Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe

Studienarbeit

Ensuring the Soundness of Taclets
–
Construction of Proof Obligations for JavaCardDL Taclets

Philipp Rümmer

20. Dezember 2003

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Richard Bubel, Andreas Roth

Danksagung

An dieser Stelle möchte ich meinen Betreuern Richard Bubel und Andreas Roth danken, die sich mit bewundernswerter Ausdauer mit meinen verschiedenen Ansätzen zu dieser Studienarbeit auseinandergesetzt haben, und deren Kommentare und Vorschläge für mich unersetzlich waren. Danken möchte ich ebenfalls Bernhard Beckert, dessen Hinweise Ausgangspunkt verschiedener Konzepte dieser Arbeit waren, sowie Professor Peter Schmitt, der die Erstellung dieser Studienarbeit ermöglicht hat. Für viele Hinweise zu den Raffinessen der englischen Sprache, und hilfreichen Anmerkungen zum Aufbau dieses Dokuments möchte ich André Platzer danken. Schliesslich sei meinen Eltern für die fortlaufende finanzielle und moralische Unterstützung gedankt.

Erklärung

Ich erkläre hiermit, diese Arbeit selbständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 20. Dezember 2003

Philipp Rümmer

Contents

1. Introduction

The KeY project [ABB$^+$03] of the University of Karlsruhe and the Chalmers University of Technology, Gothenburg has the goal to develop tools that enable formal specification and verification of software, with the particular intention to make these tools as easy to use as possible. A central part of this project is an (mostly) interactive prover component, which is based on a sequent calculus for modal logics, and in particular for a dynamic logic treating the programming language JavaCard [Sun02]. The calculus rules of this prover are formulated as so-called *taclets*, which is a concept that has first been described in [Hab00]. Taclets pose an intuitive and easy way to describe rules of sequent calculi schematically, and at the same time they are especially suited to realize rules that are to be applied interactively.

In [Hab00], taclets are introduced together with a method to construct for a given taclet a *proof obligation*, which is a first-order formula that in a certain way represents the meaning of a taclet. These formulas are used to reduce the problem of *soundness* of the rule represented by a taclet to the problem of *validity* (of the proof obligation), and pose a convenient way to maintain large sets of available calculus rules, still ensuring the soundness of the calculus.

At the present time, within the KeY system no automated generation of proof obligations for taclets is implemented. This has mainly two reasons:

- The definition of taclets and proof obligations in [Hab00] is only performed for first-order logic, and thus not (immediately) applicable to the situation in KeY, where it is necessary to treat dynamic logic

- Taclets in the KeY system are implemented introducing a concept called *schema variables*, which are variables that have a purely syntactic meaning, and that can only occur within taclets (and not as part of a proof). In [Hab00], for this purpose (first-order) object variables are used.

In this thesis, we will describe a construction analogous to the creation of proof obligations for taclets in [Hab00], but suited for the realisation of taclets in KeY.

1.1. Proving the Soundness of Taclets

On a semantic level, by the soundness of a rule (r) for a given calculus **K** we mean the impossibility to prove formulas not valid, by including applications of the rule in proofs. To show that a given rule is sound, it is thus necessary to consider each possible application of the rule that can emerge within a proof attempt of a formula that is *not* valid, and ensure that the application of the rule cannot lead to a successful continuation of the attempt.

Immediately referring to this definition to show the soundness of a rule is rather inconvenient, as it contains at least two complex components:

- The definition directly refers to applications of the rule, which makes it necessary to treat the semantics of a given (formal) description of the rule to furnish a proof of its soundness

- It is necessary to cover all possible applications of the rule, i.e. one has to perform some kind of induction over the structure of the circumstances, like the situation of the proof, within which the rule can be applied.

It is therefore desirable to find a more convenient characterisation of the soundness of rules. Because the soundness problem for taclets is usually undecidable (except for taclets that treat propositional logics), it seems natural to formulate these *proof obligations*, which guarantee the soundness of rules t, as a formula φ_{po}, such that:

$$\text{the formula } \varphi_{\mathrm{po}} \text{ is valid} \quad \Longleftrightarrow \quad t \text{ is sound} \tag{1}$$

To make this construction useful, thereby the creation of formulas φ_{po} for given taclets t has to be computable. In any case, the formula φ_{po} already represents the semantics of the rule defined through a taclet t, therefore this transformation of the soundness problem eliminates one of the two complicated parts we have mentioned. If it additionally is possible to formulate φ_{po} as a formula within a "convenient" logic, at best within the logic the considered calculus $\mathbf{K}$ treats itself, then also the second item is handled.

A given construction procedure of proof obligations is only valuable, if it is possible to show that equivalence (1) really holds for this procedure. The more important direction of relation (1) is the one from left to right, and the aspect we will concentrate on most of the time in this document. Knowing that this implication holds already guarantees a calculus to stay sound, as long as only rules with valid proof obligations are added.

The other direction, being responsible for prohibiting proof obligations stronger than necessary, thus "forbidding" even sound taclets, has a more practical meaning. There is a trade-off between the simplicity of (building and proving) the proof obligation, and the "extravagance" of rules that ought to be proved sound (provided that the rules *are* sound). This issue, as well as aspects of completeness will not be treated in more detail in this thesis.

Proving equivalence (1) has, however, a striking disadvantage: Namely, it needs to refer to semantics, defined for the considered logic (to formulate a usable proof obligation it may also be necessary to enrich this logic by further elements). As we are most interested in the logic JavaCardDL, this semantics would at least subsume the JavaCard language, and thus be rather complex, and also inflexible for considering subsets and enhancements.

In this thesis, we will therefore pursue a concept slightly different. Instead of treating the soundness problem using a semantic approach, we will concentrate on the formulation of *lemma rules*, which are rules whose effect can be reproduced referring to already existing rules. For a sound and complete calculus $\mathbf{K}$, both notions are obviously equivalent (and for a calculus that is sound, the rules that can be introduced are in particular sound, too). Likewise, instead of formulating proof obligations whose *validity* is of interest, we are satisfied with the term *derivability*. Equivalence (1) then becomes

$$\varphi_{\mathrm{po}} \text{ is derivable in } \mathbf{K} \quad \Longleftrightarrow \quad t \text{ is reproducible in } \mathbf{K} \tag{2}$$

It should be stressed, however, that for both ways of proving the soundness of the taclet t the actual proof obligations φ_{po} are not necessarily different.

$$
\begin{tikzcd}
 & \text{Taclet } t \arrow[dl] \arrow[dr] & \\
\text{Application } a_t \text{ of } t \arrow[d, dashed, leftrightarrow] & & \text{Proof obligation } \varphi_{\mathrm{po}} \arrow[d] \\
\text{Proof } H(a_t) & & \text{Proof } H(\varphi_{\mathrm{po}}) \arrow[ll]
\end{tikzcd}
$$

Figure 1: Overview of our method

The second, purely syntactic approach has (beside others) the following advantages for us:

- While not being completely independent from the underlying semantics (in some situations we will need to assume the existence of certain rules, expressing specific properties of the semantics we rely on), this is indeed the case for certain aspects. For example, the method to construct proof obligations that is described in Sect. 4.4, which refers to first-order logic, can as well be applied to richer logics (as dynamic logic), provided that the expressiveness of rules is not also enhanced (i.e. with this restricted procedure we can't reason about rules modifying specific constructs of dynamic logic, not present within first-order logic, like programs)

- We will get a very "modular" proof of the implication from left to right in equation (2), that can easily and stepwise be extended upon introduction of new features of the logic or the considered rules.

1.2. The Important Steps

For a taclet t whose soundness is to be proved, and referring to a calculus $\mathbf{K}$ that is regarded as correct, we want to provide a construction mechanism for a formula φ_{po} that satisfies the direction from left to right of equivalence (2) from the previous section. In Sect. 3, a uniform representation of taclets as a set of formulas by using schema variables is defined. The derivation of the proof obligation from the rule (in this representation) essentially consists of the replacement of schema variables with suitable skolem symbols.

To show that the implication (of equivalence (2)) holds, we follow the diagram of figure 1:

1. We assume that a proof $H(\varphi_{\mathrm{po}})$ of the proof obligation φ_{po} exists, using the calculus $\mathbf{K}$ (or a calculus enriched by a number of special rules, needed to deal with the introduced skolem symbols); this is the "derivability"-premise of the implication to be shown

2. We consider an arbitrary application a_t of the rule t; as the rule is described using schema variables, this means that we are given a tuple of replacements (instantiations) for the schema variables (and the position of application, for which we also define an appropriate schema variable type)

3. The proof $H(\varphi_{\text{po}})$ of the proof obligation is transformed into a "proof" $H(a_t)$ of the rule application a_t, which means that from $H(\varphi_{\text{po}})$ we derive a sequence of rule applications in the calculus $\mathbf{K}$ (i.e. only using rules that are known to be sound, and especially *not* using t), having the same effect as the rule application a_t. The most important part of this proof transformation is the replacement of the skolem symbols introduced in φ_{po} by the schema variable instantiations of a_t; for that, we will have to define a number of substitution operators (see appendix A).

1.3. How to read the Thesis

- In **Section 2**, some minor assumptions and definitions about the used logic JavaCardDL are given; the section does however *not* contain a complete introduction to JavaCardDL

- **Section 3** consists of an introduction to taclets, restricted to the features treated in this thesis, and a definition of the operational semantics of taclets through meaning formulas; furthermore, schema variables are defined, distinguishing variables sufficient for first-order logic (FOL), and supplementary variables necessary for JavaCardDL. Section 3 is the only section dealing with taclets, later on only meaning formulas of taclets turn up, which pose a more general representation of rules

- **Section 4** is the central part of the document, in which first some basic tools are supplied, and later the derivation of proof obligations from meaning formulas is defined and verified. Like Section 3, Section 4 contains two parts, in which proof obligations are introduced for first-order logic and for JavaCardDL respectively

- **Section 5** sketches some possibilities to extend the methods of Section 4 to cover further kinds of rules and schema variables

- In **Appendix A**, two kinds of substitutions (again corresponding to the treatment of first-order logic and JavaCardDL respectively) are defined and discussed, which are mainly used in Section 4.

2. JavaCardDL

2.1. Introduction to JavaCardDL and Basic Notations

JavaCardDL is an instance of first-order dynamic logic for the programming language JavaCard, which is essentially a subset of Java (see [Bec01, HKT00, GJSB00, Sun02]). As central part of the KeY project [ABB+02, ABB+03], a prover has been implemented based on a sequent calculus for JavaCardDL, that is used to reason about JavaCard programs annotated with OCL constraints (these constraints are first translated to JavaCardDL, see [OMG01, Kel02]).

We only give a very short enumeration of properties of JavaCard and JavaCardDL in this section, which are either not self-evident or unusual. Beside that, we refer to the more extensive descriptions of JavaCardDL in [Bec01, ABB+03].

First we introduce an additional statement, which is added to JavaCard as part of JavaCardDL to handle method calls:

- A statement **method−frame** is added, which is mainly responsible for catching **return**-jumps, and which can optionally be equipped with a program variable to store the result value of a **return**-call. The necessary rule for the Java grammar (referring to [GJSB00], Sect. 14.5) is:

  StatementWithoutTrailingSubstatement:
  method-frame ($Identifier_{opt}$) *Block*

  If a statement **return** is executed, the innermost **method−frame**-statement will complete abruptly. If this **method−frame** is furnished with a result variable v, then additionally the result of **return** is assigned to v (this implies the requirement that a result exists and is assignable to v). The target of a **break**- or **continue**-statement that is executed within a **method−frame** must not lie outside of the **method−frame**.[1]

- Eventually we will use the notation **skip** for the neutral statement, which is preferred to a single semicolon for aesthetic reasons.

Within the whole thesis, we will (implicitly) refer to the following basic properties of JavaCardDL:

- JavaCardDL subsumes first-order logic, i.e. provides the usual first-order quantifiers as well as function and predicate symbols (which are regarded as rigid when considering dynamic aspects). A vocabulary for JavaCardDL (at least) consists of the following sets:
  - A set *Sort* of sorts
  - A set *Func* of function symbols
  - A set *Pred* of predicate symbols
  - A set *PVar* of program variables.

[1] This is a requirement that should most probably be enforced by static analysis.

Furthermore we assume that the following two sets pose sufficient reservoirs of symbols:

- The set *Var* of logical variables
- The set *Label* of Java labels.

The sets of JavaCardDL formulas and terms are denoted with *For* and *Term* respectively. We also use the notation *Syn* for the set of all formulas, terms and programs of JavaCardDL, as these elements can be treated alike in many situations.

- Beside the set *Sort* of sorts of a vocabulary, there is also a set *JavaTypes*, consisting of all types the JavaCard language provides. Each Java type $T \in \mathit{JavaTypes}$ is embedded in a sort $S_T \in \mathit{Sort}$, and whenever a program element of type T (e.g. a program variable) occurs as a term outside of a program, the sort of this term is S_T. A detailed description of this topic can be found in [Sch02, Bec01].

- There are two kinds of *variables*:
  - Logical variables, which are rigid and are assigned a *sort*[2]
  - Program variables, i.e. the variables the Java programming language provides, which are non-rigid and are assigned a *Java type.* A small selection of aspects of program variables that is important for this thesis is discussed in Sect. 2.3.

- JavaCardDL provides three families of *modal operators*, namely
  - the diamond operator $\langle\alpha\rangle\varphi$, which is coloured with a JavaCard program α, and essentially has the usual meaning as described in [HKT00]
  - the box operator $[\ \alpha\ ]\varphi$, which has the semantics

  $$[\ \alpha\ ]\varphi \equiv \neg\langle\alpha\rangle\neg\varphi$$

  - the (simultaneous) update operator, as described in [ABB+03]

  $$\{\mathrm{v}_1 \mathrel{:=} t_1, \ldots, \mathrm{v}_l \mathrel{:=} t_l\}\varphi.$$

  Further kinds of modalities, like the trace-modalities introduced in [BS01], are not considered in this thesis.

- JavaCardDL contains an *object-level substitution operator*[3]

  $$\{x\ \ t\}T$$

  where x is a logical variable, t is a term having the same sort as x and T either is a term or a formula. Occurrences of x in T are in this case regarded as *bound* by the substitution.

[2] We won't use the common notation "object variables" to avoid confusion resulting from similar names of Java classes.

[3] In the KeY system, this operator is only defined for formulas and terms used to define calculus rules.

To conclude this section, we define some terms and notations that are used throughout the document:

- We call two formulas, terms or programs φ_1, φ_2 of JavaCardDL *equal modulo bound renaming*, written as

$$\varphi_1 =_{\mathrm{br}} \varphi_2,$$

  if φ_1 and φ_2 can be identified using collision free bound renaming of

  - Logical variables, which are bound by quantifiers and the object-level substitution operator
  - Program variables, which are bound by local variable declarations within Java programs (see Sect. 2.3 on this topic)
  - Program labels, which are bound by the labelling of a statement within a Java program. The scope of a label l is the statement that is labelled with l.

- To distinguish object-level substitution operators (as introduced above) and *meta-level substitution*, for the latter one we write

$$\{x_1/t_1, \dots, x_l/t_l\}\varphi, \qquad \sigma = \{x_1/t_1, \dots, x_l/t_l\}, \qquad \sigma(\varphi)$$

  where $x_1, \dots, x_l$ are logical variables, and $t_1, \dots, t_l$ are terms having the same sorts as $x_1, \dots, x_l$ respectively. The application of σ to terms and formulas is defined as usual, see for example [Fit96]. We assume, however, that no implicit resolution of collisions is performed.

  Substitutions for other kinds of symbols than logical variables are defined in appendix A.

2.2. Sorts

As used in the KeY project and introduced above, JavaCardDL is a sorted logic. While the reasoning in this document does not depend on the exact definitions by which a term is assigned its sort, or on the provided family *Sort* of sorts, we formulate the following presumptions that will be used implicitly throughout the following sections:

- Replacing a sub-term s of a term or a formula T by a term s', which has the same sort as s, leads to a syntactically correct term or formula T'. If T is a term, then T' has the same sort as T
- Analogously, replacing a subexpression of a JavaCard expression α with an expression of the same Java type shall not alter the type of α
- Terms occurring as arguments of function and predicate symbols s within syntactically correct terms or formulas shall exactly have the sorts demanded by the signature of s.[4]

[4] For terms of a real sub-sort, we assume that some kind of cast is implicitly inserted; actually this method is also used in the KeY-system implicitly.

2.3. Program Variables

By the term "program variable" (PV) we denote local/stack program variables and class attributes, in particular not instance attributes or array elements of Java.[5] Hence program variables are nullary, non-rigid symbols, which are assigned an arbitrary Java type. Program variables need to be distinguished from *logical variables,* which are rigid and are assigned a sort rather than a Java type. Like the occurrences of logical variables,[6] occurrences of program variables can either be bound, which means declared locally within a statement block (especially within a program block, belonging to a modality), or free (both inside program blocks and within terms).

It is important that scopes of program variables differ from scopes of logical variables (see [GJSB00], Sect. 14.4.2), which leads to interesting kinds of collisions (example 2.4). While the latter are always sub-formulas/subtrees of a formula below a quantifier or a similar operator binding variables, the scope of a program variable covers all statements that follow the declaration of the variable within the innermost statement block (in special cases, e.g. parameter declarations of catch blocks, all statement of the following block):

```
{ j = 0; int i; i = 0; }
```

Contrary to the current situation in the KeY-system, we define the scopes of program variables in particular to be limited by program blocks; this means that in

$$\langle \textbf{int}\ \mathrm{i} = 1; \rangle \mathrm{i} \doteq 0$$

the two occurrences of i denote (conceptually) different variables. Renaming the first i to j would not alter the meaning of the formula.

2.1 *Definition (Statements respecting Scopes):* We say that a statement or a list of statements α *respects scopes*, if program variables declared in α cannot be observed outside α (i.e. scopes of variables declared in α are completely contained by α). $*$

2.2 *Example:* The statement α

```
int i;
```

does not respect scopes, as the scope of i is not limited to α. On the other hand, the statement β

```
{ int i; }
```

obviously respects scopes. $*$

2.3 *Lemma (Bound Renaming):* Let β be a Java program fragment, and α be a statement of β that respects scopes. If α' is obtained from α by renaming bound program variables or labels, and β' from β by replacing an occurrence of α with α', then β and β' are equal modulo renaming. $*$

[5] These variables are logically treated as non-rigid mappings of the sort representing a Java class or the array index type. It is however not necessary to spend much considerations on these mappings in the scope of this document, mostly because occurrences cannot be bound.

[6] Within calculi, we will usually forbid the occurrence of free logical variables within top-level formulas however.

2.4 *Example:* To demonstrate that Lem. 2.3 does not hold if α does not respect scopes, we consider the program

```
int i;
i = 0;
```

Replacing the statement **int** i; with **int** j; leads to a program *not* equal modulo renaming. Similar problems arise if a statement is replaced with another statement that does not respect scopes. *

3. Taclets

3.1. Introduction

"Taclets" are rules for sequent calculi, formulated in a simple and intuitive syntax, and originally developed by Elmar Habermalz as "Schematic Theory Specific Rules" (STSR) in [Hab00]. Taclets are in particular appropriate for defining rules to be applied interactively, but are not restricted to that purpose. Though at first taclets were introduced for a first-order logic, in the KeY system taclets have been adapted to the logic JavaCardDL. A description of taclets that applies to all (first-order) modal logics is [BGH$^+$03].

In this section, we will mostly give an introduction to the taclet language and its semantics, specific for JavaCardDL. Beside that, we refer to the more complete descriptions in [Hab00, BGH$^+$03]. The other parts of this document are indeed mostly independent from taclets, as we introduce a more general and simpler representation of calculus rules, which (on a quite theoretic level) subsumes taclets.

For our purposes, a taclet t is given by an expression like

$$
\begin{array}{l}
[\,\texttt{if}(\Gamma_{\texttt{if}} \;\vdash\; \Delta_{\texttt{if}})\,] \quad [\,\texttt{find}(A)\,] \quad [\,\texttt{sameUpdateLevel}\,] \\
\qquad [\,\texttt{varcond}(c_1, \ldots, c_l)\,] \\
\qquad [\,\texttt{replacewith}(B_1)\,] \quad [\,\texttt{add}(\Gamma_{\texttt{add},1} \;\vdash\; \Delta_{\texttt{add},1})\,] \quad [\,\texttt{addprogvar}(\mathbf{V}_1)\,]; \\
\qquad\qquad\qquad \vdots \\
\qquad [\,\texttt{replacewith}(B_k)\,] \quad [\,\texttt{add}(\Gamma_{\texttt{add},k} \;\vdash\; \Delta_{\texttt{add},k})\,] \quad [\,\texttt{addprogvar}(\mathbf{V}_k)\,]
\end{array}
$$

in which

- the first line determines the situations and positions in which the taclet can be applied; namely to sequents containing the formulas of the sequent $\Gamma_{\texttt{if}} \;\vdash\; \Delta_{\texttt{if}}$, and to formulas and terms A within these sequents

- the second line contains conditions to be fulfilled by instantiations of *schema variables* contained in formulas and terms within the taclet (these are described below)

- the k lines below are the "goal templates", describing the effect an application of the taclet has; namely k new goals (sequents) are created, in which A (at the position of application) is replaced by B_i, and to which the formulas of $\Gamma_{\texttt{add},i} \;\vdash\; \Delta_{\texttt{add},i}$ are added respectively (the statement `addprogvar` is also described below).

