{Sk}(x_{Sk}).$$

In this formula, x_{Sk} denotes a new program variable, and ψ_{Sk} is a fresh non-rigid skolem symbol for formulas. Symbols of this kind do not exist in JAVA CARD DL*, but are in [Bubel et al., 2004] introduced specifically for taclet proof obligations; they are comparable to non-rigid predicate symbols.*

7. Implementation Issues

Now that the theoretical background of the taclet mechanism has been discussed, it is natural to ask how all this can be implemented in an efficient manner. In this section, we review some implementation issues and explain how they were resolved in the KeY prover.

The KeY prover is implemented in the JAVA programming language [Gosling et al., 2000], using the Swing GUI library [Walrath and Campione, 1999]. The coordination between the displayed proof tree, the current sequent, etc., and the underlying logical data structures follows the *Model, View, Controller* architecture, making intensive use of the *Observer* design pattern [Gamma et al., 1995]. Every change in the data structures representing the proof tree triggers an event for which the concerned user interface components wait. While this is not the fastest conceivable technique, it has helped to ensure a good modularisation of the system.

7.1. Highlighting

To assist the user in selecting the focus formula or term, the KeY prover highlights the whole subformula or term the mouse pointer is over as it moves over the sequent (see also Sect. 2.2.). For instance, in the formula

$$p \wedge (q \vee r \wedge s) \quad ,$$

given that $\wedge$ has priority over $\vee$, the right conjunct $(q \vee r \wedge s)$ is going to be highlighted when the pointer is over the $\vee$ or one of the parentheses, $r \wedge s$ will be highlighted when the pointer is over the right $\wedge$, and

[22] This problem does also motivate the introduction of further taclet statements to restrict applications, or further kinds of schema variables syntactically distinguishing rigid and non-rigid formulas.

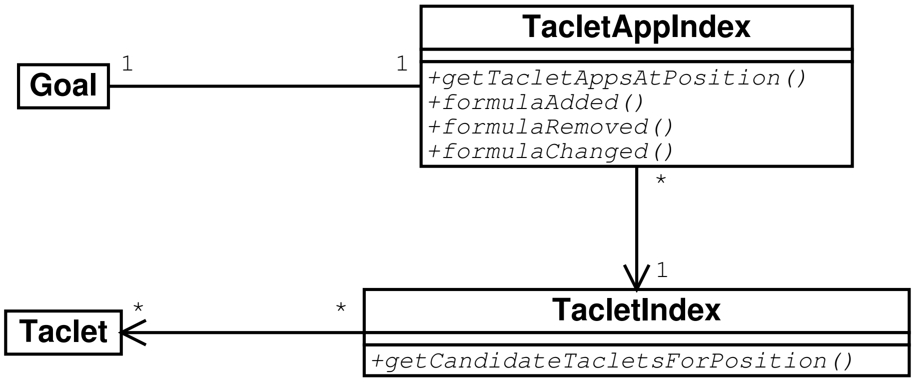

Figure 6. Indexing Data Structures for Taclets

the whole formula if it is over the left $\wedge$. If the pointer is over one of the symbols p, q, r, s, only that symbol is highlighted. The implementation of this feature relies on a fast mechanism to find the term position corresponding to a certain character in the displayed sequent. This is achieved using *position tables*, which record the start and end of nested formulas and terms in every subformula/term of the sequent. Position tables are built by the pretty-printer during layout, at a low additional cost, and they are very efficient. There is no perceivable delay due to highlighting when the mouse is moved over the sequent. The position tables have the same tree structure as the represented terms, so the time to find the position corresponding to a character is linear in the depth of the term. This has so far proved to be fast enough. An additional feature of the position tables is that they store only *offsets* of subterms for each position, instead of absolute positions in the string representation of the sequent. This makes it possible to reuse the position table for a formula that is not affected by a taclet application, provided its layout does not change. This optimisation has not yet been implemented in the KeY system though. Once the text range to be highlighted has been calculated using the position tables, the actual painting is done using the standard highlighting functionality provided by the JAVA libraries.

7.2. The Taclet Application Index

For a pleasant user experience, it is also important that the available taclets at a certain position are displayed with minimal delay when the user clicks somewhere. The first ingredient for this is of course the position table, which yields a handle on the logic data structures corresponding to the mouse position. The actual list of applicable taclets is computed from this using a number of indexing data structures, see Fig. 6. Considering that the taclet set for JAVA CARD DL comprises hundreds of taclets, it is clearly not an option to iterate through the whole set of taclets while the user waits for the menu. Instead, for every open goal, a *taclet application index* is kept, that stores all taclet applications possible in a sequent at any position. In this context, a *taclet application* consists of a taclet along with a position where it is applied and a number of schema variable bindings determined by the position.[23] The taclet application index is organised in such a way that quick access to the applicable taclets is possible based on the position in the sequent. Only taclet applications that are actually possible are stored. Regard for instance the taclet

$$\texttt{find (} \vdash \phi \rightarrow \psi\texttt{) replacewith(}\phi \vdash \psi\texttt{).}$$

from Sect. 2., which has to be applied on an implication in the succedent. Only for such positions is a taclet application going to be put in the taclet application index, and only then will it be displayed to the user. The

[23]There may be unbound schema variables left; the instantiations of those are asked for interactively. These taclet applications are entities in the implementation which do not completely coincide with what was called a taclet application in previous sections.

nice thing about the taclet application index is that most of a sequent usually remains unchanged between taclet applications, and accordingly most of the taclet applications remain valid. It is sufficient to remove taclet applications referring to changed formulas and to add some for new formulas after each proof step. Again however, this is an optimisation which is not strictly necessary. Until recently, the taclet application index was simply recalculated before each interaction.

7.3. The Taclet Index

To update the taclet application index, one needs to pick the taclets that are applicable at a certain position in a formula from among a set of maybe several hundred taclets. KeY uses another indexing data structure to do this efficiently: The *taclet index*. This contains the set of all available taclets, and provides an operation to determine a set of candidates that might be applicable, given some formula and its position in a sequent. The idea is to go through all subformulas of a newly introduced formula in a sequent and ask the taclet index for a (hopefully small) set of potentially applicable taclets. For each taclet in this set, it is then checked whether all conditions for the application are actually satisfied, and if so, a corresponding taclet application is put into the taclet application index.

What indexing mechanism is sensible for the taclet index is of course dependent on the set of taclets in use. For instance, many of the taclets currently used in the KeY prover serve the symbolic execution of programs. Therefore, we make sure that the indexing can differentiate between taclets for various kinds of JAVA statements. We use a hash table indexed by the top operator of the formula or term in question, and in case of program modalities, by the type of the first executable statement in the program in question. This gives very acceptable performance for interactive use: the time required to apply a rule, to build the new taclet application index and to layout and display the new sequent lies mostly below half a second. The standard set of taclets usually worked with comprises several hundred taclets for propositional and predicate logic, integers, sets and above all for JAVA CARD. When taclets are applied automatically using the heuristics, performance ranges between 20 rule applications per second for the more complicated symbolic execution taclets to about 500 per second for simple propositional logic on a current Linux workstation.

The performance of the taclet index might become unacceptable in the future, due for instance to an enlarged taclet base. In that case, our course will be to progressively optimise the indexing data structures. In fact, this has already been done twice in the past: originally there was no taclet index at all. As the number of predicate logic rules grew, hashing on the top function symbol was introduced. Finally, with the addition of DL rules, indexing on program statements became necessary.

Another conceivable future optimisation is to compile taclets: As taclets have a quite operational semantics, it would be possible to produce JAVA byte code for the actions of a taclet, instead of the current interpretative approach. In particular the matching part might become faster than with the current approach of comparing two term data structures. It is not clear whether this will become necessary, as the system performs quite satisfactorily so far.