A can either be a simple formula or term (and then each B_i has to be a formula or a term, too), or a sequent containing exactly one formula (then each B_i has to be a sequent); these two possibilities distinguish between taclets being applicable to arbitrary sub-formulas or sub-terms (rewrite taclets), and taclets that can be applied only to top-level formulas, either of the antecedent or the succedent of a sequent (lemma taclets).

3.1 Example: A very simple rewrite taclet t_1 is

$$\texttt{find}(1+1) \quad \texttt{replacewith}(2)$$

and an application could be

$$\frac{\vdash \ \langle \mathrm{i} = 2; \rangle 2 \doteq i}{\vdash \ \langle \mathrm{i} = 2; \rangle 1+1 \doteq i} \ t$$

*

In this example, the taclet t was applied to a term below a modality; while applications of this kind are correct for t, there is a number of other rules, in particular taclets that handle equations, for which applications below modalities must not be allowed. For this reason the keyword `sameUpdateLevel` can be added to the definition of a taclet.[7]

Example 3.1 reveals that taclets as introduced so far are not very flexible, as they merely perform replacements and additions of constant formulas and terms, and the component still missing are variables that can be used in formulas and terms of the taclet description. These variables are defined independently of taclets, and pose the most important concept of this document:

3.2. Schema Variables

Schema variables, as they are defined in the following sections, were first introduced in the KeY project as part of the implementation of a taclet application mechanism. A formal description that mostly reflects the current implementation within the KeY system can be found in [BGH+03].

To describe the subject of this document in a most abstract way, one could say that we will deduce features of certain subsets of the set *For*, namely of sets that are defined by formulas containing placeholders, to be replaced by different formulas, terms or other syntactical constructs. Such sets/formulas arise as abstract descriptions of taclets, in a way such that each possible application of a taclet is represented by one element of the set, and such that an application is sound if (and only if) this element is valid. For example, the formula

$$\forall \#x.\#p$$

denotes the set of all formulas, obtained by replacing the symbol $\#x$ by a variable y of the logic, and $\#p$ by an arbitrary formula $\varphi \in \mathit{For}$ not containing free variables except y. The used placeholders are called *schema variables*.

3.2 Definition (Schema Variable): Schema variables are symbols $\mathbf{S} = \{\#s_i \mid i \in \mathbf{I}\}$, not occurring in the vocabulary of the underlying logic, together with a map

$$p : \mathbf{S} \to \mathbf{T} \times \mathbf{P}$$

mapping each schema variable $\#s_i$ onto a tuple $p(\#s_i) = (T, P)$

[7] In the KeY system, this keyword furthermore enforces that updates occurring above the position of application also occur in front of the formulas $\Gamma_{\texttt{if}} \vdash \Delta_{\texttt{if}}$, and are prepended to formulas $\Gamma_{\texttt{add},k} \vdash \Delta_{\texttt{add},k}$. The same exceptional behaviour is allowed for lemma rules. These special cases are not discussed in this document.

- T being the type of the schema variable (several types are defined later in this section, e.g. for variables specifically representing terms or formulas); the kind of the schema variable also determines its arity, which in most cases is however zero. This type should not be mistaken for the sort that can be assigned to some variables
- P being a tuple of attributes of the schema variable (which will be called *properties* in this section); possible values of P depend on the type T of the schema variable, e.g. for a term schema variable a property could determine the sort of terms represented. $*$

Schema variables may occur within terms or formulas (or other syntactic constructs a logic defines) at virtually any position. Therefore we do not distinguish between different kinds of term-like structures (like formulas, terms, programs, etc.) in the next definition, and rather denote the set of all such constructs within JavaCardDL by *Syn*. We regard the elements of *Syn* as trees, in which nodes are coloured with symbols or words (like elements of the vocabulary, logical variables or distinguished syntactical elements of the logic like $\forall x.$). From *Syn*, we derive a larger set $\mathit{Syn}_{\mathrm{SV}}$ of constructs by enriching the logic with schema variables.

3.3 Definition (Schema Variables in Formulas): Let $\mathbf{S}$ be a set of schema variables as in Def. 3.2. The set $\mathit{Syn}_{\mathrm{SV}} = \mathit{Syn}_{\mathrm{SV}}(\mathbf{S})$ of *schematic syntactical elements* is then defined inductively to be the smallest set with

- For $t \in \mathit{Syn}$, the tree t' obtained by replacing arbitrary subtrees by elements of $\mathit{Syn}_{\mathrm{SV}}$ is an element of $\mathit{Syn}_{\mathrm{SV}}$; especially $\mathit{Syn} \subset \mathit{Syn}_{\mathrm{SV}}$
- For $\#s \in \mathbf{S}$ a schema variable of arity k, and $t_1, \ldots, t_k \in \mathit{Syn}_{\mathrm{SV}}$:

$$\#s(t_1, \ldots, t_k) \in \mathit{Syn}_{\mathrm{SV}}. \qquad *$$

The only schema variables that are not nullary we are going to introduce are variables for contexts, both within formulas and programs.

Given a duplicate-free tuple $S = (\#s_1, \ldots, \#s_n)$ of schema variables, a set $\mathbf{R} \subset \mathit{Syn}^n$ of possible *instantiations* of the schema variables is distinguished.[8] For that, the definition of each schema variable type T contains conditions discriminating valid and invalid instantiations (i.e. a recursive predicate over Syn^n). The set $\mathbf{R}$ then contains exactly those instantiations that are valid for each schema variable $\#s_i$.

3.4 Example (Validity of Schema Variables Instantiations): In the example

$$\forall \#x.\#p$$

from above, the instantiation $\iota = (\alpha_1, \alpha_2)$ of the schema variables $S = (\#x, \#p)$ could be defined to be valid if (and only if) the following conditions are fulfilled:

[8] In most cases these instantiations simply pose replacements for the schema variables. Depending on the type of the schema variable, in particular for variables not nullary, there can however be more complex instantiation maps.

- $\#x$: α_1 is a (logical) variable
- $\#p$: α_2 is a formula, and $FV(\alpha_2) \subset \{\alpha_1\}$.

In this example, the instantiations would be used to replace the schema variables. $*$

3.5 Remark: For instantiations of some schema variable types, it will be necessary to use a slightly larger base set $Syn_{\#s} \supset Syn$, that differs from Syn by additional special symbols $a_{\#s}$. These symbols only occur within instantiations of the schema variable $\#s$ and not under "normal" circumstances, simply to make reasoning easier. For example, a schema variable $\#ct$ of type "context" will be instantiated with formulas that may contain a distinguished propositional atom $a_{\#ct}$, marking a position within the formula, like

$$\forall x.(p(x) \vee a_{\#ct}).$$

Upon instantiation (i.e. $\#ct$ is in a certain way replaced with such a formula) the atom $a_{\#ct}$ is also replaced, and can thus be regarded as a formal parameter. $*$

Instantiation of Schema Variables

Given an instantiation tuple $\iota = (\alpha_1, \dots, \alpha_k)$ for the tuple $S = (\#s_1, \dots, \#s_n)$ of schema variables as introduced above, we define a map

$$\lambda_\iota : Syn_{\mathrm{SV}} \to Syn$$

replacing schema variables within terms, formulas, etc. with their instantiations ι. In general, this map will be determined by

$$\lambda_\iota(op(r_1, \dots, r_l)) = \begin{cases} J_T(\alpha_m, \lambda_\iota(r_1), \dots, \lambda_\iota(r_l)) & \text{if } op = \#s_m \text{ is a SV of type } T \\ op(\lambda_\iota(r_1), \dots, \lambda_\iota(r_l)) & \text{otherwise} \end{cases} \tag{3}$$

where for each type T the map J_T defines the exact procedure of replacement for schema variables of type T (these maps are defined below). λ_ι can roughly be seen as a substitution, replacing schema variables with their instantiations.

In fact, most schema variables are nullary symbols (i.e. $l = 0$, this depends on the type T of the variable), thus for most variables $op = \#s_m$, equation (3) can be reduced to

$$\lambda_\iota(op) = \lambda_\iota(\#s_m) = J_T(\alpha_m) = \text{const}$$

and additionally for most nullary types we have $J_T = id$, thus

$$\lambda_\iota(\#s_m) = \alpha_m.$$

To make instantiations of schema variables more treatable, we introduce the following two conventions:

- As for substitutions, we will use the notation

  $$\iota = \{\#s_1/\alpha_1, \dots, \#s_k/\alpha_k\}$$

  instead of writing ι as the tuple $\iota = (\alpha_1, \dots, \alpha_k)$, implicitly referring to a given tuple $S = (\#s_1, \dots, \#s_n)$ of schema variables

- We will identify the instantiation tuple ι and the corresponding instantiation map λ_ι, and write
$$\iota(a) := \lambda_\iota(a).$$

Within the next sections, different kinds of schema variables will be introduced, based on Def. 3.2. Most of the following variables are (also) used to describe and define taclets, some however occur only implicitly (within the taclet mechanism of KeY, and within [Hab00]), e.g. the context schema variable, representing a position within a formula, which are defined explicitly within this document to achieve a more uniform representation.

We will always assume that a set $\mathbf{S}$ of schema variables is given, together with a single formula $\varphi \in \mathit{Syn}_{\mathrm{SV}}(\mathbf{S})$ in which the elements of $\mathbf{S}$ may occur (we do not require that *all* variables of $\mathbf{S}$ actually turn up in φ). For this situation, we describe the possible types and properties the variables may have, the resulting ranges w.r.t. instantiation and the instantiation maps J_T.

3.2.1. First-Order Schema Variables

Within this section, we ignore elements of JavaCardDL that exceed first-order logic, which are in particular modal operators. For first-order logic, there are only four types of schema variables, representing logical variables, terms, formulas and contexts. To avoid problems that arise with the possibility of collisions, in the next definition we make rather strict postulations regarding logical variables and the corresponding schema variables, that could also be replaced with weaker (and more complicated) conditions.

<table>
<tr><th colspan="2">Variable Schema Variables (VariableSV)[9]</th></tr>
<tr><td>sort[10]</td><td>Instantiations allowed for this schema variable must have exactly this sort.</td></tr>
<tr><td colspan="2">A VariableSV is to be instantiated with logical variables. No two VariableSV within S may be instantiated with the same variable[11], and the instantiation of a VariableSV must not occur bound within the instantiation of any other schema variable within S.</td></tr>
<tr><td colspan="2">$J_{\mathrm{VariableSV}}(\alpha) = \alpha$[12]</td></tr>
</table>

The variable $\#x$ in example 3.4 is supposed to be a VariableSV. To avoid collisions, we do not want the instantiation of a VariableSV to occur in φ (i.e. except within other schema variable instantiations); in Rem. 3.8, however, we will usually forbid the occurrence of explicit logical variables (i.e. variables that are not schema variables) within φ at all, so we skip further restrictions of VariableSV at this point.

[9] This is the name of the schema variable type defined in this box.

[10] This is a property as in Def. 3.2.

[11] Within the KeY project, no restriction that strong is made for the instantiations of VariableSV, it is however possible to reduce the more tolerant definition to this one by renaming bound variables, see [Gie02].

[12] This is the instantiation map of this schema variable type.

3.6 Definition (Prefix of a Schema Variable): By the *prefix* **P** of a schema variable $\#s$ we denote a property of $\#s$ being a set, determining objects that may occur (usually free) within an instantiation of $\#s$. In most cases, elements of prefixes will again be schema variables. The exact meaning of a prefix property depends on the type of $\#s$. $*$

3.7 Example: For the following schema variables $\#t$ for terms, we will define one prefix property as a set of VariableSV whose instantiations are (exclusively) allowed to occur freely within an instantiation of $\#t$. In example 3.4, for a schema variable $\#p$ for formulas we would analogously have

$$\mathrm{prefix}_{\#p} = \{\#x\}. \qquad *$$

The parts shaded in the following tables are needed to treat dynamic aspects of JavaCardDL and are not used for FOL (explanations can be found in Sect. 3.2.2, where schema variables for dynamic logic are defined). The properties are included in the tables already at this point to provide complete descriptions.

Term Schema Variables (TermSV)	
sort	Instantiations allowed for this schema variable must have exactly this sort.
prefix	A subset of **S**. It determines the logical variables that may occur freely within instantiations t of this schema variable. The set may contain • VariableSV whose particular instantiation may occur freely in t • ContextSV $\#ct$: If $\#ct$ is to be instantiated with the formula ψ, then the variables Bound(ψ) may occur freely in t (ContextSV are defined below, and most probably the reader should skip this item for the time being) (instantiations t must not contain further free variables).
pvPrefix	A subset of **S** that contains only PVariableSV. It determines those program variables that may occur free within instantiations of this schema variable. If this property has the value **V**, then the following program variables are allowed to occur free: • Variables that do not occur as instantiations of any PVariableSV of **S** • Variables that are instantiations of elements of **V**.
A TermSV is to be instantiated with terms.	
$J_{\mathrm{TermSV}}(\alpha) = \alpha$	

Formula Schema Variables (FormulaSV)	
prefix	(exactly as the prefix property of TermSV)
pvPrefix	(exactly as the pvPrefix property of TermSV)
A FormulaSV is to be instantiated with formulas.	
$J_{\mathrm{FormulaSV}}(\alpha) = \alpha$	

3.8 Remark: If a TermSV or FormulaSV $\#s$ occurs in the formula φ within the scope of an operator (e.g. a quantifier) binding an explicit logical variable x, then it is not possible to choose the prefix-property of $\#s$ such that x (but no other variable) is allowed to occur freely within instantiations of $\#s$. The is caused by the restriction that the prefix-property of $\#s$ must not contain logical variables, but only VariableSV. Thus the formula $\forall x.p(x)$ is usually *not* an instance of $\forall x.\#s$, but it could be an instance of $\forall \#x.\#s$, depending on the prefix of $\#s$.

This is an arbitrary convention, and it would be easy to extend the definition of prefixes to make explicit variables possible; it is, however, always possible to replace a logical variable with a VariableSV without changing semantics, so a more general definition would only be a minor improvement. Indeed we will usually require schematic formulas not to contain logical variables at all (within the KeY project, this is also required for all taclets). $*$

3.9 Example: We have a look at the meaning formula (i.e. an axiomatic representation, which is defined in Sect. 3.3.2) of the rule (all_left), for quantified variables of sort S:

$$\varphi_1 = \forall \#x.\#p \rightarrow \{\#x \quad \#t\}\#p$$

The schema variables within this formula and their particular properties are (properties not existing for a type are simply left out in the following table)

Symbol	Type	sort	prefix
$\#x$	VariableSV	S	
$\#p$	FormulaSV		$\{\#x\}$
$\#t$	TermSV	S	$\emptyset$

$*$

As mentioned before, the following schema variable type is used to model the position of an application of a rewrite rule within a term or formula in an explicit way. These *context schema variables* are unary, and are instantiated with formulas in which a special symbol marks the position at which the argument is to be substituted.

3.10 Example: A formula in which a context variable $\#ct$ occurs is

$$\#ct(c)$$

where c is a constant symbol. By instantiating $\#ct$ with the formula $p(f(a_{\#ct}, d))$ (where $a_{\#ct}$ is the insertion symbol for the variable $\#ct$) the instance

$$\iota(\#ct(c)) = p(f(c, d))$$

is obtained. $*$

To define the instantiation of context variables, we employ *f-substitutions*, that are introduced in appendix A.1. For the time being, the only notable difference between f-substitutions and normal substitutions is that the former ones replace function and predicate symbols instead of logical variables. For example 3.10, this would be

$$\iota(\#ct(c)) = \{a_{\#ct}/c\}p(f(a_{\#ct}, d)) = p(f(c, d)).$$

Context Schema Variables (ContextSV)	
sort	The sort of the nullary function symbol $a_{\#ct}$ (see below) that may occur within ContextSV instantiations, or $\perp$.
unique	A boolean property, determining whether the symbol $a_{\#ct}$ (see below) has to occur exactly once within the instantiation; this flag is mainly included to make the following proofs easier, its meaning will be explained later in detail.
SUL	If this flag is true for a ContextSV $\#ct$, then an instantiation $\iota(\#ct)$ of $\#ct$ must not contain the symbol $a_{\#ct}$ below any modalities (like updates and box or diamond operators).
A ContextSV $\#ct$ is a unary symbol ($\#ct(t)$), unlike the other schema variables we have defined so far. It is to be instantiated with a closed formula ψ, which contains a special constant or nullary predicate symbol $a_{\#ct}$ (see Rem. 3.5). There are two different cases: • The sort property is *not* $\perp$ (and the argument t has to be a term[13]): Then $a_{\#ct}$ is a constant of this sort • The sort property is $\perp$ (and the argument t has to be a formula): Then $a_{\#ct}$ is a nullary predicate symbol If the unique-property is true, $a_{\#ct}$ has to occur exactly once in ψ (otherwise there may by arbitrarily many occurrences, or none). When instantiating $\#ct$, the argument t is substituted for $a_{\#ct}$, and the resulting formula is substituted for $\#ct(t)$. The set of variables bound within ψ above each occurrence of $a_{\#ct}$ (i.e. for multiple occurrences the intersection of the sets for each occurrence) shall be denoted by $\mathrm{Bound}(\psi)$. If $a_{\#ct}$ does not occur in ψ at all, we define $\mathrm{Bound}(\psi) = \mathit{Var}$.	
$J_{\mathrm{ContextSV}}(\alpha, r) = \{a_{\#ct}/r\}\alpha$ (using f-substitutions, as defined in appendix A.1)	

3.11 *Remark:* We always assume that the schematic formula φ is syntactically correct for every valid instantiation of the schema variables (the validity of an instantiation does not depend on φ, but on the types and properties of the particular schema variables). Usually φ is also required to be closed for every valid instantiation. This implies that

- VariableSV occur only bound within φ
- If a TermSV or FormulaSV $\#t$ contains a VariableSV $\#v$ within its prefix, then $\#v$ is bound above each occurrence of $\#t$
- If a TermSV or FormulaSV $\#t$ contains a ContextSV $\#ct$ within its prefix, then $\#t$ occurs only within arguments of $\#ct$. $*$

3.12 *Example:* We consider the meaning formula of a rule expressing the symmetry of a relation p with argument sort S:

$$\varphi_2 = \#ct(p(\#s, \#t)) \leftrightarrow \#ct(p(\#t, \#s))$$

The schema variables within this formula are

[13]Following Def. 3.3, this is in fact an implication of Rem. 3.11 below.

Symbol	Type	sort	prefix	unique
$\#ct$	ContextSV	$\perp$		*true*
$\#s$	TermSV	S	$\{\#ct\}$	
$\#t$	TermSV	S	$\{\#ct\}$	

and valid instances/instantiations are (note that the instantiations in the last line would not be allowed for $\#ct \notin \text{prefix}_{\#s}$ or $\#ct \notin \text{prefix}_{\#t}$)

φ_2	$\#ct$	$\#s$	$\#t$
$p(c,d) \leftrightarrow p(d,c)$	$a_{\#ct}$	c	d
$(r \wedge p(c,d)) \leftrightarrow (r \wedge p(d,c))$	$r \wedge a_{\#ct}$	c	d
$\forall x.\forall y.p(x,y) \leftrightarrow \forall x.\forall y.p(y,x)$	$\forall x.\forall y.a_{\#ct}$	x	y

If the unique-flag of $\#ct$ was false, the following instances would additionally be correct (in the last line, the instantiations of $\#s$ and $\#t$ are irrelevant):

φ_2	$\#ct$	$\#s$	$\#t$
$(p(c,d) \wedge p(c,d)) \leftrightarrow (p(d,c) \wedge p(d,c))$	$a_{\#ct} \wedge a_{\#ct}$	c	d
$(\forall x.p(x,c) \wedge \forall x.p(x,c)) \leftrightarrow (\forall x.p(c,x) \wedge \forall x.p(c,x))$	$\forall x.a_{\#ct} \wedge \forall x.a_{\#ct}$	x	c
$r \leftrightarrow r$	r		

$*$

3.13 *Example:* The following axiom, in which $\#s$ and $\#t$ are TermSV, represents the congruence property of the equality $\doteq$:

$$\varphi_3 = \#s \doteq \#t \rightarrow \big(\#ct(\#s) \leftrightarrow \#ct(\#t)\big)$$

If Rem. 3.11 is supposed to hold for φ_3, then we must have

$$\text{prefix}_{\#s} = \text{prefix}_{\#t} = \emptyset$$

because φ_3 contains occurrences of $\#s$ and $\#t$ that do not lie within the scope of $\#ct$ or any operator binding VariableSV. This means that instantiations of $\#s$ and $\#t$ are not allowed to contain free variables. Otherwise, there would also be instances of φ_3 that are not closed. $*$

3.2.2. JavaCardDL Schema Variables

The following schema variable types are very specific to JavaCardDL, and while they could of course be adapted to other languages as well, we are only aiming at treating JavaCardDL as completely as feasible.

From the set of types that are used to handle the Java language (in the KeY project), we concentrate on the most important ones, and make some assumptions about the remaining types that are needed (and sufficient) to discuss the validity of schematic formulas. Within the following pages, we use terminology of [GJSB00].

For the following type, we are referring to a set of "unused" program variables, denoted by $PVar_u$, which basically contains all program variables not explicitly used in any taclet (an exact definition of $PVar_u$ is provided in Sect. 4.6, as it does not make any sense without a specific application).

Program Variable Schema Variables (PVariableSV)	
javaType	Instantiations of this PVariableSV must have exactly the given Java type.
unusedOnly	Only the elements of $PVar_u$ are possible instantiations of this PVariableSV, and the instantiation of this PVariableSV must not occur within the instantiation of any other schema variable, except when this is explicitly allowed by a suitable pvPrefix-property (see below).
PVariableSV are to be instantiated with program variables (as is Sect. 2.3), that may however be declared either globally or locally, corresponding to free and bound logical variables. As with logical variables, we do not allow the instantiation of a PVariableSV to occur bound within the instantiation of any other schema variable.	
$J_{\mathrm{PVariableSV}}(\alpha) = \alpha$	

The unusedOnly-property will be used for PVariableSV declared as *new* in a taclet definition, see Sect. 3.3.1 below.

As for logical variables, we introduce prefixes for PVariableSV, and therefore add the following properties to TermSV and FormulaSV:

Term Schema Variables (TermSV)	
pvPrefix	A subset of $\mathbf{S}$ that contains only PVariableSV. It determines those program variables that may occur free within instantiations of this schema variable. If this property has the value $\mathbf{V}$, then the following program variables are allowed to occur free: • Variables that do not occur as instantiations of any PVariableSV of $\mathbf{S}$[14] • Variables that are instantiations of elements of $\mathbf{V}$.

Formula Schema Variables (FormulaSV)	
pvPrefix	(exactly as the pvPrefix property of TermSV)

We do not treat ContextSV as "producers" of program variables that may occur within instantiations, i.e. contrary to the prefix-property for logical variables ContextSV are not allowed to be elements of the pvPrefix-set. This is in fact no restriction, as we have defined that the scopes of program variables do not exceed program blocks (Sect. 2.3). Program variables defined within a program block (within the instantiation of a ContextSV) above a certain position are thus not visible. For an illustration, consider the formula

$$\langle \mathbf{int}\ \mathrm{i} = 1; \rangle a_{\#ct}$$

which is a possible instantiation of a ContextSV $\#ct$. The symbol $a_{\#ct}$ in this formula is not regarded to lie within the scope of the local program variable i.

[14]This is necessary because program variables can occur free within top-level formulas, contrary to logical variables (in the KeY system). Hence it is not feasible to specify *all* program variables that can turn up within instantiations of schema variables.

Label Schema Variables (LabelSV)
LabelSV are to be instantiated with labels l, that occur in Java programs to give statements names. LabelSV are treated exactly like bound variables (the labelling of a statement corresponds to an operator binding a variable, and the occurrence of a label as argument of a jump statement to the occurrence of a variable), which means that we do not allow statements labelled with l to occur within the instantiations of other schema variables, and we do not allow two LabelSV to be instantiated with the same label.[15]
$J_{\text{LabelSV}}(\alpha) = \alpha$

The following two schema variable types are most important for JavaCardDL, and treat statements and expressions. For statements we will again introduce a new kind of prefix, namely a set controlling potential abrupt terminations of code fragments. According to [GJSB00], Sect. 14.1, a statement can complete abruptly by the following reasons:

- Execution of a **break**- or **continue**-statement
- Execution of a **return**-statement
- Occurrence of an exception.

Which of these reasons may really turn up for a given concrete statement, and with which parameters, depends on static properties of the enclosing program, e.g. whether there are enclosing loops. Thus we formulate a criterion (for the first two items of the list), based on a simple static analysis of statements, that determines whether a given statement is compatible with a particular context. This criterion will later be used to distinguish valid instantiations of schema variables for statements.

3.14 Definition (Compliant Termination Behaviour): Let **T** be a set of Java statements of the following kinds:

- **return**-statements, which can optionally have a program variable as argument (x has to be a program variable, and we allow **T** to contain at most one return-statement): **return**, **return** x
- **break**- and **continue**-statements, which can either be anonymous or have a label as argument: **break**, **continue**, **break** l, **continue** l

We call a fragment α of Java code *compliant* to **T** regarding termination behaviour, if

- For every return-statement s_1 occurring in α (and that is not enclosed in α by a **method−frame**): **T** contains a return-statement s_2, such that either both statements have no argument, or the argument of s_1 is assignable to the argument of s_2 (which, as defined above, is a program variable)

[15] Again, such a condition does not exist within the KeY system for the taclet mechanism; it is always possible to establish the condition by renaming labels appropriately.

- Every break- or continue-statement s occurring in α whose target does not lie within α is an element of $\mathbf{T}$. $*$

The definition does not contain any restrictions regarding the exceptions α might throw. This is an arbitrary decision (that complies to the KeY policy towards exceptions, however), and it would as well be possible to allow **throw**-statements as elements of the set $\mathbf{T}$; but as we do not need such restrictions in this document, and the definition of compliant Java fragments would become a lot more difficult, we omit them. Exceptions will however be treated in Sect. 4.6.

3.15 Example: We consider the statement α, given by[16]

m : { **break**; **return** 5; **break** m; }

A set $\mathbf{T}$ to which α is compliant has to fulfil the following propositions:

- $\mathbf{T}$ has to contain the statement **break**, which is not enclosed by a target statement in α
- $\mathbf{T}$ has to contain a statement **return** r, where r is a program variable to which the literal 5 can be assigned, e.g. of Java type **int**
- $\mathbf{T}$ does not have to contain the statement **break** m, whose target is the enclosing statement block and thus lies within α.

Provided that the type of r is compatible, α therefore is compliant to

$$\mathbf{T} = \{\mathbf{break}, \mathbf{return}\ \mathrm{r}\}. \qquad *$$

[16] We ignore that parts of this statement are actually unreachable, which causes compile-time errors according to [GJSB00].

<table>
<tr><th colspan="2">Statement Schema Variables (StatementSV)</th></tr>
<tr><td>pvPrefix</td><td>(exactly as the pvPrefix property of TermSV)</td></tr>
<tr><td>jumpPrefix</td><td>A set J, determining the ways an instantiation of this StatementSV may choose to terminate abruptly. J can contain:
• At most one PContextSV $\#pct \in \mathbf{S}$ (see below)
• A number of jump statements, which may be chosen from:
– return-statements ($\#pv \in \mathbf{S}$ has to be a PVariableSV): return, return $\#pv$
– break- and continue-statements, either anonymous or with labels ($\#l \in \mathbf{S}$ has to be a LabelSV): break, continue, break $\#l$, continue $\#l$.
Reflecting Def. 3.14, which states that the set T (of this definition) has to contain at most one return-statement, we add the following restriction: J must not contain more than one return-statement, and J must not contain both a PContextSV and a return-statement.</td></tr>
<tr><td colspan="2">A StatementSV $\#s$ is to be instantiated with Java statements respecting scopes (following Def. 2.1),[17] that do not contain any method−frame-statements.[18]
To decide about the validity of an instantiation ι (regarding $\#s$), first a set $\mathbf{T} = \mathbf{T}_{\#s}(\iota)$ of statements is derived from the jumpPrefix-property as the smallest set with:
• For a PContextSV $\#pct$ within the jumpPrefix: $\mathrm{Jumps}(\iota(\#pct)) \subset \mathbf{T}$ (see below for PContextSV)
• For a statement st within the jumpPrefix: $\iota(st) \in \mathbf{T}$.
The instantiation ι is invalid, if $\iota(\#s)$ is not compliant with T (following Def. 3.14).</td></tr>
<tr><td colspan="2">$J_{\mathrm{StatementSV}}(\alpha) = \alpha$</td></tr>
</table>

3.16 *Example:* In example 3.15, for a statement α the following (minimal) set of jump statements was determined, to which α is termination compliant:

$$\mathbf{T} = \{\mathbf{break}, \mathbf{return}\ \mathrm{r}\}.$$

Then α also is a valid instantiation of a StatementSV $\#s$ that fulfils the following condition:

$$\mathrm{jumpPrefix}_{\#s} \supset \{\mathbf{break}, \mathbf{return}\ \#r\}$$

provided that r and $\#r$ have the same Java type (this condition is not necessary however, the jumpPrefix can also be chosen differently, see example 3.18 below). $*$

[17] This is not demanded in the KeY system, but really seems to be a desirable feature. Otherwise the instantiation of a StatementSV can alter the binding of subsequent variable occurrences, as it is demonstrated in Sect. 2.3.

[18] This restriction has been added for reasons of simplicity, and could be removed without difficulty.

3.17 *Remark (PVariableSV within Prefixes):* In Def. 3.14, for a **return**-statement within the jumpPrefix of a StatementSV, the program variable given as argument of the statement is not important as a location; only the type of the variable determines whether a given value can be assigned. To make the considerations of the following sections easier, we assume however that a PVariableSV occurring within the jumpPrefix of a StatementSV $\#s$ also is an element of the pvPrefix of $\#s$.[19] $*$

Expression Schema Variables (ExpressionSV)	
javaType	Instantiations (expressions) allowed for this schema variable must have exactly this Java type.
pvPrefix	(exactly as the pvPrefix property of TermSV)
ExpressionSV are to be instantiated with Java expressions.	
$J_{\mathrm{ExpressionSV}}(\alpha) = \alpha$	

ExpressionSV do not have a jumpPrefix-property, as the only way an expression may terminate abruptly is through an exception (this way of termination is not considered for StatementSV either at the time, but only in Sect. 4.6).

Both for PVariableSV and for ExpressionSV, we restrict the instantiations to expressions having a certain Java type (and, to make things a bit easier, we further assume that no implicit conversions are applied to the types of instantiations).

The following schema variable type is similar to the ContextSV type, but describes not an enclosing formula, but enclosing Java blocks and trailing statements. Usually the schema variable occurs only as the top-level statement of a program block.

[19] As it is always possible to introduce a dedicated PVariableSV, only to be used within the jumpPrefix and the pvPrefix of a StatementSV, without further occurrences in a formula, this is no restriction.

Program Context Schema Variables (PContextSV)
A PContextSV $\#pct$ is a unary symbol, and is written as a statement having one argument ($\#pct(\beta)$, like ContextSV). It is to be instantiated with a list α of Java statements (i.e. a Java program fragment), in which exactly once a special symbol $a_{\#pct}$ occurs (which again acts as a statement list placeholder, and is to be replaced with β; also see Rem. 3.5). There are only a few positions in a program α where $a_{\#pct}$ may appear, namely only within the first statement of α, and the only program constructs that may precede $a_{\#pct}$ within this statement are the opening parts of the following environments: • Blocks marked by braces $\{\ldots\}$ • Try-blocks • **method−frame**-blocks. Given an instantiation $\alpha = \iota(\#pct)$, we define a set $\mathrm{Jumps}(\alpha)$ of "admissible" reasons for the abrupt completion of instances of β, corresponding to Def. 3.14. The elements of $\mathrm{Jumps}(\alpha)$ are determined by those blocks within α that enclose $a_{\#pct}$, but that do not contain any **method−frame** itself containing $a_{\#pct}$: • For a (try-)block with label l, **break** l $\in \mathrm{Jumps}(\alpha)$ • For a **method−frame** with return variable r, **return** r $\in \mathrm{Jumps}(\alpha)$; if the frame does not have a return variable, **return** $\in \mathrm{Jumps}(\alpha)$.
$J_{\mathrm{PContextSV}}(\alpha, r) = \alpha'$, where α' is obtained from α by replacing $a_{\#pct}$ with r

There is no need to allow program variable prefixes (pvPrefix) of StatementSV or ExpressionSV to contain PContextSV (contrary to ContextSV, which may appear within the prefixes of TermSV and FormulaSV), as there must not be any variable declarations preceding the symbol $a_{\#pct}$ within the instantiation of a PContextSV.

We will usually use the notation

$$..\ \alpha\ ..._{\#pct} \quad := \#pct(\alpha)$$

for program contexts.

3.18 Example (Jump Statements for Program Contexts): For the following instantiation γ of a PContextSV $\#pct$

```
l0 : {
  method-frame ( x ) {
    l1 : try {
      a_#pct
    }
    ...
  }
}
```

the set Jumps(γ) is determined by looking at the two inner blocks (the block labelled with l0 is outside of a **method−frame** and thus not considered):

$$\text{Jumps}(\gamma) = \{\textbf{break}\ \text{l1}, \textbf{return}\ \text{x}\}.$$

Another possible choice for the jumpPrefix from example 3.16 is

$$\text{jumpPrefix}_{\#s} = \{\textbf{break}, \#pct\}$$

and then the instantiation $\iota = \{\#s/\alpha, \#pct/\gamma\}$ is valid, provided that the types of the program variables r and x are compatible (α is the Java statement from example 3.15). Putting it all together, for the schematic program

$$\varphi_4 = ..\ \textbf{while}\ (\ \textbf{true}\ )\ \{\#s\}\ ..._{\#pct}$$

this instantiation would describe the instance $\iota(\varphi_4)$, in which the **break**-statement of α is "caught" by the loop, and the **return**-statement by the **method−frame** of the context:

```
l0 : {
  method-frame ( x ) {
    l1 : try {
      while ( true ) { m : { break; return 5; break m; } }
    }
    ...
  }
}
```

*

3.19 Remark (Additions to Rem. 3.8): Rem. 3.8 (about the possibility to forbid explicit occurrences of logical variables, without loss of generality) also applies to program variables and labels. Following that remark, we will later usually forbid the occurrences of

- explicit locally defined (i.e. bound) program variables, which can always be replaced by PVariableSV
- explicit labels, which can be replaced by LabelSV. *

3.20 Remark (Additions to Rem. 3.11): For φ to be syntactically correct for all valid instantiations of the schema variables **S**, the following conditions are necessary:

- If the jumpPrefix of a StatementSV $\#s$ contains a jump statement, $\#s$ has to be enclosed by an appropriate target statement (being labelled with the correct LabelSV, having a compatible result variable, etc.), such that there is no **method−frame** and no PContextSV between $\#s$ and the target
- LabelSV must not occur "freely", i.e. every occurrence of a LabelSV $\#l$ has to be within a statement labelled with the $\#l$ (and must again not be shadowed by a **method−frame** or a PContextSV). *

Finally, we need a possibility to prevent contexts from containing modalities at illegal positions; this is needed to model the keyword `sameUpdateLevel` for taclets. We only support a simplified variant of this flag, and do not treat updates in a way distinguished from other modal operators:

Context Schema Variables (ContextSV)	
SUL	If this flag is true for a ContextSV $\#ct$, then an instantiation $\iota(\#ct)$ of $\#ct$ must not contain the symbol $a_{\#ct}$ below any modalities (like updates and box or diamond operators).

3.3. Taclets with Schema Variables

Having the schema variable toolbox at hand, it is possible to define taclets that represent more general rules. Namely, by using schematic formulas and terms within the definition of taclets, whole classes of rules can be described, whose elements are derived by instantiating schema variables in different ways.

3.21 *Example:* A quite simple rewrite taclet t_2, containing the TermSV $\#x$ and $\#y$ is

$$\texttt{find}(\#x + \#y) \quad \texttt{replacewith}(\#y + \#x)$$

which subsumes the instantiated taclet $t_{2,\iota}$ with $\iota = \{\#x/z, \#y/\mathrm{i}\}$:

$$\texttt{find}(z + \mathrm{i}) \quad \texttt{replacewith}(\mathrm{i} + z)$$

An application of $t_{2,\iota}$ is

$$\frac{\vdash \ \exists z.\langle \mathrm{i} \ = \ 2;\rangle \mathrm{i} + z \doteq 0}{\vdash \ \exists z.\langle \mathrm{i} \ = \ 2;\rangle z + \mathrm{i} \doteq 0}\ t_{2,\iota}$$

*

The statement `varcond` that can be included in the definition of a taclet is used to make instantiations of schema variables fulfil certain conditions; the syntax of the statement is (as far as it is relevant in this place):

$$\texttt{varcond}(c_1, \ldots, c_l)$$

where each c_i can be one of

- "$\#a$ `not free in` $\#b$": The instantiation of the VariableSV $\#a$ must not occur freely within instantiations of $\#b$; $\#b$ may be a TermSV or FormulaSV
- "$\#v$ `new`": The instantiation of the PVariableSV $\#v$ has to be a new program variable, i.e. a variable not already occurring in the proof or within the instantiations of other schema variables[20]

[20] In the KeY system, this variable condition turns up in two different shapes, and has the possibility to specify the Java type the new variable shall be given; this is necessary for the untyped PVariableSV of KeY, but not for our definition.

- "$\#a$ `new depending on` $\#b$": The instantiation of the TermSV $\#a$ must be a new skolem function, whose arguments are the meta variables (i.e. free variables of the calculus) that occur within the instantiation of the TermSV or FormulaSV $\#b$. As meta variables are not treated in this document, the second argument actually has no meaning for us, and the skolem function is a skolem constant.[21]

To control program variables that can occur freely within instantiations of schema variables, a goal template can be equipped with a statement

$$\texttt{addprogvar}(\mathbf{V}_i)$$

where $\mathbf{V}_i$ is a set (list) of PVariableSV; the effect of such a directive is that the concerned program variables are considered as "free variables" within the sequent described by the goal template, and can thus appear in instantiations freely without being bound above.

3.3.1. Restrictions on Schema Variables

The usage of schema variables in taclets is restricted in different aspects; in particular the values of many properties are not allowed to be chosen arbitrarily, and ContextSV must not be used "manually" when defining a taclet. To describe this restrictions, let $\#s_1, \ldots, \#s_k$ be the schema variables of a taclet t (i.e. occurring in the definition of the taclet).

If t is a rewrite taclet, first a further schema variable, a ContextSV $\#ct$ is introduced, representing the position of an application of the taclet.

3.22 *Example (Example 3.21 continued):* To apply the taclet t_2 from example 3.21, the instantiation ι would be continued by

$$\iota(\#ct) = \exists z.\langle \mathrm{i} = 2; \rangle a_{\#ct} \doteq 0. \qquad *$$

We demand that t does not contain explicit logical variables, explicit bound program variables and explicit labels (Rem. 3.8 and 3.19), i.e. that those symbols are replaced with schema variables. The following properties of $\#s_1, \ldots, \#s_k, \#ct$ are fixed and cannot be chosen when defining t:

- The value of the prefix-property of a TermSV or FormulaSV $\#s_i$ is determined by the terms and formulas in which $\#s_i$ occurs, to be the smallest set with:[22]
  - If the VariableSV $\#s_j = \#v_j$ is bound by an operator above any occurrence of $\#s_i$, then $\#v_j \in \mathbf{P}$, except there is a variable condition

    $$\#v_j \ \texttt{not free in} \ \#s_i$$

  - If t is a rewrite taclet, then $\#ct \in \mathbf{P}$, provided that there is at most one `replacewith`-statement, and $\#s_i$ does not occur in `if`- or `add`-statements

[21] In the KeY system, the first argument $\#a$ is not a TermSV, but has a distinguished type only used for introducing skolem symbols.

[22] This is taken from [Gie02], with some of the conditions having been removed, as they are implied by Rem. 3.20 we are going to use.