8. Case Studies

Taclets are not merely a theoretical concept but were successfully used to implement the theorem prover of the KeY system. Here we list a number of major case studies that involved writing and using taclets:

- The interactive theorem prover and simplifier of the KeY tool [Ahrendt et al., 2004, 2002] are implemented on the basis of taclets. The target language of the prover is the full JAVA CARD language. This shows that taclets are powerful enough to describe the whole JAVA CARD semantics in a comparatively concise way. This JAVA CARD semantics is even executable. Therefore, taclets are powerful enough to write a JAVA CARD interpreter. Taclets that deal with JAVA CARD programs contain special constructs for symbolic state updates. Application and simplification of updates was not realised with taclets (but could have been) to boost system performance. All other aspects of the interactive theorem prover were done with taclets.

- The main aspect where JAVA CARD goes beyond JAVA is its transaction mechanism. Handling transactions requires extension of the JAVA CARD DL with a "throughout" modality [Beckert and Mostowski, 2003]. To this end the taclet mechanism is extended with schematic modal operators, a kind of placeholder for modal operators. This allows to generalise most common rules for a set of modalities. Realisation of support for the throughout modality now boils down to writing a number of rules that differ from the diamond/box rules mainly in how variable assignment is handled. The remaining rules are taken from the diamond rules using schematic modal operators. Merely one aspect of JAVA CARD transactions had to be hard coded into the prover. The transactions extension was implemented in a few person months. In a recent case study involving the verification of a JAVA CARD electronic purse the KeY prover equipped with suitable heuristics found proofs with thousands of taclet applications automatically within one minute [Hähnle and Mostowski, to appear, 2004].

- KeY was applied to analysis of secure information flow [Darvas et al., 2003]. Traditionally this is done by static analyses based on specialised type systems. Although efficient, such approaches need to approximate complex language constructs such as loops, reference types, or exceptions. Verification is not fully automatic, but yields higher precision. Taclets allow to combine both approaches, as they make it easy to add incomplete rules as used in type-based systems. It was a matter of hours to add such rules and emulate results from type-based analysis.

- There is an instance of the KeY system realising an axiomatisation of Abstract State Machines (ASM) instead of JAVA CARD as its target language for verification. It is based on the ASM logic developed in [Stärk and Nanchen, 2001] and covers parallel and recursive ASMs. This shows that taclets are general enough to support a completely different target language than JAVA CARD.

9. Related Work

Taclets were introduced under the name of *schematic theory specific rules (STSR)* by Habermalz [Habermalz, 2000a,b]. The concept of interactive theorem proving by pointing the mouse at the formula a rule should act upon was inspired by the prover InterACT [Geisler et al., 1996]. That theorem prover had a more or less hard-wired set of rules, however. Domain-specific reasoning was only possible by application of conditional equations taken from an algebraic specification. With taclets domain-specific reasoning is possible in a way that matches human reasoning in the domain, not the underlying specification language.

An idea for using mouse gestures to control a theorem prover, known as "Proof by Pointing" has already been suggested earlier by Bertot, Kahn and Théry [Bertot et al., 1994]. The peculiarity of the proof by pointing approach is that a single mouse click on some subformula can trigger a whole series of rule applications that decompose a formula until the selected subformula is on the top level of the sequent. Proof by pointing is limited to a fixed sequent calculus, with no domain-specific rules at all.

Semantically, taclets bear an obvious resemblance to tactics and/or derived rules in systems based on higher-order logic like Isabelle [Paulson, 1994] or PVS [Owre et al., 1996], but also to concepts from the proof planning world like the methods of the ΩMEGA system [Siekmann et al., 2002]. Indeed, in a taclet-based theorem prover, taclets often play the role of (possibly derived) rules or tactics, and they do encode knowledge about domain-specific reasoning like methods. Taclets differ from the cited concepts in that they (i) include an operational semantics for both automated and interactive application; (ii) do *not* provide any programming constructs, and thus (iii) can be justified with respect to other taclets by reasoning in the object logic, and not in some higher-order 'meta' logic.

10. Conclusion & Future Work

We presented the idea of taclets, which are a new means for constructing interactive theorem provers. They are the technology of choice for implementing all those calculi (including domain-specific simplifiers) that require user interaction and have a large number of rules—for such calculi they are, in our experience, superior to the more general approach of logical frameworks.

As only a restricted class of logics (first-order modal logic) is considered, a large part of the proof construction techniques can be implemented efficiently and once and for all as part of the taclet system; and new features can be quickly implemented. Taclets provide a clear separation of (a) logical content, (b) context of usage, and (c) heuristics for automated/interactive application.