- If t is a rewrite taclet, then the unique-flag of the ContextSV $\#ct$ is true, and the SUL-flag is true iff the directive `sameUpdateLevel` is given
- For a PVariableSV $\#s_i = \#v_i$, the unusedOnly-flag is true iff[23] there is a variable condition

$$\#v_i \ \texttt{new}$$

- For a schema variable $\#s_i$ having a pvPrefix-property, its value **P** contains a PVariableSV $\#s_j = \#v_j$ iff[24]
  - the unusedOnly-flag of $\#v_j$ is false or
  - each occurrence of $\#s_i$ lies within the scope of a declaration of $\#v_j$,[25] or it is part of a goal template in which $\#v_j$ is argument of a `addprogvar`-statement
- For a StatementSV $\#s_i$, the value **J** of the jumpPrefix-property is determined by those blocks and PContextSV that enclose any occurrence of $\#s_i$, but that do not contain a **method−frame** or another PContextSV itself containing $\#s_i$ (similar to the map Jumps defined for PContextSV):
  - Provided that $\#s_i$ is enclosed by a suitable target statement respectively, **break**, **continue**, **break** $\#l$, **continue** $\#l \in \mathbf{J}$
  - For a **method−frame** with result variable r (this can also be a PVariableSV), **return** $\#v \in \mathbf{J}$, where $\#v$ shall be a new PVariableSV having the same type as r (and that is also to be included in the pvPrefix of $\#s_i$, see Rem. 3.17); if the frame does not have a result variable, **return** $\in \mathbf{J}$
  - For an enclosing PContextSV $\#pct$, $\#pct \in \mathbf{J}$.

3.23 Example: For the rewrite taclet t_3, defined by

$$\texttt{if}\big(\#i \doteq 0 \ \vdash\ \big) \quad \texttt{find}\big(\langle ..\ \#l\text{: }\mathbf{if}\ (\ \#i\text{==}\ 0\ )\ \#s;\ ..._{\#pct}\rangle \#p\big) \quad \texttt{sameUpdateLevel}$$
$$\texttt{replacewith}\big(\langle ..\ \#l\text{: }\{\ \#s;\ \}\ ..._{\#pct}\rangle \#p\big)$$

the following schema variable properties would be enforced:

Sym.	Type	u.Only	prefix	pvPrefix	jumpPrefix	unique	SUL
$\#ct$	ContextSV					*true*	*true*
$\#pct$	PContextSV						
$\#p$	FormulaSV		$\{\#ct\}$	$\{\#i\}$			
$\#s$	StatementSV			$\{\#i\}$	$\{\#pct, \mathbf{break}\ \#l\}$		
$\#l$	LabelSV						
$\#i$	PVariableSV	*false*					

∗

[23] It would also be reasonable to set the flag for PVariableSV occurring only bound within a taclet, i.e. for PVariableSV that occur within the scope of an appropriate variable declaration within a Java program. This would however require program variable prefixes to be defined also for ContextSV and PContextSV.

[24] The following conditions are not very useful for PContextSV not appearing in `find`- or `if`-statements (but only in `replacewith` or `add`), but this is not relevant for the KeY system. Otherwise, it would again be necessary to introduce prefixes for PContextSV.

[25] And this also implies that $\#s_i$ is not enclosed by a **method−frame** that also lies within the scope of the declaration.

To ensure that instantiated taclets are syntactically correct, we refer to Rem. 3.11 and 3.20 made in Sect. 3.2. The remarks are required to hold for the following formulas of a taclet t:

- For each formula of the sequents $\Gamma_{\texttt{if}} \vdash \Delta_{\texttt{if}}, \Gamma_{\texttt{add},1} \vdash \Delta_{\texttt{add},1}, \ldots, \Gamma_{\texttt{add},k} \vdash \Delta_{\texttt{add},k}$ (as in Sect. 3.1)
- If t is a lemma taclet, for the formulas of the sequents A, and for $B_1, \ldots, B_k$
- If t is a rewrite taclet, for the terms or formulas $\#ct(A), \#ct(B_1), \ldots, \#ct(B_k)$.

3.3.2. Meaning Formulas of Taclets

While possible instantiations of a taclet t have been determined through the definition of schema variables, the instructions for applying t in a certain situation was only roughly described upon the introduction of taclets. In the KeY system, this application mechanism is hard-coded in Java and of considerable complexity, a detailed analysis is therefore beyond the scope of this document. The meaning of a taclet (i.e. the semantics belonging to the abstract syntax of a taclet, as given in Sect. 3.1) is rather specified by the definition of a map $\mathfrak{M}$, which assigns each taclet t a *meaning formula* (for which we mostly follow [Hab00]), which describes the effect of a taclet in an axiomatic way. In general, the meaning formula of a taclet contains schema variables and should thus be regarded as a whole family of formulas, whose members are obtained by instantiating the variables.

This map $\mathfrak{M}$ can be seen as a method to reduce a calculus **Ta**, whose rules **R** are given by taclets, to a calculus **HT** with a small set of basic rules (e.g. propositional rules) and set $\mathbf{A} = \mathfrak{M}(\mathbf{R})$ of axioms (this is formalised in Def. 4.11). By postulating that proofs of the calculus **Ta** can be translated to proofs of **HT**

$$\vdash_{\mathbf{Ta}} \varphi \implies \vdash_{\mathbf{HT}} \varphi$$

a criterion for the correctness of a taclet application mechanism is given, provided that the basic rules of the calculus **HT** and the map $\mathfrak{M}$ are considered as "correct".

A possible calculus **HT** (a Hilbert-style calculus) is introduced in Sect. 4.2.

In the following paragraphs, we will describe a construction of the meaning formula for a taclet t (mostly taken from [Hab00]). In several aspects, there is however no "canonical" way to interpret the description of a taclet, and the definitions given in this place should therefore be regarded as a proposal. A more detailed explanation of the shape both of meaning formulas in general and of the following definitions can be found in [BGH+03]. We are using the same notation for a taclet as in Sect. 3.1:

$$\begin{array}{l}
[\,\texttt{if}(\Gamma_{\texttt{if}} \vdash \Delta_{\texttt{if}})\,] \quad [\,\texttt{find}(A)\,] \quad [\,\texttt{sameUpdateLevel}\,] \\
\quad [\,\texttt{varcond}(c_1, \ldots, c_l)\,] \\
\quad [\,\texttt{replacewith}(B_1)\,] \quad [\,\texttt{add}(\Gamma_{\texttt{add},1} \vdash \Delta_{\texttt{add},1})\,] \quad [\,\texttt{addprogvar}(\mathbf{V}_1)\,]; \\
\qquad \vdots \\
\quad [\,\texttt{replacewith}(B_k)\,] \quad [\,\texttt{add}(\Gamma_{\texttt{add},k} \vdash \Delta_{\texttt{add},k})\,] \quad [\,\texttt{addprogvar}(\mathbf{V}_k)\,]
\end{array}$$

To get a shorter notation, in the following paragraphs we identify a sequent with the disjunction of its formulas

$$\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m \qquad \mapsto \qquad \neg\varphi_1 \vee \cdots \vee \neg\varphi_n \vee \psi_1 \vee \cdots \vee \psi_m,$$

and for missing `if`-, `add`- or `find`-statements in a taclet description we define the argument to be the empty sequent respectively, for a missing `replacewith`-statement the argument of the `find`-statement shall be used.

For a lemma taclet t the central part of the meaning formula is

$$\mathfrak{M}'(t) = \bigwedge_{i=1}^{k} \Big(B_i \vee \big(\Gamma_{\mathtt{add},i} \vdash \Delta_{\mathtt{add},i}\big)\Big) \to \Big(A \vee \big(\Gamma_{\mathtt{if}} \vdash \Delta_{\mathtt{if}}\big)\Big)$$

and for a rewrite taclet

$$\mathfrak{M}'(t) = \bigwedge_{i=1}^{k} \Big(\big(\#ct(A) \leftrightarrow \#ct(B_i)\big) \to \big(\Gamma_{\mathtt{add},i} \vdash \Delta_{\mathtt{add},i}\big)\Big) \to \big(\Gamma_{\mathtt{if}} \vdash \Delta_{\mathtt{if}}\big).$$

For the complete meaning formula it is furthermore necessary to treat variable conditions; we assume that t contains the statement

$$\begin{aligned}&\mathtt{varcond}(\#t_1 \ \mathtt{new\ depending\ on}\ \_, \ldots, \#t_m \ \mathtt{new\ depending\ on}\ \_,\\ &\qquad \#v_1 \ \mathtt{new}, \ldots, \#v_l \ \mathtt{new},\\ &\qquad \_ \ \mathtt{not\ free\ in}\ \_, \ldots).\end{aligned}$$

We need additional VariableSV, not occurring in $\mathfrak{M}'(t)$:

- m variables $\#x_1, \ldots, \#x_m$, having the same sorts as $\#t_1, \ldots, \#t_m$
- l variables $\#y_1, \ldots, \#y_l$, having the sorts assigned to the PVariableSV $\#v_1, \ldots, \#v_l$.

$\mathfrak{M}''(t)$ shall denote the formula obtained from $\mathfrak{M}'(t)$ by replacing $\#t_1, \ldots, \#t_m$ with the VariableSV $\#x_1, \ldots, \#x_m$. The complete meaning formula finally is

$$\mathfrak{M}(t) = \exists \#x_1. \ldots \exists \#x_m. \exists \#y_1. \ldots \exists \#y_l. \{\#v_1 \ \mathtt{:=}\ \#y_1, \ldots, \#v_l \ \mathtt{:=}\ \#y_l\} \mathfrak{M}''(t).$$

3.24 *Example:* The meaning formulas of the rewrite taclets t_2 and t_3 from the examples 3.21 and 3.23 are

$$\begin{aligned}\mathfrak{M}(t_2) &= \#ct\big(\#x + \#y\big) \leftrightarrow \#ct\big(\#y + \#x\big)\\ \mathfrak{M}(t_3) &= \#i \doteq 0 \to \big(\#ct(\varphi_1) \leftrightarrow \#ct(\varphi_2)\big)\end{aligned}$$

where

$$\varphi_1 = \langle ..\ \#l\colon \mathbf{if}\ (\ \#i{==}\ 0\ )\ \#s;\ ..._{\#pct}\rangle \#p \qquad \varphi_2 = \langle ..\ \#l\colon \{\ \#s;\ \}\ ..._{\#pct}\rangle \#p. \qquad *$$

3.25 *Example:* For the taclet t_4, in which $\#u$ and $\#v$ denote PVariableSV, and $\#e$ an ExpressionSV

$$\texttt{find}\big(\langle .. \ \#u = \ \#e; \ ..._{\#pct}\rangle \#p\big) \quad \texttt{varcond}\big(\#v \ \texttt{new}\big)$$
$$\texttt{replacewith}\big(\langle .. \ \textbf{int} \ \#v = \ \#e; \ \#u = \ \#v; \ ..._{\#pct}\rangle \#p\big)$$

the meaning formula is

$$\mathfrak{M}(t_4) = \exists \#x.\{\#v \ \texttt{:=} \ \#x\}\big(\#ct(\varphi_1) \leftrightarrow \#ct(\varphi_2)\big)$$
$$\varphi_1 = \langle .. \ \#u = \ \#e; \ ..._{\#pct}\rangle \#p \qquad \varphi_2 = \langle .. \ \textbf{int} \ \#v = \ \#e; \ \#u = \ \#v; \ ..._{\#pct}\rangle \#p.$$

In $\mathfrak{M}(t_4)$, the quantifier and the update are actually quite useless, as the concerned program variable is declared as a local variable within the program block. A more interesting example would thus be t_5, in which a program variable is used as a skolem constant:

$$\texttt{find}\big(\exists \#x.(\#x \doteq \#t) \ \vdash \ \big) \quad \texttt{varcond}\big(\#x \ \texttt{not free in} \ \#t, \#v \ \texttt{new}\big)$$
$$\texttt{replacewith}\big(\#v \doteq \#t \ \vdash \ \big) \quad \texttt{addprogvar}\big(\#v\big)$$

$$\mathfrak{M}(t_5) = \exists \#y.\{\#v \ \texttt{:=} \ \#y\}\big(\exists \#x.(\#x \doteq \#t) \rightarrow \#v \doteq \#t\big).$$

∗

4. Construction of Proof Obligations

4.1. Basic Definitions

In Sect. 3.2 schema variables are introduced, which can be used to describe sets of formulas, terms or programs. This concept is generalised (for formulas) in the following definition:

4.1 *Definition (Schematic Formula):* A *schematic formula* $\mathfrak{P}$ is a recursive set of formulas. An element $\varphi \in \mathfrak{P}$ is called an *instance* of $\mathfrak{P}$. $*$

This rather abstract definition makes it possible to describe numerous kinds of calculus rules by reducing them to schematic formulas that are regarded as axioms. Examples for formulas that will be used and described in more detail in this document are

- The meaning formulas of taclets, as defined in Sect. 3.3.2 (which are formulated using schema variables)
- Formulas that describe the application of object-level substitution operators, e.g. the set $\mathfrak{S} = \{\varphi \leftrightarrow \varphi' \mid \varphi \in \mathit{For}\}$, where φ' is derived from φ by applying a substitution operator occurring within φ (resolving collisions if necessary). These formulas are discussed in Sect. 4.4.3 and 5.1.1
- Similar formulas that define valid operations on update operators, the execution of meta operators, etc. (Sect. 5).

We continue the notion of *derivability* to schematic formulas by considering their elements:

4.2 *Definition (Derivability of Schematic Formulas):* A schematic formula $\mathfrak{P}$ is called *derivable* in the calculus $\mathbf{K}$, written as

$$\vdash_{\mathbf{K}} \ \mathfrak{P}$$

if each element $\varphi \in \mathfrak{P}$ is derivable in $\mathbf{K}$: $\ \vdash_{\mathbf{K}} \ \varphi$. $*$

The most important kind of schematic formula we discuss in this document are formulas containing schema variables to be substituted by syntactic constructs like formulas, different kinds of variables, terms or programs, as defined in Sect. 3.2. In that section, it has already been stated that formulas which contain schema variables describe sets of formulas:

4.3 *Lemma (Schematic Formulas and Schema Variables):* Let $\mathbf{S} = \{\#s_1, \ldots, \#s_k\}$ be a set of schema variables, and $\varphi \in \mathit{Syn}_{\mathrm{SV}}(\mathbf{S})$ (as in Def. 3.3) be a formula, possibly containing schema variables. Then the set

$$\mathfrak{P}(\varphi) := \{\iota(\varphi) \mid \iota \text{ is a valid instantiation for each } \#s_i\}$$

is a schematic formula.[26] $*$

4.4 *Remark:* To get a shorter notation, we will identify the formula φ and the schematic formula $\mathfrak{P}(\varphi)$, and write $\mathfrak{P}$ for both. $*$

[26] One could add the premise that the validity of an instantiation ι has to be decidable for each schema variable $\#s_i$.

4.2. A Hilbert-style Calculus using Schematic Formulas

In this section, we introduce a very simple calculus that is based on axioms given by schematic formulas, as it has already been announced in Sect. 3.3.2. This calculus will be used to represent sequent calculi defined through taclets in a more convenient way: Instead of referring to a specific taclet application mechanism, taclets will first be transformed into their *meaning formulas*. Further considerations can then be performed independently from the exact definition of taclet applications. Beside taclets, other kinds of rules can be represented as axioms in a natural way, too.

To make proofs about sequent calculi easier, in this section we will use the following convention: After having performed l rule applications, the state of a sequent calculus proof can be represented by a finite set $\{G_1, \ldots, G_k\}$ of sequents, which are the goals that have not yet been closed. We treat such a state as a single formula, namely each sequent G_i is regarded as the disjunction G_i^* of its formulas (the formulas of the antecedents are negated), and the whole proof state is represented by the conjunction of these formulas $G_1^*, \ldots, G_k^*$ (note that we do not allow the formulas to contain free variables). A closed sequent calculus proof of a formula φ is then given by a list of formulas:

$$\varphi, \xi_1, \ldots, \xi_{n-1}, \xi_n = \mathit{true}$$

where each ξ_i is derived from ξ_{i-1} by applying a rule. Thus a sequent calculus rule r can be seen as a description how to obtain formulas ξ_i from ξ_{i-1}.

The basis of the calculus we introduce is an implicit handling of propositional transformations. This is enabled by the well-known fact that the propositional satisfiability problem for finite sets of formulas is decidable. First we need two basic definitions:

4.5 *Definition (JavaCardDL as a Propositional Logic):* We call a JavaCardDL formula φ a *propositional atom*, if the top-level operator of φ is *not* a propositional junctor. Two propositional atoms φ, ψ are regarded as equal, if they are equal modulo bound renaming: $\varphi =_{\mathrm{br}} \psi$. An arbitrary JavaCardDL formula φ can then be regarded as a propositional formula, consisting of junctors and the maximal propositional atoms contained by φ. ∗

4.6 *Example:* As an example, we consider the following formula, in which maximal atoms are shaded:

$$\big(\forall x.p(x) \land \langle\alpha\rangle q\big) \rightarrow \big(q \lor \forall y.p(y)\big)$$

Thereby the atoms $\forall x.p(x)$ and $\forall y.p(y)$ are regarded as equal, and thus the propositional structure of the formula is $(A \land B) \rightarrow (C \lor A)$. ∗

4.7 *Definition (Propositional Reasoning):* Based on Def. 4.5, the following notions can be introduced straightforward:

- A set $\mathbf{F}$ of JavaCardDL formulas is called *propositionally unsatisfiable*, if there is no complete replacement of the occurring maximal propositional atoms with *true* or *false*, such that equal atoms are replaced with the same value, and such that all formulas $\varphi \in \mathbf{F}$ become true.

- *Propositional derivation* is written and defined as

$$\mathbf{F} \vDash_0 \varphi \quad :\Longleftrightarrow \quad \mathbf{F} \cup \{\neg\varphi\} \text{ is propositionally unsatisfiable}^{27}$$

  where $\mathbf{F}$ is a set of JavaCardDL formulas, and φ is a single JavaCardDL formula.

- *Propositional equivalence* of two JavaCardDL formulas φ, ψ is defined as

$$\varphi \equiv_0 \psi \quad :\Longleftrightarrow \quad \vDash_0 \varphi \leftrightarrow \psi. \qquad *$$

The abstract way we defined schematic formulas (which are to be used as axioms) implies that changes made to a vocabulary of a logic in most cases require further changes to be made to the schematic formulas (treated as sets of formulas). To avoid this, we will assume that our vocabularies always contain sufficient sets of skolem symbols we are eventually going to introduce. These symbols are (in most cases) just normal predicate or function symbols, that we however require to be "sufficiently unused". By "unused" we mean that no rule or axiom explicitly distinguishes certain representatives of the symbols, instead we require that it is always possible to exchange occurring symbols. The term "skolem symbol" is in the following pages avoided, as we will later define a family of symbols that is exclusively nominated that way.

4.8 Definition (Unused Symbols): A set of *unused symbols* regarding a set $\mathbf{A}$ of schematic formulas is an infinite set $\mathbf{Sk}$ of symbols of a specific kind (e.g. constants of a given sort, functions with a given signature, etc.), that satisfies the following condition: For $c, d \in \mathbf{Sk}$ and $\psi \in \mathfrak{P} \in \mathbf{A}$ containing c but not d, the formula obtained from ψ by replacing c with d must also be an element of $\mathfrak{P}$. *

Consequently, for the following definition we assume to have a set $\mathbf{C}_{(\mathrm{Ex})}$ of infinitely many unused constant symbols for each sort of *Sort*.

4.9 Definition (Hilbert-style Taclet Calculus): For a given set $\mathbf{A}$ of schematic formulas, such that $\mathbf{C}_{(\mathrm{Ex})}$ is by Def. 4.8 a set of unused symbols regarding $\mathbf{A}$, the calculus $\mathbf{HT} = \mathbf{HT_A}$ is defined in the following way:

1. An $\mathbf{HT}$-proof of a formula φ is a list of formulas

$$\neg\varphi, \psi_1, \ldots, \psi_k$$

   where each of the formulas ψ_i has been introduced by one rule application

2. The rules available in $\mathbf{HT}$ are

$$\frac{}{\psi}\ (\mathrm{Ax}) \qquad\qquad \frac{}{\exists x.\zeta \leftrightarrow \{x/c\}\zeta}\ (\mathrm{Ex})$$

   where $\psi \in \mathfrak{P} \in \mathbf{A}$ is instance of a schematic formula and $c \in \mathbf{C}_{(\mathrm{Ex})}$ is an unused constant symbol, that has the same sort as x. c must neither occur within the proof so far, nor in ζ, and ζ must not contain free variables except x

[27] In this document, the formulas φ for which we apply the definition are always closed.

3. An **HT**-proof is closed if the set $\{\neg\varphi, \psi_1, \dots, \psi_k\}$ is propositionally unsatisfiable, or equivalently if φ is entailed propositionally by $\{\psi_1, \dots, \psi_k\}$:

$$\{\psi_1, \dots, \psi_k\} \models_0 \varphi. \qquad *$$

The rule (Ex) is needed for the introduction of new symbols, as it is not possible to formulate a skolemisation-rule by using schematic formulas only (a particular instance of a schematic formula can always be introduced more than once within a single proof). Note that it is possible to rename a constant $e \in \mathbf{C}_{(\mathrm{Ex})} \backslash \{c\}$, occurring in a formula

$$\exists x.\zeta \leftrightarrow \{x/c\}\zeta$$

introduced by the rule (Ex) by any other symbol $d \in \mathbf{C}_{(\mathrm{Ex})} \backslash \{c, e\}$, provided that d does also not occur in the proof or in ζ, without invalidating the application of (Ex). Together with Def. 4.8 (of unused symbols) it is therefore always possible to rename symbols $c \in \mathbf{C}_{(\mathrm{Ex})}$ that have been introduced by the rule (Ex) by new symbols $d \in \mathbf{C}_{(\mathrm{Ex})}$ within a whole proof H, without making rule applications invalid or a closed proof open.

4.10 *Example (**HT**-proof):* We demonstrate a simple **HT**-proof, using the axioms

$$\mathbf{A} = \{\mathfrak{M}_{(\mathrm{all\_left})}, \mathfrak{S}\}$$

where $\mathfrak{S}$ is the substitution formula, defined in the beginning of Sect. 4.1, and $\mathfrak{M}_{(\mathrm{all\_left})}$ is the meaning formula of the rule (all_left)

$$\mathfrak{M}_{(\mathrm{all\_left})} = \forall \#x.\#p \rightarrow \{\#x \quad \#t\}\#p.$$

The formula to be proved is

$$\varphi = \forall x.p(x) \rightarrow \exists x.p(x)$$

and the proof H_φ could be:

$$\begin{array}{ll}
\neg(\forall x.p(x) \rightarrow \exists x.p(x)), & \neg\varphi \\
\exists x.p(x) \leftrightarrow p(c), & (\mathrm{Ex}) \\
\forall x.p(x) \rightarrow \{x \quad c\}p(x), & \mathfrak{M}_{(\mathrm{all_left})} \\
\{x \quad c\}p(x) \leftrightarrow p(c) & \mathfrak{S}
\end{array}$$

The proof is closed, as the formulas of the left column form a propositionally unsatisfiable set. *

For a given sequent calculus **Ta** having the rules $\mathbf{A}'$, we will construct a set $\mathbf{A}$ of schematic formulas to be used with the calculus **HT**. This will be done by defining a *meaning formula* for each sequent calculus rule r, with the intention to make the complete set of meaning formulas equivalent to the original set of rules:

4.11 *Definition (Axiomatisation of Sequent Calculus Rules):* Let $\mathbf{A}'$ be the rules of the sequent calculus, $\mathbf{A}$ a set of schematic formulas. $\mathbf{A}$ is said to be *equivalent* to $\mathbf{A}'$, if

1. For all instances $\varphi \in \mathfrak{P} \in \mathbf{A}$: $\vdash_{\mathbf{Ta}_{\mathbf{A}'}} \varphi$
2. For each sequent rule application, deriving ξ' from ξ (in the sequent calculus representation from above): There is an (not necessarily closed) $\mathbf{HT_A}$-proof

$$\neg\xi, \psi_1, \ldots, \psi_k$$

such that $\neg\xi'$ can be derived propositionally (according to Def. 4.7):

$$\{\neg\xi, \psi_1, \ldots, \psi_k\} \vDash_0 \neg\xi'.$$

*

This definition can also be seen as the definition of correct rule application mechanisms; namely when providing a method to construct the set $\mathbf{A}$ from a set of rules $\mathbf{A}'$ (as it is done in Sect. 3.3.2 for taclets by the derivation of meaning formulas), the propositions of Def. 4.11 can be postulated and determine whether a mechanism applying the rules $\mathbf{A}'$ is correct.

4.12 *Example:* To illustrate item 2 of Def. 4.11, we consider the rule application

$$\frac{\vdash\ p(c)}{\vdash\ \forall x.p(x)}\ \text{(all\_right)}$$

i.e. we have $\xi = \forall x.p(x)$ and $\xi' = p(c)$. The axioms are now $\mathbf{A} = \{\mathfrak{M}_{(\text{all\_left})}, \mathfrak{S}\}$, where the meaning formula of (all_right) looks like

$$\mathfrak{M}_{(\text{all\_right})} = \exists \#y.(\{\#x \ \ \#y\}\#p \rightarrow \forall \#x.\#p)$$

and the $\mathbf{HT}$-proof would be established by using $\mathfrak{M}_{(\text{all\_right})}$, (Ex) and by applying the substitution formula $\mathfrak{S}$:

$\neg\forall x.p(x)$,	
$\exists y.(\{x \ \ y\}p(x) \rightarrow \forall x.p(x))$,	$\mathfrak{M}_{(\text{all\_right})}$
$\exists y.(\{x \ \ y\}p(x) \rightarrow \forall x.p(x)) \leftrightarrow (\{x \ \ c\}p(x) \rightarrow \forall x.p(x))$,	(Ex)
$\{x \ \ c\}p(x) \leftrightarrow p(c)$	$\mathfrak{S}$

For the set $\mathbf{F}$ of the formulas in the left column we obviously have

$$\mathbf{F} \vDash_0 \neg p(c).$$

*

If the rules $\mathbf{A}'$ of a sequent calculus $\mathbf{Ta}$ are represented by an equivalent set of schematic formulas $\mathbf{A}$, then the calculus $\mathbf{HT_A}$ is also equivalent to the sequent calculus $\mathbf{Ta}_{\mathbf{A}'}$:

4.13 *Lemma:* Given that

- $\mathbf{A}'$ and $\mathbf{A}$ are equivalent
- $\mathbf{Ta}_{\mathbf{A}'}$ provides the cut-rule
- $\mathbf{Ta}_{\mathbf{A}'}$ is propositionally complete (following Def. 4.5)

- $\mathbf{Ta}_{\mathbf{A}'}$ provides a rule equivalent to (Ex)

$$\frac{\Gamma, \exists x.\zeta \leftrightarrow \{x/c\}\zeta \ \vdash \ \Delta}{\Gamma \ \vdash \ \Delta} \text{ (Ex')} \qquad c \in \mathbf{C}_{(\text{Ex})} \text{ new}$$

then $\mathbf{HT}_{\mathbf{A}}$ and $\mathbf{Ta}_{\mathbf{A}'}$ are able to prove the same formulas:

$$\vdash_{\mathbf{Ta}_{\mathbf{A}'}} \ \varphi \Longleftrightarrow \ \vdash_{\mathbf{HT}_{\mathbf{A}}} \ \varphi$$ *

Before we prove Lem. 4.13, we will formulate another, very simple one that will be useful eventually:

4.14 *Lemma:* Let $\mathbf{E}$ be a propositionally unsatisfiable set of formulas with $\varphi \in \mathbf{E}$, and $\mathbf{F}$ a second set of formulas, such that

$$\mathbf{F} \ \vDash_0 \ \varphi.$$

Then the set $\mathbf{E}\backslash\{\varphi\} \cup \mathbf{F}$ is propositionally unsatisfiable. *

Proof (4.13): "$\Longrightarrow$" We show that every $\mathbf{Ta}$-proof of a formula φ can be translated into an $\mathbf{HT}$-proof by induction over the length n of the $\mathbf{Ta}$-proof

$$\varphi, \xi_1, \ldots, \xi_{n-1}, \xi_n = \mathit{true}$$

- $n = 0$: $\varphi = \mathit{true}$, therefore the set $\{\neg \mathit{true}\}$ is unsatisfiable
- $n > 0$: We assume to have a closed $\mathbf{HT}$-proof

  $$\neg\xi_1, \psi_1, \ldots, \psi_k$$

  of ξ_1. From Def. 4.11, it follows that there is an $\mathbf{HT}$-proof

  $$\neg\varphi, \psi_1', \ldots, \psi_l'$$

  satisfying

  $$\{\neg\varphi, \psi_1', \ldots, \psi_l'\} \ \vDash_0 \ \neg\xi_1.$$

  Without loss of generality, applications of (Ex) within $\psi_1, \ldots, \psi_k$ introduce only symbols that do not occur in $\{\neg\varphi, \psi_1', \ldots, \psi_l'\}$ (renaming is always possible). Then

  $$\neg\varphi, \psi_1', \ldots, \psi_l', \psi_1, \ldots, \psi_k$$

  is a closed $\mathbf{HT}$-proof (by Lem. 4.14).

"$\Longleftarrow$" We construct a $\mathbf{Ta}$-proof of φ from the $\mathbf{HT}$-proof

$$\neg\varphi, \psi_1, \ldots, \psi_k$$

by starting with the sequent $\vdash \varphi$ and adding the formulas $\psi_1, \ldots, \psi_k$ to the antecedent:

- If ψ_i has been introduced by the (Ax) rule, we use the cut-rule and are able to close one of the arising sequents by Def. 4.11, 1

- If

$$\psi_i = \exists x.A \leftrightarrow \{x/c\}A$$

has been introduced by the (Ex) rule, we use the rule (Ex') of **Ta** (and replace the original constant c within $\psi_{i+1}, \ldots, \psi_k$ by the new one)

The resulting sequent $\psi_1, \ldots, \psi_k \vdash \varphi$ can be closed by propositional rules. □

The rule (Ex') that was used in the lemma is not a rule sequent calculi usually provide, but can be simulated up to propositional transformations with standard rules (the symbols Γ, Δ are omitted, and we use $\zeta' := \{x/c\}\zeta$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\zeta', \exists x.\zeta \vdash \qquad \begin{array}{c} * \\ \vdots \\ \zeta' \vdash \exists x.\zeta \end{array}}{\zeta' \vdash}\,(\text{cut})}{\exists x.\zeta \vdash}\,(\delta) \qquad \dfrac{\begin{array}{c} * \\ \vdots \\ \zeta' \vdash \exists x.\zeta \end{array} \qquad \vdash \exists x.\zeta, \zeta'}{\vdash \exists x.\zeta}\,(\text{cut})}{\exists x.\zeta \to \exists x.\zeta \vdash}\,(\beta) \qquad \begin{array}{c} * \\ \vdots \\ \vdash \exists x.\zeta \to \exists x.\zeta \end{array}}{\vdash}\,(\text{cut})$$

4.3. Introduction of Lemma Rules

In Sect. 3.3.2 we have defined a map $\mathfrak{M}$ that reduces taclets to axioms, which are schematic formulas in terms of Sect. 4.1. In the previous section, this reduction was continued by introducing an appropriate calculus that employs these axioms, and that is equivalent to a sequent calculus defined through the corresponding taclets.

We assume that we are given a set $\mathbf{A}'$ of sequent calculus rules, and a further rule (r) that is supposed (and to be shown) to be a lemma which can be derived from $\mathbf{A}'$. As (r) is a calculus rule, this means that every formula ξ that can be proved by the rules $\mathbf{A}' \cup \{(\mathrm{r})\}$ can even be proved by $\mathbf{A}'$:

$$\vdash_{\mathbf{Ta}_{\mathbf{A}' \cup \{(\mathrm{r})\}}} \xi \implies \vdash_{\mathbf{Ta}_{\mathbf{A}'}} \xi.$$

A way to show this implication is to translate both $\mathbf{A}'$ and (r) to schematic formulas $\mathbf{A}$, $\mathfrak{M}_{(\mathrm{r})}$, and then prove $\mathfrak{M}_{(\mathrm{r})}$ in the calculus $\mathbf{HT_A}$ (by Def. 4.2, this means that each instance of $\mathfrak{M}_{(\mathrm{r})}$ has to be proved). Subsequently, it is possible to apply Lem. 4.14, and eliminate the axiom $\mathfrak{M}_{(\mathrm{r})}$ from $\mathbf{HT}$-proofs

$$\vdash_{\mathbf{HT}_{\mathbf{A} \cup \{\mathfrak{M}_{(\mathrm{r})}\}}} \xi \implies \vdash_{\mathbf{HT_A}} \xi$$

because instances of $\mathfrak{M}_{(\mathrm{r})}$ in the proof of ξ can be replaced with their $\mathbf{HT_A}$-proofs. The whole procedure is illustrated in the left part of figure 2.

In the remaining part of Sect. 4 we discuss how to prove axioms representing sequent calculi rules, i.e. how to provide the premise $\vdash_{\mathbf{HT_A}} \mathfrak{M}_{(\mathrm{r})}$ for Lem. 4.14 (the right part of the diagram).

$$\begin{tikzcd}
 & & \text{Rule (r)} \arrow[d, "\text{Sect. 3.3.2}"] & \\
\vdash_{\mathbf{Ta}_{\mathbf{A}'\cup\{(\mathrm{r})\}}} \xi \arrow[r, "\text{Lem. 4.13}"] \arrow[d, dashed] & \vdash_{\mathbf{HT}_{\mathbf{A}\cup\{\mathfrak{M}_{(\mathrm{r})}\}}} \xi \arrow[d] & \text{Axiom } \mathfrak{M}_{(\mathrm{r})} \arrow[r] \arrow[d, dashed] & \text{Proof obl. } \varphi_{\mathrm{po}} \arrow[d] \\
\vdash_{\mathbf{Ta}_{\mathbf{A}'}} \xi & \vdash_{\mathbf{HT}_{\mathbf{A}}} \xi \arrow[l, "\text{Lem. 4.13}"'] & \vdash_{\mathbf{HT}_{\mathbf{A}}} \mathfrak{M}_{(\mathrm{r})} \arrow[l, "\text{Lem. 4.14}"'] & \vdash_{\mathbf{HT}_{\mathbf{A}}} \varphi_{\mathrm{po}} \arrow[l, "\text{Sect. 4.4, 4.6}"'] \arrow[d, leftrightarrow, "\text{Lem. 4.13}"'] \\
 & & & \vdash_{\mathbf{Ta}_{\mathbf{A}'}} \varphi_{\mathrm{po}}
\end{tikzcd}$$

Figure 2: The proof of a lemma rule (r) for a sequent calculus by deriving it from existing rules $\mathbf{A}'$. This diagram can be seen as a refinement of figure 1.

4.4. Proving Schematic Formulas for First-Order Logic

To show the derivability of the meaning formula $\mathfrak{P} := \mathfrak{M}(t)$ of a taclet, which is a formula that contains schema variables, we have to derive each instance of $\mathfrak{P}$ (following Def. 4.2):

$$\vdash_{\mathbf{K}} \mathfrak{P} \quad \Longleftrightarrow \quad \text{for all } \varphi \in \mathfrak{P}: \ \vdash_{\mathbf{K}} \varphi.$$

This issue is discussed for first-order logic in this section, and the reasoning will be extended to dynamic logic later. The proof obligations we construct for first-order formulas are essentially identical to the proof obligations described in [Hab00], but we are using a different approach to show the correctness of the method.

For first-order logic, we allow the schematic formula $\mathfrak{P}$ to contain schema variables of the types

- VariableSV
- TermSV
- FormulaSV

as defined in Sect. 3.2.1 (in particular we will not consider the supplementary properties for some of the schema variables, introduced in Sect. 3.2.2). We do neither consider modalities and program variables, nor treat ContextSV explicitly at the time, i.e. we forbid $\mathfrak{P}$ to contain any. Further we assume Rem. 3.11 to hold for $\mathfrak{P}$, and, to make life a bit easier, we forbid the occurrence of any explicit logical variables within $\mathfrak{P}$ (which is no real restriction, as it is always possible to replace variables by VariableSV).

The important steps of the method, as seen by bird's-eye (and in figure 2) are

1. We instantiate $\mathfrak{P}$ with skolem constants, functions and predicates (which, for FOL, are just normal symbols and syntactic elements; for handling dynamic logic, however, we will have to introduce special kinds of skolem symbols, behaving in a well-defined way, especially when applying updates). The resulting formula $\varphi_{\mathrm{po}} \in \mathfrak{P}$ is called the *proof obligation* of the taclet (or of the schematic formula $\mathfrak{P}$).

2. The formula φ_{po} is to be proved (which is not our task). We are assuming that we are given an **HT**-proof of φ_{po}, but if the calculus **HT** is made a sound and complete calculus for FOL by choosing an appropriate set **A** of axioms, obviously a proof may be constructed using an arbitrary calculus. Also notice that Lem. 4.13 can be used, and that it will usually be sufficient to prove φ_{po} within the sequent calculus **Ta**.

3. Finally, the derivability of $\mathfrak{P}$ is shown by lifting the proof of φ_{po} from the second step, i.e. by showing that for each instance $\varphi \in \mathfrak{P}$ we are able to modify the proof, gaining a closed proof of φ. This translation depends on the rules and axioms **HT** provides, whose applications need to be modified without making them invalid; for FOL we will therefore assume that **A** does only contain two classes of rules, namely taclets (or, more generally, schematic formulas defined using schema variables) and the substitution formula $\mathfrak{S}$ (actually we will use a slightly different formula, which is introduced in the next section).

4.4.1. Definition of the Proof Obligation

For FOL and the mentioned three types of schema variables, the formula φ_{po} is obtained from $\mathfrak{P}$ by a straight-forward instantiation of the contained schema variables using skolem functions and predicates. For that, we require the considered vocabularies to contain both skolem functions and skolem predicates for every possible signature, following Def. 4.8.

Def. 4.8 formulates a condition on the set **A** of schematic formulas that will later be used to prove the proof obligation. Hence this set **A** has to be fixed already at this point, and in particular before the skolem symbols are chosen. This means that the proof obligation of a schematic formula depends on the rules available to prove it; this fact will become much more important for richer logics (namely dynamic logic), but also for FOL the formula φ_{po} we are about to define is a sufficient obligation only for a well-defined set of rules (which are, roughly spoken, all axioms that can be defined using the FOL schema variables from Sect. 3.2, and substitution; a more precise definition is given in 4.4.2).

φ_{po} is derived from $\mathfrak{P}$ by performing the following instantiations:

- VariableSV $\#v$ are instantiated with arbitrary distinct logical variables that have the same sort as $\#v$
- TermSV $\#t$ are instantiated with terms $f_{\text{Sk}}(x_1, \ldots, x_k)$, where
  - $x_1, \ldots, x_k$ are the instantiations of VariableSV within the prefix of $\#t$, which are exactly the logical variables that can occur free in instantiations of $\#t$ by the definition of VariableSV (note that the prefix only contains VariableSV)
  - f_{Sk} is a skolem function symbol that does not yet occur within $\mathfrak{P}$ or as instantiation of any schema variable, and that has the signature $(S_1, \ldots, S_k) \to S$
  - $S_1, \ldots, S_k$ are the sorts of $x_1, \ldots, x_k$
  - S is the sort of $\#t$

- FormulaSV $\#p$ are instantiated with formulas $p_{\mathrm{Sk}}(x_1, \ldots, x_k)$ which are chosen analogously to the instantiations of TermSV.

The chosen instantiations satisfy the schema variable definitions of 3.2, thus $\varphi_{\mathrm{po}} \in \mathfrak{P}$.

4.4.2. Lifting Proofs

In this section we show that the implication

$$\vdash_{\mathbf{HT_A}} \varphi_{\mathrm{po}} \Longrightarrow \vdash_{\mathbf{HT_A}} \mathfrak{P} \tag{4}$$

holds for schematic formulas $\mathfrak{P}$ (as defined in the beginning of Sect. 4.4) and the particular proof obligation φ_{po} of $\mathfrak{P}$, which was constructed in 4.4.1. The set $\mathbf{A}$ is defined to consist of two kinds of schematic formulas

- Formulas $\mathfrak{Q}$, which are given by formulas that may contain schema variables of the types VariableSV, TermSV, FormulaSV and ContextSV, as defined in Sect. 3.2. We assume that Rem. 3.11 holds, that $\mathfrak{Q}$ does not contain explicit logical variables (see Rem. 3.8), and that for each ContextSV the unique-property is not set. These formulas correspond to a set of taclets specifying a sequent calculus $\mathbf{Ta}$
- The substitution formula $\mathfrak{S}_{\mathfrak{m}}$, as defined in 4.4.3.

We are supposing the premise of implication (4), i.e. that the proof obligation φ_{po} has been proved using the calculus $\mathbf{HT_A}$: The (closed) proof H shall be given by the formulas

$$\neg\varphi_{\mathrm{po}}, \psi_1, \ldots, \psi_k.$$

We have to show that not only φ_{po}, but every instance $\varphi \in \mathfrak{P}$ can be proved. For that, in each formula of H we replace each skolem symbol introduced in 4.4.1 by the "real" instantiation of the same schema variable within φ, thus getting a modified proof H'

$$\neg\varphi'_{\mathrm{po}}, \psi'_1, \ldots, \psi'_k.$$

We claim that H' already is a proof of φ, i.e. that φ'_{po} and φ are equal modulo bound renaming, and that the new "proof" is still valid, which means:

1. Each modified ψ'_i can be introduced by a valid rule application. It would also be sufficient to show that ψ'_i is derivable (then we are able to apply Lem. 4.14 and replace ψ'_i with its $\mathbf{HT_A}$-proof). For formulas created by the rule (Ex) in the original proof H, this alternative condition does usually not hold.
2. The modified proof is still closed.