Moreover, taclets are a compact and clear notation. They are easy to use—even for persons with limited experience in logic. Theory and practice of the taclet language can be learned quickly—the basic concepts in less than a day. This helps with the quick development of interactive provers and makes taclets a good choice for educational purposes.

For the future, we plan to implement more calculi using our taclet mechanism, including static program analyses (for JAVA) and program logic calculi for other programming languages such as C.

Acknowledgement. We would like to thank Richard Bubel, Vladimir Klebanov, Wojciech Mostowski, and Peter H. Schmitt for careful reading of the manuscript and important feedback.

References

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print, DOI: 10.1007/s10270-004-0058-x.

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Proc. Fundamental Approaches to Software Engineering. 5th International Conference, FASE 2002, Grenoble, France*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

Bernhard Beckert and Wojciech Mostowski. A program logic for handling Java Card's transaction mechanism. In Mauro Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference*, volume 2621 of *LNCS*, pages 246–260, Warsaw, Poland, April 2003. Springer.

Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In M. Hagiya and J. C. Mitchell, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Software*, LNCS 789, pages 141–160. Springer, 1994.

Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring correctness of lightweight tactics for Java Card Dynamic Logic. In *Proceedings of Workshop on Logical Frameworks and Meta-Languages (LFM) at Second International Joint Conference on Automated Reasoning 2004*, pages 84–105, 2004. URL `http://i12www.ira.uka.de/~aroth/publications/TacletCorrectness.pdf`.

Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Roberto Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

Dov M. Gabbay. *Labelled Deductive Systems*, volume 1—Foundations. Oxford University Press, 1996.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

R. Geisler, M. Klar, and F. Cornelius. *InterACT*: An interactive theorem prover for algebraic specifications. In *Proc. AMAST'96, 5th International Conference on Algebraic Methodology and Software Technology*, LNCS 1101, pages 563–566. Springer, July 1996.

Martin Giese. Taclets and the KeY prover. In Christoph Lüth and David Aspinall, editors, *Intl. Workshop on User Interfaces for Theorem Provers, UITP 2003, Rome, Italy*, ENTCS. Elsevier, 2004. To appear.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.

Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln*. PhD thesis, Universität Karlsruhe, 2000a.

Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000b. URL `http://www.key-project.org/doc/2000/stsr.ps.gz`.

Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, LNCS. Springer-Verlag, to appear, 2004.

David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *LNCS*, pages 411–414. Springer, July/August 1996.

Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994.

Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. In *Proceedings of the CADE-15 Workshop on Integration of Deduction Systems*, pages 51–60, July 1998. Available as Technical Report 441, Computer Laboratory, University of Cambridge.

J. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.P. Wirth, and J. Zimmer. Proof development with ΩMEGA. In A. Voronkov, editor, *Proceedings of the 18th Conference on Automated Deduction (CADE–18)*, volume 2392 of *LNAI*, pages 144 –149, Copenhagen, Denmark, 2002. Springer Verlag, Germany.

Robert F. Stärk and Stanislas Nanchen. A logic for abstract state machines. *Journal of Universal Computer Science*, 7(11):981–1006, 2001.

M. Supp. Implementierung eines integriert interaktiven und reflexiven Beweisers für Prädikatenlogik. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 1998.

Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison Wesley, 1999.

Martin Giese, Reiner Hähnle and Philipp Rümmer
Chalmers University of Technology & Göteborg University
Department of Computing Science
S-41296 Göteborg, Sweden
giese|reiner|philipp@cs.chalmers.se

Andreas Roth and Steffen Schlager
University of Karlsruhe, Department of Computer Science
D-76128 Karlsruhe, Germany
aroth|schlager@ira.uka.de

Bernhard Beckert
University of Koblenz-Landau, Institute for Computer Science
D-56070 Koblenz, Germany
beckert@uni-koblenz.de

Elmar Habermalz
sd&m AG, software design & management
Herrnstraße 57
D-63065 Offenbach, Germany
elmar.habermalz@sdm.de