First, we describe the modifications leading to H':
Without loss of generality, the following assumptions are made:

- VariableSV are instantiated with the same variables both in φ_{po} and φ (we are always able to rename bound variables in φ_{po} or φ because of Def. 4.5)

- Constants $c \in \mathbf{C}_{(\mathrm{Ex})}$ introduced by (Ex) in H, and skolem functions or predicates that are used to create φ_{po} do not occur in φ (because of Def. 4.8 it is always possible to replace those symbols by unused ones).

Now, in each formula $\xi \in \{\neg\varphi_{\mathrm{po}}, \psi_1, \ldots, \psi_k\}$ we replace skolem symbols f_{Sk} and p_{Sk} from 4.4.1. This is performed using a f-substitution σ (as defined in Sect. A.1). Contrary to normal substitutions, f-substitutions do not replace logical variables, but function and predicate symbols (which do not have to be nullary) with terms and formulas respectively.

- σ_{cont} (as in Def. A.4) shall be given by

  $$\sigma_{\mathrm{cont}}(f_{\mathrm{Sk}}) = (s, \langle x_1, \ldots, x_k \rangle), \qquad \sigma_{\mathrm{cont}}(p_{\mathrm{Sk}}) = (\zeta, \langle y_1, \ldots, y_l \rangle)$$

  for each TermSV $\#t$ instantiated with $f_{\mathrm{Sk}}(x_1, \ldots, x_k)$ in φ_{po}, and with the term s in φ; and for each FormulaSV $\#p$ instantiated with $p_{\mathrm{Sk}}(y_1, \ldots, y_l)$ in φ_{po}, and with the formula ζ in φ. This means that σ replaces f_{Sk} (resp. p_{Sk}) with the term s (resp. the formula ζ), whereby the logical variables $(x_1, \ldots, x_k)$ (resp. $(y_1, \ldots, y_l)$) are treated as formal parameters within the replacements.

- There is an f-substitution σ corresponding to σ_{cont}, as the conditions of Def. A.4 regarding sorts are fulfilled:

  - The construction in 4.4.1 immediately entails that the argument sorts of a skolem symbol s_{Sk} are compatible with the variables $(x_1, \ldots, x_k)$.
  - For TermSV $\#t$: The instantiation s of $\#t$ in φ has the same sort as $\#t$ (this follows from the schema variable definitions in 3.2), which is also the sort of the skolem symbol f_{Sk}.

- As the instantiations of TermSV and FormulaSV may only contain free variables that are allowed by the prefix of the schema variable, Lem. A.8 holds for σ, i.e. applications of σ do not introduce free logical variables.

- Depending on the way ξ had been introduced in H, we obtain an appropriate f-substitution σ_ξ from σ by renaming bound variables, that can at least b applied to ξ without collisions:

  - If ξ results from an application of the rule (Ax), we use the f-substitution $\omega =: \sigma_\xi$ Lem. A.18 (about the application of f-substitutions to instances of schematic formulas) provides; namely, because of Def. 4.8, skolem symbols s_{Sk} must not occur in $\mathfrak{Q}$, and insertion points $a_{\#ct}$ of ContextSV $\#ct$ cannot occur in σ either by Rem. 3.5 (also see Sect. 4.4.3 for $\mathfrak{S}_{\mathfrak{m}}$).
  - Otherwise, we use the f-substitution $\omega =: \sigma_\xi$ given by Lem. A.12, which can be applied without collisions to ξ.

The resulting formulas $\xi' = \sigma_\xi(\xi)$ are the formulas of the proof H'. Note that the substitution σ itself does not depend on ξ, and that all substitutions σ_ξ are equal modulo bound renaming.

The names we did just introduce will also be used within the following paragraphs.

We get a proof for the right formula, i.e. $\varphi'_{\mathbf{po}} =_{\mathbf{br}} \varphi$:
We show that even σ can be applied to φ_{po} without collisions (by considering Def. A.10 of collisions):

- We have already seen that Lem. A.8 holds for σ, therefore the instantiations of TermSV or FormulaSV in φ do not contain additional free variables.

- Skolem symbols occur within φ_{po} only as $s_{\text{Sk}}(x_1, \ldots, x_k)$, thus

$$\begin{aligned} \sigma(s_{\text{Sk}}(x_1, \ldots, x_k)) &= \{x_1/x_1, \ldots, x_k/x_k\}T = T, \\ \text{with} \quad \sigma_{\text{cont}}(s_{\text{Sk}}) &= (T, \langle x_1, \ldots, x_k \rangle) \end{aligned} \tag{5}$$

  cannot lead to any collisions.

As σ and $\sigma_{\neg\varphi_{\text{po}}}$ are equal modulo bound renaming, so are $\sigma(\varphi_{\text{po}})$ and $\sigma_{\neg\varphi_{\text{po}}}(\varphi_{\text{po}}) = \varphi'_{\text{po}}$ (Lem. A.14). Furthermore we have $\sigma(\varphi_{\text{po}}) = \varphi$ because, following equation (5), σ replaces every instantiation $s_{\text{Sk}}(x_1, \ldots, x_k)$ of $\#sv$ in φ_{po} by the corresponding instantiation T in φ (and instantiations of VariableSV are equal by assumption).

The modified formulas ψ'_i can be introduced by valid rule applications:
There are three different cases, discriminated by the way ψ_i is introduced in the original proof H:

- The rule (Ax) is used, together with a schematic formula $\mathfrak{Q}$ that is defined using schema variables. Then Lem. A.18 tells us that ψ'_i can also be introduced, again using the rule (Ax) and $\mathfrak{Q}$.

- The rule (Ax) is used, together with the substitution formula $\mathfrak{S}_{\mathfrak{m}}$. In Sect. 4.4.3 we prove that $\sigma_{\psi_i}(\psi_i) \in \mathfrak{S}_{\mathfrak{m}}$.

- The rule (Ex) is used, and

$$\psi_i = \exists x.\alpha \leftrightarrow \{x/c\}\alpha, \quad \text{with} \quad c \in \mathbf{C}_{(\text{Ex})},\ \alpha \in \mathit{For}.$$

  Then we have to show:

  - The constant c does not occur within $\{\neg\varphi'_{\text{po}}, \psi'_1, \ldots, \psi'_{i-1}\}$: By assumption we have $c \notin FS(\varphi)$ (thus $c \notin FS(\varphi'_{\text{po}})$), and therefore c is not introduced by σ either: $c \notin \text{Cod}(\sigma)$ and $c \notin \text{Cod}(\sigma_{\psi_j})$. Because c does not turn up in the formulas $\psi_1, \ldots, \psi_{i-1}$ of the original proof by the definition of the rule (Ex), this entails $c \notin FS(\psi'_j) = FS(\sigma_{\psi_j}(\psi_j))$
  - ψ'_i can also be created by (Ex), i.e. $\psi'_i = \exists x.\zeta \leftrightarrow \{x/c\}\zeta$ for a suitable formula ζ that does not contain c:

$$\begin{aligned} \psi'_i &= \sigma_{\psi_i}(\psi_i) = \sigma_{\psi_i}(\exists x.\alpha \leftrightarrow \{x/c\}\alpha) \\ &= \exists x.\sigma_{\psi_i}(\alpha) \leftrightarrow \sigma_{\psi_i}(\{x/c\}\alpha) \overset{(*)}{=} \exists x.\sigma_{\psi_i}(\alpha) \leftrightarrow \{x/\sigma_{\psi_i}(c)\}\sigma_{\psi_i}(\alpha) \\ &= \exists x.\sigma_{\psi_i}(\alpha) \leftrightarrow \{x/c\}\sigma_{\psi_i}(\alpha) \end{aligned}$$

where $(*)$ uses Lem. A.16 about the concatenation of f-substitutions. The presumptions of this lemma hold, as by construction the application of σ_{ψ_i} to α is collision free, and because neither $\mathrm{Dom}(\sigma_{\psi_i})$ nor $\mathrm{Cod}(\sigma_{\psi_i})$ contain the logical variable x. Furthermore the definition of σ immediately entails $c \notin \mathrm{Dom}(\sigma_{\psi_i})$, which enabled the last transformation.

$\sigma_{\psi_i}(\alpha)$ does not contain c because of $c \notin \mathrm{Cod}(\sigma)$, and because c does not occur in α.

The proof H' is closed:
f-Substitutions do not modify propositional operators, therefore the ξ' arises from ξ by replacing propositional atoms α with $\sigma_\xi(\alpha)$. It is sufficient to show that propositional atoms α, β which are equal in H (which means, they differ only on bound variables) are represented by formulas α', β' in H' that are also equal (modulo bound renaming). This follows directly from Lem. A.14.

4.4.3. About Substitutions

In 4.1, as an example we defined a schematic formula describing the application of (object-level) substitution operators. For comfortable proof lifting, however, we need a formula that is defined a bit different, namely the more general substitution formula $\mathfrak{S}_\mathfrak{m} \supset \mathfrak{S}$, which also contains formulas that describe repeated application of substitution operators (at least for certain situations, namely a sub-term or -formula appearing more than once). This formula will have the nice property of being closed under certain kinds of f-substitutions (if collisions are resolved when applying the f-substitution by renaming bound variables).

To reuse results that are available for schema variables, we define $\mathfrak{S}_\mathfrak{m}$ to be a subset of the schematic formula

$$\mathfrak{R} = \#ct(\#a) \leftrightarrow \#ct(\#b), \qquad \mathrm{prefix}_{\#a} = \mathrm{prefix}_{\#b} = \{\#ct\}^{28}$$

where $\#a$, $\#b$ are either both TermSV or both FormulaSV, and $\#ct$ does have the unique-flag set to false. For $\mathfrak{S}_\mathfrak{m}$, we restrict the instantiations of $\#a$, $\#b$ to $\{\#a/\{x \;\; s\}t\}$, $\{\#b/\{x/s\}t'\}$, with the first substitution being the object-level substitution operator, and the second one the meta-level substitution. We require that collisions which may arise upon application of the meta-level substitution are resolved in $t' =_{\mathrm{br}} t$ by bound renaming.

4.15 *Lemma:* Let $\varphi \in \mathfrak{S}_\mathfrak{m}$ be an instance of $\mathfrak{S}_\mathfrak{m}$, and σ be an f-substitution that satisfies the conditions of Lem. A.18 (about the application of f-substitutions to instances of schematic formulas) regarding the formula $\varphi = \iota(\mathfrak{R})$. Then we have

$$\omega(\varphi) \in \mathfrak{S}_\mathfrak{m}$$

where $\omega =_{\mathrm{br}} \sigma$ is the f-substitution that is obtained from σ by bound renaming. $*$

[28] This means that variables bound in instantiations of $\#ct$ may occur free within instantiations of $\#a$ and $\#b$ (provided that scopes are respected, i.e. instances of $\mathfrak{R}$ do not contain free variables).

Roughly spoken, this means that $\mathfrak{S}_{\mathfrak{m}}$ is closed under (collision free) applications of f-substitutions.

Proof: Suppose that $\varphi = \iota(\mathfrak{R}) \in \mathfrak{S}_{\mathfrak{m}}$ is an instance of $\mathfrak{S}_{\mathfrak{m}}$ with

$$\iota = \{\#ct/\zeta, \#a/\{x \quad s\}t, \#b/\{x/s\}t'\}$$

and σ is an f-substitution satisfying the conditions of Lem. A.18. Then (by Lem. A.18) there is an f-substitution $\omega =_{\mathrm{br}} \sigma$, which can be applied to φ without collisions, such that

$$\omega(\varphi) = \kappa(\mathfrak{R}), \qquad \kappa = \{\#ct/\omega(\zeta), \#a/\omega(\{x \quad s\}t), \#b/\omega(\{x/s\}t')\}$$

i.e. in particular $\omega(\varphi) \in \mathfrak{R}$. We can furthermore assume that the application of ω to t' is collision free by Lem. A.12, because we have $\mathrm{Cod}(\sigma) \cap \mathit{Var} = \emptyset$ as a premise of Lem. A.18.

For the stronger statement $\omega(\varphi) \in \mathfrak{S}_{\mathfrak{m}} \subset \mathfrak{R}$ (which is our goal), we can observe that ω (or equivalently σ) satisfies the conditions of Lem. A.16 regarding the substitution $\{x/s\}$, which provides a way to concatenate the two substitutions. Namely, for f-substitutions in general we have $\mathrm{Dom}(\sigma) \cap \mathit{Var} = \emptyset$, and $\mathrm{Cod}(\sigma) \cap \mathit{Var} = \emptyset$ has already been observed. The application of ω to t' is collision free by assumption. Then we have

$$\omega(\{x/s\}t') = \{x/\omega(s)\}\omega(t')$$

and in this equation the substitution of x on the right side cannot lead to collisions, as the applications of ω and $\{x/s\}$ on the left side cause none by Lem. A.18.

On the other hand, by definition we have

$$\omega(\{x \quad s\}t) = \{x \quad \omega(s)\}\omega(t).$$

and therefore (using Lem. A.14, which provides $\omega(t') =_{\mathrm{br}} \omega(t)$) we have proved the conjecture $\omega(\varphi) \in \mathfrak{S}_{\mathfrak{m}}$. □

To summarise this section, we have found a schematic formula $\mathfrak{S}_{\mathfrak{m}}$ that is closed under f-substitutions ω, as supplied by Lem. A.18, provided that the unique-flag of $\#ct$ is false.

To stress the importance of the unique-flag, in the next example we also include the schematic formula $\mathfrak{S}^{\mathfrak{u}}_{\mathfrak{m}} \subset \mathfrak{S}_{\mathfrak{m}}$, which is defined almost exactly as $\mathfrak{S}_{\mathfrak{m}}$. The only difference between $\mathfrak{S}^{\mathfrak{u}}_{\mathfrak{m}}$ and $\mathfrak{S}_{\mathfrak{m}}$ is that for $\mathfrak{S}^{\mathfrak{u}}_{\mathfrak{m}}$ we require the unique-flag of $\#ct$ to be true instead of false. $\mathfrak{S}^{\mathfrak{u}}_{\mathfrak{m}}$ is closely related to the original substitution formula $\mathfrak{S}$, defined in the beginning of Sect. 4.1: $\mathfrak{S}^{\mathfrak{u}}_{\mathfrak{m}}$ does exactly contain the closed formulas of $\mathfrak{S}$.

4.16 Example: Some formulas, and whether they are contained by the different substitution formulas respectively:

	$\mathfrak{S}$	$\mathfrak{S}_{\mathfrak{m}}$	$\mathfrak{S}_{\mathfrak{m}}^{\mathfrak{u}}$
$\{x\ \ c\}p(x) \leftrightarrow p(c)$	$\times$	$\times$	$\times$
$\{x\ \ y\}\forall y.p(x) \leftrightarrow \forall z.p(y)$	$\times$		
$\forall y.\{x\ \ y\}\forall y.p(x) \leftrightarrow \forall y.\forall z.p(y)$	$\times$	$\times$	$\times$
$(\{x\ \ c\}p(x) \wedge \{x\ \ c\}p(x)) \leftrightarrow (p(c) \wedge p(c))$		$\times$	
$(\{x\ \ c\}p(x) \wedge \{x\ \ c\}q(x)) \leftrightarrow (p(c) \wedge q(c))$			

The formula in the last line is not an instance of $\mathfrak{S}_{\mathfrak{m}}$, because the two substitution operators are applied to different formulas, which is not covered by the definition of $\mathfrak{S}_{\mathfrak{m}}$. *

4.5. An Example

We prove the schematic formula

$$\mathfrak{P} = \forall\#x.\forall\#y.\#p \to \forall\#y.\forall\#x.\#p \qquad \text{prefix}_{\#p} = \{\#x, \#y\}$$

using a standard FOL set $\mathbf{A}'$ of sequent calculus rules, by proving the proof obligation within the sequent calculus $\mathbf{Ta}_{\mathbf{A}'}$. The sequent calculus proof is then translated to $\mathbf{HT}_{\mathbf{A}}$ (where $\mathbf{A}$ and $\mathbf{A}'$ shall be equivalent according to Def. 4.11), and lifted to obtain a proof for an instance of $\mathfrak{P}$.

The proof obligation for $\mathfrak{P}$, as defined in Sect. 4.4.1, is

$$\varphi_{\text{po}} = \iota(\mathfrak{P}) = \forall x.\forall y.p_{\text{Sk}}(x, y) \to \forall y.\forall x.p_{\text{Sk}}(x, y), \quad \iota = \{\#x/x, \#y/y, \#p/p_{\text{Sk}}(x, y)\}$$

and is proved by

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{p_{\text{Sk}}(d,c) \ \vdash\ p_{\text{Sk}}(d,c)}\,(\text{close\_goal})}{\forall y.p_{\text{Sk}}(d,y) \ \vdash\ p_{\text{Sk}}(d,c)}\,(\text{all\_left})}{\forall x.\forall y.p_{\text{Sk}}(x,y) \ \vdash\ p_{\text{Sk}}(d,c)}\,(\text{all\_left})}{\forall x.\forall y.p_{\text{Sk}}(x,y) \ \vdash\ \forall x.p_{\text{Sk}}(x,c)}\,(\text{all\_right})}{\forall x.\forall y.p_{\text{Sk}}(x,y) \ \vdash\ \forall y.\forall x.p_{\text{Sk}}(x,y)}\,(\text{all\_right})}{\vdash\ \forall x.\forall y.p_{\text{Sk}}(x,y) \to \forall y.\forall x.p_{\text{Sk}}(x,y)}\,(\text{imp\_right})$$

This **Ta**-proof is translated to the **HT**-proof H (see table 1) by replacing each rule with an equivalent list of **HT**-rule applications (as in the proof of Lem. 4.13 about the equivalence of the two calculi), which is basically the addition of an equivalent instance of the meaning formula, sometimes however inserting substitution formulas or applying the rule (Ex) (in **Ta**, both are assumed to be part of (all_right)). As **HT** has its own rules for propositional logic, it is not necessary to do anything for propositional sequent calculus rules.

Comparing the **HT**-rule applications for the two applications of (all_right), one can see that different orders have been chosen respectively. While this has mainly been done to make the following lifting process more interesting, and both translations have the same length, there is of course an infinite number of further modifications, as **HT** does not restrict the set of rules that may be applied in a certain situation at all.

Ta-Rule	Meaning Formula, Result	**HT**-Rule
	$\neg(\forall x.\forall y.p_{\mathrm{Sk}}(x,y) \rightarrow \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$	
(imp_right)	$(\#p \rightarrow \#q) \vee (\#p \wedge \neg\#q)$	
	(no rule applications necessary)	
(all_right)	$\exists\#y.(\neg\{\#x\ \ \#y\}\#p \vee \forall\#x.\#p)$	
	$\exists z.(\neg\{y\ \ z\}\forall x.p_{\mathrm{Sk}}(x,y) \vee \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$	(Ax)+MF
	$\exists z.(\neg\{y\ \ z\}\forall x.p_{\mathrm{Sk}}(x,y) \vee \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$	(Ex)
	$\quad\leftrightarrow (\neg\{y\ \ c\}\forall x.p_{\mathrm{Sk}}(x,y) \vee \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$	
	$\{y\ \ c\}\forall x.p_{\mathrm{Sk}}(x,y) \leftrightarrow \forall x.p_{\mathrm{Sk}}(x,c)$	(Ax)+$\mathfrak{S}_{\mathfrak{m}}$
(all_right)	$\exists\#y.(\neg\{\#x\ \ \#y\}\#p \vee \forall\#x.\#p)$	
	$\exists z.(\neg\{x\ \ z\}p_{\mathrm{Sk}}(x,c) \vee \forall x.p_{\mathrm{Sk}}(x,c))$	(Ax)+MF
	$\exists z.(\neg\{x\ \ z\}p_{\mathrm{Sk}}(x,c) \vee \forall x.p_{\mathrm{Sk}}(x,c))$	(Ax)+$\mathfrak{S}_{\mathfrak{m}}$
	$\quad\leftrightarrow \exists z.(\neg p_{\mathrm{Sk}}(z,c) \vee \forall x.p_{\mathrm{Sk}}(x,c))$	
	$\exists z.(\neg p_{\mathrm{Sk}}(z,c) \vee \forall x.p_{\mathrm{Sk}}(x,c))$	(Ex)
	$\quad\leftrightarrow (\neg p_{\mathrm{Sk}}(d,c) \vee \forall x.p_{\mathrm{Sk}}(x,c))$	
(all_left)	$\neg\forall\#x.\#p \vee \{\#x\ \ \#t\}\#p$	
	$\neg\forall x.\forall y.p_{\mathrm{Sk}}(x,y) \vee \{x\ \ d\}\forall y.p_{\mathrm{Sk}}(x,y)$	(Ax)+MF
	$\{x\ \ d\}\forall y.p_{\mathrm{Sk}}(x,y) \leftrightarrow \forall y.p_{\mathrm{Sk}}(d,y)$	(Ax)+$\mathfrak{S}_{\mathfrak{m}}$
(all_left)	$\neg\forall\#x.\#p \vee \{\#x\ \ \#t\}\#p$	
	$\neg\forall y.p_{\mathrm{Sk}}(d,y) \vee \{y\ \ c\}p_{\mathrm{Sk}}(d,y)$	(Ax)+MF
	$\{y\ \ c\}p_{\mathrm{Sk}}(d,y) \leftrightarrow p_{\mathrm{Sk}}(d,c)$	(Ax)+$\mathfrak{S}_{\mathfrak{m}}$
(close_goal)	$\neg\#p \vee \#p$	
	(no rule applications necessary)	

Table 1: The **HT**-proof H, and how it is obtained from the **Ta**-proof. In the left column (going downwards) the rules applied in the **Ta**-proof are enumerated, and the formulas next to the rule names in the middle column are the meaning formulas of these rules, as defined in Sect. 3.3.2. The shaded entries below each meaning formula contain the formulas added to the **HT**-proof to simulate the particular **Ta**-rule application, and next to these formulas in the right column the **HT**-rules used are given.
The topmost formula (of the middle column) is the negated proof obligation, and the list of all shaded formulas of the table represents the complete (and closed) **HT**-proof.

To show how the **HT**-proof of the proof obligation is lifted to a proof of a second instance of $\mathfrak{P}$, we consider the following instantiation

$$\varphi = \kappa(\mathfrak{P}) = \forall x.\forall y.\exists z.q(x,z) \to \forall y.\forall x.\exists z.q(x,z),$$
$$\kappa = \{\#x/x, \#y/y, \#p/\exists z.q(x,z)\}$$

in which the variable z (which already occurs in H) has been inserted to provoke a collision (according to the assumption in Sect. 4.4.2, in φ VariableSV are instantiated with the same logical variables as in φ_{po}).

Proof H of φ_{po}	Proof H' of φ
$\neg(\forall x.\forall y.p_{\mathrm{Sk}}(x,y) \to \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$	$\neg(\forall x.\forall y.\exists z.q(x,z) \to \forall y.\forall x.\exists z.q(x,z))$
$\exists z.(\neg\{y\ \ z\}\forall x.p_{\mathrm{Sk}}(x,y) \lor \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$	$\exists z.(\neg\{y\ \ z\}\forall x.\exists u.q(x,u) \lor \forall y.\forall x.\exists u.q(x,u))$
$\exists z.(\neg\{y\ \ z\}\forall x.p_{\mathrm{Sk}}(x,y) \lor \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$ $\leftrightarrow (\neg\{y\ \ c\}\forall x.p_{\mathrm{Sk}}(x,y) \lor \forall y.\forall x.p_{\mathrm{Sk}}(x,y))$	$\exists z.(\neg\{y\ \ z\}\forall x.\exists z.q(x,z) \lor \forall y.\forall x.\exists z.q(x,z))$ $\leftrightarrow (\neg\{y\ \ c\}\forall x.\exists z.q(x,z) \lor \forall y.\forall x.\exists z.q(x,z))$
$\{y\ \ c\}\forall x.p_{\mathrm{Sk}}(x,y) \leftrightarrow \forall x.p_{\mathrm{Sk}}(x,c)$	$\{y\ \ c\}\forall x.\exists z.q(x,z) \leftrightarrow \forall x.\exists z.q(x,z)$
$\exists z.(\neg\{x\ \ z\}p_{\mathrm{Sk}}(x,c) \lor \forall x.p_{\mathrm{Sk}}(x,c))$	$\exists z.(\neg\{x\ \ z\}\exists v.q(x,v) \lor \forall x.\exists v.q(x,v))$
$\exists z.(\neg\{x\ \ z\}p_{\mathrm{Sk}}(x,c) \lor \forall x.p_{\mathrm{Sk}}(x,c))$ $\leftrightarrow \exists z.(\neg p_{\mathrm{Sk}}(z,c) \lor \forall x.p_{\mathrm{Sk}}(x,c))$	$\exists z.(\neg\{x\ \ z\}\exists w.q(x,w) \lor \forall x.\exists w.q(x,w))$ $\leftrightarrow \exists z.(\neg\exists w.q(z,w) \lor \forall x.\exists w.q(x,w))$
$\exists z.(\neg p_{\mathrm{Sk}}(z,c) \lor \forall x.p_{\mathrm{Sk}}(x,c))$ $\leftrightarrow (\neg p_{\mathrm{Sk}}(d,c) \lor \forall x.p_{\mathrm{Sk}}(x,c))$	$\exists z.(\neg\exists u.q(z,u) \lor \forall x.\exists u.q(x,u))$ $\leftrightarrow (\neg\exists u.q(d,u) \lor \forall x.\exists u.q(x,u))$
$\neg\forall x.\forall y.p_{\mathrm{Sk}}(x,y) \lor \{x\ \ d\}\forall y.p_{\mathrm{Sk}}(x,y)$	$\neg\forall x.\forall y.\exists z.q(x,z) \lor \{x\ \ d\}\forall y.\exists z.q(x,z)$
$\{x\ \ d\}\forall y.p_{\mathrm{Sk}}(x,y) \leftrightarrow \forall y.p_{\mathrm{Sk}}(d,y)$	$\{x\ \ d\}\forall y.\exists z.q(x,z) \leftrightarrow \forall y.\exists z.q(d,z)$
$\neg\forall y.p_{\mathrm{Sk}}(d,y) \lor \{y\ \ c\}p_{\mathrm{Sk}}(d,y)$	$\neg\forall y.\exists z.q(d,z) \lor \{y\ \ c\}\exists z.q(d,z)$
$\{y\ \ c\}p_{\mathrm{Sk}}(d,y) \leftrightarrow p_{\mathrm{Sk}}(d,c)$	$\{y\ \ c\}\exists z.q(d,z) \leftrightarrow \exists z.q(d,z)$

Table 2: The original proof H of the proof obligation, and the lifted one of φ. The skolem formulas of the left side are shaded, and in the right proof they are replaced with the concrete instantiations.

The f-substitution σ defined in Sect. 4.4.2 is then given by

$$\sigma_{\mathrm{cont}}(p_{\mathrm{Sk}}) = (\exists z.q(x,z), \langle x,y\rangle)$$

and the proof resulting from the application of σ (or an f-substitution in which bound variables have been renamed) can be found in table 2. Reasons for renaming the variable z bound in σ (and that also appear in this example) are:

- The definition of schema variables, which does not allow the instantiations of VariableSV to occur bound within TermSV- or FormulaSV-instantiations (this is enforced by Lem. A.18)
- Real collisions that may occur when substituting formal parameters (when applying f-substitutions).

4.6. Proving Schematic Formulas for JavaCardDL

Having seen a strategy to prove a certain kind of schematic formulas for FOL in Sect. 4.4, we are now ready to treat a more complicated logic, namely JavaCardDL.

Therefore we will use the additional schema variables of Sect. 3.2.2, and also introduce some new kinds of rules that are necessary to handle dynamic logic.

Another method to create proof obligations for such kinds of JavaCardDL formulas, which is similar to the method we will introduce, is outlined in [Hab].

The schematic formula $\mathfrak{P}$ that is to be derived may be defined using the schema variable types

- VariableSV, TermSV, FormulaSV, the types used for FOL
- PVariableSV, StatementSV, ExpressionSV and LabelSV, the most important types introduced for JavaCardDL

and the set of axioms we are referring to shall be denoted by $\mathbf{A}$ (in the next section we will define which kinds of axioms $\mathbf{A}$ is allowed to contain).

The remaining schema variables for contexts (within formulas and programs) we will not consider yet, but explain later how to add them to the following method. Furthermore we assume that Rem. 3.11, 3.20 and 3.17 hold, and that $\mathfrak{P}$ does not contain explicit logical variables, explicit bound program variables or explicit labels (Rem. 3.19).

Again the calculus **HT** will be used to manage proofs, and the underlying principle will be the same as in 4.4 (things will however get a bit more complicated, and some details are different):

1. A proof obligation φ_{po} is derived from the schematic formula $\mathfrak{P}$.
2. φ_{po} has to be proved, and we assume that an $\mathbf{HT}_{\mathbf{A}_{\mathrm{Sk}}}$-proof H exists for a set $\mathbf{A}_{\mathrm{Sk}}$ of axioms; in comparison with the basic set $\mathbf{A}$, $\mathbf{A}_{\mathrm{Sk}}$ contains additional formulas determining the nature of skolem symbols that are needed to formulate φ_{po}. Beside that, in $\mathbf{A}_{\mathrm{Sk}}$ the rules of $\mathbf{A}$ are adapted to the richer vocabulary.
3. The proof H is lifted to $\mathbf{HT}_{\mathbf{A}}$-proofs H' of each instance φ of $\mathfrak{P}$, by replacing skolem symbols that have been inserted into φ_{po}, and doing some further modifications. This derived proofs do no longer contain any rules not present in $\mathbf{A}$. More formally, in the third step we show the implication[29]

$$\vdash_{\mathbf{A}_{\mathrm{Sk}}} \varphi_{\mathrm{po}} \Longrightarrow \ \vdash_{\mathbf{A}} \mathfrak{P}. \tag{6}$$

4.17 Example: To motivate the following definitions, we demonstrate the derivation of the proof obligation φ_{po} for a schematic formula $\mathfrak{P}$, which is

$$\mathfrak{P} = \langle \#s; \#v = \#v; \rangle \#p \leftrightarrow \langle \#s; \rangle \#p$$

where the appearing schema variables are

Symbol	Type	javaType	unusedOnly	prefix	pvPrefix	jumpPrefix
$\#s$	StatementSV				$\{\#v\}$	$\emptyset$
$\#v$	PVariableSV	**int**	*false*			
$\#p$	FormulaSV			$\emptyset$	$\{\#v\}$	

[29] If the used calculus is clearly determined by the context, we will write $\vdash_{\mathbf{A}}$ instead of $\vdash_{\mathbf{HT}_{\mathbf{A}}}$.

As for FOL, to construct the proof obligation the schema variables of $\mathfrak{P}$ are replaced with skolem symbols. For FormulaSV and StatementSV, these symbols are now distinguished from the normal vocabulary of JavaCardDL (the definition can be found in the next section). The PVariableSV is instantiated with an ordinary program variable, and altogether the resulting formula looks like

$$\varphi_{\mathrm{po}} = \langle s_{\mathrm{Sk}}(\mathrm{v}_{\mathrm{Sk}}, \mathrm{t}, \mathrm{d}; \mathbf{throw}\ \mathrm{t}); \mathrm{v}_{\mathrm{Sk}} = \mathrm{v}_{\mathrm{Sk}};\rangle p_{\mathrm{Sk}}(\mathrm{v}_{\mathrm{Sk}}) \leftrightarrow \langle s_{\mathrm{Sk}}(\mathrm{v}_{\mathrm{Sk}}, \mathrm{t}, \mathrm{d}; \mathbf{throw}\ \mathrm{t});\rangle p_{\mathrm{Sk}}(\mathrm{v}_{\mathrm{Sk}})$$

and has some more or less astonishing features (apart from being three times longer than the schematic formula):

- As it has been announced, the PVariableSV has become a program variable, which is nothing really exciting.
- The skolem symbols for the schema variables $\#s$ and $\#p$ have got the argument v_{Sk}, which indicates that the variable v_{Sk} can occur within instantiations of $\#s$ and $\#p$ (this follows directly from the values of the pvPrefix-properties). Note, however, that the definition of the pvPrefix nevertheless allows further free program variables beside v_{Sk} to turn up within these instantiations (and also see footnote 14).
- The schema variable $\#s$ has got the very strange argument **throw** t (and the program variable t, which is of type **Throwable**, occurs a second time on its own). This mysterious statement forms the so called "jump table" of the symbol s_{Sk}, which enumerates (exactly) those statements that may lead to an abrupt termination when executing an instantiation of $\#s$. This list of statements corresponds to the jumpPrefix property of StatementSV, and other possible elements of the list are therefore statements like **break**, **continue** and **return**. An example with two jump statements would look like

$$s_{\mathrm{Sk}}(\mathrm{v}_{\mathrm{Sk}}, \mathrm{t}, \mathrm{d}; \mathbf{throw}\ \mathrm{t}; \mathbf{continue}).$$

- The skolem symbol s_{Sk} has a further argument d (which is a program variable of type **int**), that will be used in combination with the "jump table" to control the way of termination.

For the following, the skolem symbols within φ_{po} are however not given possible interpretations; we rather define "continuations" of the available calculus rules, in such a way that it is possible to derive proofs for $\mathfrak{P}$-instances from a proof of φ_{po}. Thus the argument **throw** t has the job to tell every rule that modifies the program block to act "as if" the statement would contain a **throw**-statement.

Within a proof of the proof obligation φ_{po}, performed using the calculus $\mathbf{HT}_{\mathbf{A}_{\mathrm{Sk}}}$, the skolem symbol p_{Sk} is mostly treated like a normal predicate symbol. Analogously, $s_{\mathrm{Sk}}(\mathrm{v}_{\mathrm{Sk}}, \mathrm{t}, \mathrm{d}; \mathbf{throw}\ \mathrm{t})$ is in most situations regarded as a normal statement.

In the following sections, the main reason to furnish symbols with a jump table is the definition of a decomposition rule, which splits the program block of a modal operator in two blocks, whereby the first one contains the leading statement of the original program block, and the second one all further statements. There are, however, other situations in which an explicit enumeration of the jump statements is necessary (but that are not treated in this thesis), e.g. the rules that unwind loops, which also modify **break**- and **continue**-statements of the loop body. $*$

4.6.1. The Axioms $\mathbf{A}$

In the first place, namely in this section, we will define the set $\mathbf{A}$ of basic axioms, which refer only to JavaCardDL. These axioms determine the derivability of the schematic formula $\mathfrak{P}$ that is discussed,

$$\vdash_{\mathbf{HT_A}} \mathfrak{P}.$$

$\mathbf{A}$ does only consist of Formulas $\mathfrak{Q}$ that are defined (possibly) utilising all schema variables of Sect. 3.2.[30] We assume that Rem. 3.11 and 3.20 hold, that $\mathfrak{Q}$ does not contain explicit logical variables, no explicit bound program variables and no explicit labels (see Rem. 3.19), and that for each ContextSV the unique-property is not true.

In Sect. 3.2.2, to define PVariableSV (schema variables for program variables) a set $PVar_u$ of "unused" program variables is introduced. For the following sections, we define this set to be

$$PVar_u := \bigcup_{T \in JavaTypes} \mathbf{PV}_T$$

where each $\mathbf{PV}_T$ is required to be an infinite set of program variables of type T, which satisfies Def. 4.8 regarding the rules $\mathbf{A}$ (i.e. the variables are "unused symbols"), and which is a maximal subset of the used vocabulary with this property.

Conversely, we define $PVar_e := PVar \backslash PVar_u$ to be the set of program variables that in some way occur "explicitly" within an axiom, and that will have to be treated exceptionally. Most prominently, examples for such program variables are variables that explicitly (and freely) occur within a taclet. Almost as bad are program variables occurring (freely) in $\mathfrak{P}$ (the schematic formula we want to prove), if $\mathfrak{P}$ is regarded as a "real" formula containing schema variables (and not as a set of formulas as in Def. 4.1). Thus we denote the set of these variables with $\mathbf{E}$, and define

$$PVar_e(\mathfrak{P}) := \mathbf{E} \cup PVar_e.$$

4.6.2. Skolem Symbols

In contrast to FOL, where we were able to use normal function and predicate symbols for skolemization of TermSV and FormulaSV when formulating the proof obligation, we will now distinguish specials sets of symbols as *skolem symbols*. These symbols are treated in a well-defined way by the rules that are available to prove the formula φ_{po}, and are added to the original vocabulary given. We will denote the extended logic by JavaCardDL$_{\mathrm{Sk}}$, and the set of all term-like constructs (i.e. formulas, terms and programs) of JavaCardDL$_{\mathrm{Sk}}$ by Syn_{Sk}.

To deal with the new symbols, it is also necessary to extend the set $\mathbf{A}$ of axioms to a set $\mathbf{A}_{\mathrm{Sk}}$. $\mathbf{A}_{\mathrm{Sk}}$ consists of continuations $\mathfrak{Q}_{\mathrm{Sk}}$ of the axioms $\mathfrak{Q} \in \mathbf{A}$ (to reflect the

[30]Further kinds of axioms are discussed in Sect. 5.

extended vocabulary of $\text{JavaCardDL}_{\text{Sk}}$) and some additional axioms $\mathbf{C}_{\text{Sk}}$, which are introduced in detail in the next section:

$$\mathbf{A}_{\text{Sk}} = \{\mathfrak{Q}_{\text{Sk}} \supset \mathfrak{Q} \mid \mathfrak{Q} \in \mathbf{A}\} \cup \mathbf{C}_{\text{Sk}}.$$

The skolem symbols we introduce are:

- Sets $\mathit{Func}_{\text{Sk}}$ and $\mathit{Pred}_{\text{Sk}}$ of distinguished function and predicate symbols. Furthermore we define a mapping of the two sets[31]

  $$\mathit{PVArg} : \mathit{Func}_{\text{Sk}} \cup \mathit{Pred}_{\text{Sk}} \rightarrow \mathit{JavaTypes}^*$$

  (where "*PVArg*" stands for "program variable arguments"). We regard *PVArg* as a part of the signature of the symbols $\mathit{Func}_{\text{Sk}} \cup \mathit{Pred}_{\text{Sk}}$, which as a whole is based on the signatures ordinary function and predicate symbols have. Thus the signature of a symbol $f_{\text{Sk}} \in \mathit{Func}_{\text{Sk}}$ consists of

  – a tuple of argument sorts and one result sort (for the set $\mathit{Pred}_{\text{Sk}}$, the result sort is left out)
  – the tuple $\mathit{PVArg}(f_{\text{Sk}})$ of Java types.

  Accordingly, occurrences of a symbol $s_{\text{Sk}} \in \mathit{Func}_{\text{Sk}} \cup \mathit{Pred}_{\text{Sk}}$ are equipped with an additional tuple of arguments. These arguments may only be program variables which have exactly the Java type given by *PVArg*, i.e. in particular not complex terms or expressions. We will use the notations

  $$f_{\text{Sk}} : (S_1, \ldots, S_k; T_1, \ldots, T_m) \rightarrow S \qquad t = f_{\text{Sk}}(r_1, \ldots, r_k; \mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{m})$$

  for a symbol $f_{\text{Sk}} \in \mathit{Func}_{\text{Sk}}$ with $\mathit{PVArg}(f_{\text{Sk}}) = (T_1, \ldots, T_m)$, where $r_1, \ldots, r_k$ are terms and $\mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{m}$ are program variables. $r_1, \ldots, r_k$ can be compared to the arguments of a normal function or predicate symbol.

  Syntactically, the construct t is regarded as a term of sort S, and analogously a construct whose top-level operator $p_{\text{Sk}} \in \mathit{Pred}_{\text{Sk}}$ is a predicate symbol is a formula.

  The symbols $\mathit{Func}_{\text{Sk}} \cup \mathit{Pred}_{\text{Sk}}$ are used as skolem symbols for the schema variables TermSV and FormulaSV. For the proof obligation of example 4.17 we have $p_{\text{Sk}} \in \mathit{Pred}_{\text{Sk}}$.

- A set $\mathit{Statement}_{\text{Sk}}$ of symbols used to skolemize StatementSV, together with a map

  $$\mathit{PVArg} : \mathit{Statement}_{\text{Sk}} \rightarrow \mathit{JavaTypes}^*$$

  which has exactly the same meaning as for the sets $\mathit{Func}_{\text{Sk}}$ and $\mathit{Pred}_{\text{Sk}}$, and a second map

  $$\mathit{JTSize} : \mathit{Statement}_{\text{Sk}} \rightarrow \mathbb{N}$$

  that determines the size of the jump table. *PVArg* and *JTSize* together form the signature of elements of $\mathit{Statement}_{\text{Sk}}$. The notation we will use is

  $$s = s_{\text{Sk}}(\mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{m}; \textbf{break}; \textbf{throw}\ \mathrm{v}_1)$$

[31] By $\mathit{JavaTypes}^*$ we denote the set of finite sequences over *JavaTypes*.

in which generally a list of program variables $\mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{m}$ (as given by *PVArg*) is followed by $\mathit{JTSize}(s_{\mathrm{Sk}})$ statements (for s we have $\mathit{JTSize}(s_{\mathrm{Sk}}) = 2$).

The statements that are admissible members of a jump table are

– **return**-statements, with or without an argument

– **break**- and **continue**-statements, with or without a label

– **throw**-statements.[32]

Informally the statement s from above could be described as: "An arbitrary statement, in which the program variables $\mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{m}$ may occur (free), and which can terminate abruptly only by executing the statements **break** or **throw** v_1". We will however not define explicit semantics similar to this description. Instead the set $\mathbf{A}_{\mathrm{Sk}}$, which is defined in the next section, contains axioms that reflect this informal meaning of the symbols.

Syntactically, s is regarded as a statement, and within Java programs it is allowed to occur at every position at which statements are allowed, and where additionally each of the jump statements of s would not lead to static errors (e.g. because of undefined labels). The following rules could be added to the Java grammar of [GJSB00], in particular referring to Sect. 14.5:

StatementWithoutTrailingSubstatement:
  SkolemStatement

SkolemStatement:
  StatementSkolemSymbol ($\mathit{IdentifierList}_{opt}$; $\mathit{JumpStatementList}_{opt}$)

IdentifierList:
  Identifier
  IdentifierList , *Identifier*

JumpStatementList:
  JumpStatement
  JumpStatementList ; *JumpStatement*

JumpStatement:
  BreakStatement
  ContinueStatement
  ReturnStatement
  ThrowStatement

The symbols are very similar to the anonymous programs that are part of propositional dynamic logic (PDL), but have the additional features to handle abrupt termination (and also update simplification).

- A set $\mathit{Expression}_{\mathrm{Sk}}$ of symbols used to skolemize ExpressionSV, for which the map

[32] These are exactly the reasons that can lead to an abrupt completion of a statement, see [GJSB00].

PVArg is once more continued (and has the same meaning as for $Statement_{\mathrm{Sk}}$),[33] but for which also a further map

$$RType : Expression_{\mathrm{Sk}} \to JavaTypes$$

is defined, which determines the result type of an expression symbol. *PVArg*, *JTSize* and *RType* together form the signature of elements of $Expression_{\mathrm{Sk}}$.

The elements of $Expression_{\mathrm{Sk}}$ can be occur at every position within a Java program where a method call with the same result type would be allowed. The following rules could be added to the Java grammar (Sect. 15.8):

PrimaryNoNewArray:
  SkolemExpression

SkolemExpression:
  ExpressionSkolemSymbol (*IdentifierList*$_{opt}$)

The four defined sets correspond to the schema variable types TermSV, FormulaSV, StatementSV and ExpressionSV respectively, and the properties of the schema variables are modelled by the introduction of the various signature extensions. StatementSV, however, do not provide the possibility to include **throw**-statements in their jumpPrefix (as shown in the example above for symbols of the set $Statement_{\mathrm{Sk}}$). According to Sect. 3.2.2, **throw**-statements can be regarded as implicit and omnipresent members of a jumpPrefix, which are only modelled for statement skolem symbols with greater accuracy.

By Sym_{Sk} we denote the set of all introduced skolem symbols:

$$Sym_{\mathrm{Sk}} := Func_{\mathrm{Sk}} \cup Pred_{\mathrm{Sk}} \cup Statement_{\mathrm{Sk}} \cup Expression_{\mathrm{Sk}}.$$

If the term "skolem symbol" is used in the following sections, we are always referring to the symbols Sym_{Sk}.

4.18 *Example (Example 4.17 continued):* For the skolem symbols in the example above, we have

$$PVArg(s_{\mathrm{Sk}}) = (\mathbf{int}, \mathbf{Throwable}, \mathbf{int}) \qquad JTSize(s_{\mathrm{Sk}}) = 1$$
$$PVArg(p_{\mathrm{Sk}}) = (\mathbf{int})$$

*

[33] It would also be possible to equip expression symbols with a jump table, as it has been done with statement symbols, to express the possibility of an expression to throw exceptions. This would however be a somewhat inconsequential notation, as an expression cannot contain statements. We will therefore assume that expression skolem symbols are implicitly and always able to complete abruptly by exceptions, which are the only possible reason for the abrupt completion of an expression, see [GJSB00].

4.6.3. The Axioms $\mathbf{A}_{\mathsf{Sk}}$

As mentioned in Sect. 4.6.2, to prove the proof obligation φ_{po} of a schematic formula $\mathfrak{P}$, the axioms $\mathbf{A}$ will on the one hand be adapted to the logic JavaCardDL$_{\text{Sk}}$ enriched by skolem symbols Sym_{Sk}, and on the other hand we will add a number of further rules $\mathbf{C}_{\text{Sk}}$:

$$\mathbf{A}_{\text{Sk}} = \{\mathfrak{Q}_{\text{Sk}} \supset \mathfrak{Q} \mid \mathfrak{Q} \in \mathbf{A}\} \cup \mathbf{C}_{\text{Sk}}$$

First we describe the continuations of the axioms $\mathbf{A}$. $\mathbf{A}$ does only contain schematic formulas $\mathfrak{Q} \in \mathbf{A}$ that are defined using the schema variables of Sect. 3.2. To continue these formulas, we first define a continuation of the predicates of Sect. 3.2 (which distinguish valid instantiations of schema variables) to the additional syntactic constructs of JavaCardDL$_{\text{Sk}}$. This is achieved by replacing skolem symbols with surrogate expressions of JavaCardDL:

4.19 *Definition (Instantiations and Skolem Symbols):* Let $S = (\#s_1, \ldots, \#s_n)$ be a tuple of schema variables, and $\iota \in \mathit{Syn}^n_{\text{Sk}}$ an instantiation candidate. To decide whether ι is a valid instantiation of S, we syntactically treat skolem symbols like the following constructs of JavaCardDL:

- Function and predicate skolem symbols $s_{\text{Sk}} \in \mathit{Func}_{\text{Sk}} \cup \mathit{Pred}_{\text{Sk}}$ are regarded as ordinary function and predicate symbols, and program variables that are arguments of s_{Sk} as ordinary terms:

$$s_{\text{Sk}}(r_1, \ldots; \mathrm{v}_1, \ldots) \qquad \mapsto \qquad s(r_1, \ldots, \mathrm{v}_1, \ldots)$$

- Expression skolem symbols $e_{\text{Sk}} \in \mathit{Statement}_{\text{Sk}}$ are regarded as static method calls

$$e_{\text{Sk}}(\mathrm{v}_1, \ldots) \qquad \mapsto \qquad \textbf{Object}.\mathrm{f}\ (\ \mathrm{v}_1,\ ...\ )$$

- Statement skolem symbols $s_{\text{Sk}} \in \mathit{Statement}_{\text{Sk}}$ are regarded as statement blocks, containing a static method call and the jump statements

$$s_{\text{Sk}}(\mathrm{v}_1, \ldots; st_1; \ldots) \qquad \mapsto \qquad \{\ \textbf{Object}.\mathrm{f}\ (\ \mathrm{v}_1,\ ...\ );\ st_1;\ ...\ \}$$

ι is a valid instantiation, if the instantiation ι', in which each skolem symbol has been replaced with the corresponding JavaCardDL term, formula or program, is valid.

Additionally, for a ContextSV $\#s_i = \#ct_i$ that has the SUL-flag set to true, in the instantiation $\iota(\#ct_i)$ no skolem symbol may occur above the symbol $a_{\#ct_i}$. *

For a formula $\mathfrak{Q} \in \mathbf{A}$ we have $\mathfrak{Q}_{\text{Sk}} \in \mathbf{A}_{\text{Sk}}$, where $\mathfrak{Q}_{\text{Sk}}$ is the set of instances of $\mathfrak{Q}$ within the extended vocabulary, following Def. 4.19.

Additional Axioms: $\mathbf{C}_{\mathsf{Sk}}$

The definition of PContextSV implies that calculus rules for Java programs always modify the leading statements within a program block. Unfortunately, the addition of program skolem symbols would thus destroy the (relative) completeness of a set of rules, because there are no rules to deal with them, should they appear in the beginning of a program. For an illustration, consider the proof obligation of example 4.17, which is a formula similar to

$$\langle s_{\mathrm{Sk}}(\ldots);\alpha\rangle\varphi \leftrightarrow \langle s_{\mathrm{Sk}}(\ldots);\beta\rangle\psi.$$

Using a decomposition rule (like the rule provided by classical dynamic logic), one could replace this formula with

$$\langle s_{\mathrm{Sk}}(\ldots)\rangle\langle\alpha\rangle\varphi \leftrightarrow \langle s_{\mathrm{Sk}}(\ldots)\rangle\langle\beta\rangle\psi$$

and then apply rewrite rules to the secondary program blocks, with the first ones being a "prefix", telling about a state possibly different from the top-level state. It would then (hopefully) be possible to transform this equivalence into something obviously valid like

$$\langle s_{\mathrm{Sk}}(\ldots)\rangle\varphi' \leftrightarrow \langle s_{\mathrm{Sk}}(\ldots)\rangle\varphi'.$$

The current JavaCardDL calculus of the KeY system does not contain a rule equivalent to the program decomposition rule of classical dynamic logic. Namely, from a semantic point of view, decomposition operators in JavaCardDL have to deal with the possibility of a statement terminating abruptly. This invalidates the naive equivalence that is utilised by classical dynamic logic

$$\langle ..\ \alpha;\beta\ ...\rangle\varphi \leftrightarrow \langle\alpha\rangle\langle ..\ \beta\ ...\rangle\varphi.$$

For JavaCardDL, this equivalence only holds if the statement α does not complete abruptly (and also provided that α is not a variable declaration). For most statements, this cannot be guaranteed, however.

In the next paragraphs, we define a family of decomposition rules specifically for statement skolem symbols, as well as rules that convert expression skolem symbols to statement symbols. These rules cope with abrupt completion by applying a transformation to the statement $\alpha = s_{\mathrm{Sk}}(\ldots)$. This transformation splits α in two parts $\alpha_1 = s'_{\mathrm{Sk}}(\ldots)$ and α_2, such that the concatenation $\alpha_1;\alpha_2$ is equivalent to the original statement α. Furthermore, the first program fragment α_1 is constructed in a way that prevents abrupt termination, and thus the equivalence

$$\langle ..\ s_{\mathrm{Sk}}(\ldots);\beta\ ...\rangle\varphi \leftrightarrow \langle s'_{\mathrm{Sk}}(\ldots)\rangle\langle ..\ \alpha_2;\beta\ ...\rangle\varphi. \tag{7}$$

holds. For the rules we define the remaining part α_2 is rather simple, and it does in particular no longer contain any skolem symbols, i.e. it is a pure JavaCard program. Hence it is possible to handle α_2 by the application of JavaCardDL rules.

It needs to be stressed that the decomposition rules we introduce are exclusively used to prove the proof obligation φ_{po} of a schematic formula $\mathfrak{P}$. The proof of φ_{po}, on the other hand, is only used as a template for the proofs of $\mathfrak{P}$-instances (as described in the beginning of Sect. 4.6):

$$\vdash_{\mathbf{A}_{\mathrm{Sk}}} \varphi_{\mathrm{po}} \Longrightarrow\ \vdash_{\mathbf{A}} \mathfrak{P}.$$

Thus we will not show the soundness of the decomposition rules, as we are not interested in applications of the rules themselves. We will only assume (in Sect. 4.6.5) that for each instance $\varphi \in \mathfrak{P}$ it is possible to replace rule applications within the proof of φ_{po} with sequences of valid rule applications, only using the rules $\mathbf{A}$.

Statements

To define the decomposition rule, first we have to formulate some requirements for the set $\mathit{Statement}_{\text{Sk}}$ (that are very specific for our situation, but can easily be generalised). Namely, as in equivalence (7) two different skolem symbols s_{Sk} and s'_{Sk} are related, we also split the set $\mathit{Statement}_{\text{Sk}}$ in two parts, and define a mapping of symbols s_{Sk} onto s'_{Sk}. Beside that, to be able to construct the program fragment α_2 of equivalence (7), we require all statement skolem symbols to have a distinguished argument of type **int** (in example 4.17, this argument turns up as the program variable d):

- $\mathit{Statement}_{\text{Sk}}$ is disjoint union of two sets

  $$\mathit{Statement}_{\text{Sk}} = \mathit{Statement}^r_{\text{Sk}} \cup \mathit{Statement}^c_{\text{Sk}}, \qquad \mathit{Statement}^r_{\text{Sk}} \cap \mathit{Statement}^c_{\text{Sk}} = \emptyset$$

  and we have a bijection Dec of $\mathit{Statement}^r_{\text{Sk}}$ to $\mathit{Statement}^c_{\text{Sk}}$:

  $$\mathit{Dec} : \mathit{Statement}^r_{\text{Sk}} \to \mathit{Statement}^c_{\text{Sk}}.$$

  We suppose that this map satisfies for each skolem symbol $s_{\text{Sk}} \in \mathit{Statement}^r_{\text{Sk}}$:

  $$\mathit{PVArg}(\mathit{Dec}(s_{\text{Sk}})) = \mathit{PVArg}(s_{\text{Sk}}) \qquad \mathit{JTSize}(\mathit{Dec}(s_{\text{Sk}})) = 0.$$

  Dec is continued on $\mathit{Statement}^c_{\text{Sk}}$ by $\mathit{Dec}|_{\mathit{Statement}^c_{\text{Sk}}} = \mathit{id}$.

- For each $s_{\text{Sk}} \in \mathit{Statement}_{\text{Sk}}$, the last program variable argument has type **int**:

  $$\mathit{PVArg}(s_{\text{Sk}}) = (\ldots, \mathbf{int}).$$

For a symbol $s_{\text{Sk}} \in \mathit{Statement}^r_{\text{Sk}}$, the decomposition axioms $\mathfrak{D}^{\diamond}_{s_{\text{Sk}}}$ and $\mathfrak{D}^{\Box}_{s_{\text{Sk}}}$ are defined analogously for diamond and box modalities.[34] We show the definition of $\mathfrak{D}^{\diamond}_{s_{\text{Sk}}}$:

$$\mathfrak{D}^{\diamond}_{s_{\text{Sk}}} = \#ct\Big(\langle ..\ s_{\text{Sk}}(\#v_1, \ldots, \#v_l; \#st_1; \ldots; \#st_k);\ ..._{\#pct}\rangle \#p\Big) \leftrightarrow$$
$$\#ct\Big(\langle \mathit{Dec}(s_{\text{Sk}})(\#v_1, \ldots, \#v_l);\rangle\langle ..\ ic\ ..._{\#pct}\rangle \#p\Big)$$

where ic (which corresponds to α_2 in equivalence (7)) is the following if-cascade, in which at most one of the jump statements represented by the schema variables $\#st_1, \ldots, \#st_k$ is selected and executed, depending on the value of the last program variable argument $\#v_l$:

```
{          if ( #v_l == 1 )
                    #st_1;
      else if ( #v_l == 2 )
                    #st_2;
```

[34] With schema variables for modalities, it is of course possible to use a single rule for both cases.

$$\begin{array}{l}\textbf{else if } (\ \#v_l == 3 \ ) \\ \qquad\qquad \dots \\ \textbf{else if } (\ \#v_l == k \ ) \\ \qquad\qquad \#st_k \, ; \qquad \} \end{array}$$

The schema variables are defined as in the following table ($\mathbf{V} := \{\#v_1, \dots, \#v_l\}$):[35]

Sym.	Type	javaType	u.Only	prefix	pvP.	j.Prefix	unique	SUL
$\#ct$	ContextSV						*false*	*false*
$\#pct$	PContextSV							
$\#p$	FormulaSV			$\{\#ct\}$	$\mathbf{V}$			
$\#v_i$	PVariableSV	$PVArg(s_{\mathrm{Sk}})_i$	*false*					
$\#st_i$	StatementSV				$\mathbf{V}$	$\{\#pct\}$		

4.20 *Example (Example 4.17 continued):* We apply the rule $\mathfrak{D}^{\diamond}_{s_{\mathrm{Sk}}}$ to the first program block of the formula φ_{po} in example 4.17:

$$\varphi_{\mathrm{po}} = \langle s_{\mathrm{Sk}}(\mathrm{v_{Sk}}, \mathrm{t}, \mathrm{d}; \textbf{throw } \mathrm{t}); \mathrm{v_{Sk}} = \mathrm{v_{Sk}};\rangle p_{\mathrm{Sk}}(\mathrm{v_{Sk}}) \leftrightarrow \langle s_{\mathrm{Sk}}(\mathrm{v_{Sk}}, \mathrm{t}, \mathrm{d}; \textbf{throw } \mathrm{t});\rangle p_{\mathrm{Sk}}(\mathrm{v_{Sk}})$$

Instantiating $\mathfrak{D}^{\diamond}_{s_{\mathrm{Sk}}}$ to match the situation in this formula results in

$$\begin{aligned}\iota(\mathfrak{D}^{\diamond}_{s_{\mathrm{Sk}}}) = \langle s_{\mathrm{Sk}}(\mathrm{v_{Sk}}, \mathrm{t}, \mathrm{d}; \textbf{throw } \mathrm{t}); \mathrm{v_{Sk}} = \mathrm{v_{Sk}};\rangle p_{\mathrm{Sk}}(\mathrm{v_{Sk}}) \leftrightarrow \\ \langle s'_{\mathrm{Sk}}(\mathrm{v_{Sk}}, \mathrm{t}, \mathrm{d});\rangle\langle \textbf{if } (\ \mathrm{d} == 1 \ ) \textbf{ throw } \mathrm{t}; \mathrm{v_{Sk}} = \mathrm{v_{Sk}};\rangle p_{\mathrm{Sk}}(\mathrm{v_{Sk}})\end{aligned}$$

with $s'_{\mathrm{Sk}} := Dec(s_{\mathrm{Sk}})$. This instance can be used to modify the formula φ_{po} (which would be done in the "destructive" calculus **Ta**), leading to

$$\begin{aligned}\langle s'_{\mathrm{Sk}}(\mathrm{v_{Sk}}, \mathrm{t}, \mathrm{d});\rangle\langle \textbf{if } (\ \mathrm{d} == 1 \ ) \textbf{ throw } \mathrm{t}; \mathrm{v_{Sk}} = \mathrm{v_{Sk}};\rangle p_{\mathrm{Sk}}(\mathrm{v_{Sk}}) \leftrightarrow \\ \langle s_{\mathrm{Sk}}(\mathrm{v_{Sk}}, \mathrm{t}, \mathrm{d}; \textbf{throw } \mathrm{t});\rangle p_{\mathrm{Sk}}(\mathrm{v_{Sk}})\end{aligned}$$

The second program block (beginning with **if**) does not contain skolem symbols anymore (the program variable $\mathrm{v_{Sk}}$ is not a skolem symbol as defined in Sect. 4.6.2), it is therefore possible to apply further rules (provided that rules exist that handle Java programs). *

Expressions

For expression skolem symbols, the same problem as for statement symbols might arise if such a symbol occurs within the first statement of a program block. It is therefore also possible to apply the same solution, which will be realized in two steps: First we provide rules that replace expression skolem symbols e_{Sk} with statement symbols s_{Sk}, to which then the already defined rule $\mathfrak{D}_{s_{\mathrm{Sk}}}$ may be applied. The rules to handle expression symbols are specifically designed for the case of a program block, whose first (active) statement is the assignment

[35] At this point the explicit restriction could be added that the StatementSV $\#st_1, \dots, \#st_k$ may only be instantiated with suitable statements, namely only with jump statements that are allowed as arguments of statement skolem symbols.

```
x = e_Sk ( ... );
...
```

i.e. the value of an expression having an expression skolem symbol as top-level operator is assigned to a program variable. The decision to concentrate on statements of this kind was made because of the design of program rules in KeY, which handle complex expressions by evaluating subexpressions in the correct order, and assigning the results to (new) program variables. Iterated application of such rules leads to elementary assignments of "atomic" expressions to program variables (like the assignment from above), as well as assignments of expressions containing exactly one operator, whose arguments are program variables.

The following rule is therefore sufficient to achieve (relative) completeness only for certain systems of rules for complex expressions[36] For other systems, it would be necessary to modify the rules or add further ones.

For the given example, the result of the rule application would look like (we assume that the result type of e_{Sk} is **int**)

```
{
  int       r;
  Throwable t;
  int       d;
  s_Sk ( ..., r, t, d; throw t );
  x = r;
}
...
```

In this code fragment, the result of the computation performed by e_{Sk} is no longer transferred as the result of an expression, but using an explicit program variable r, and the **throw**-statement (which was implicit for the expression symbol) is now added explicitly. The argument d is to be used by the statement rules from above.

Again we require the sets $\mathit{Statement}_{\mathrm{Sk}}$ and $\mathit{Expression}_{\mathrm{Sk}}$ to meet certain conditions: A map Stmt is needed to map expression skolem symbols e_{Sk} onto statement skolem symbols $s_{\mathrm{Sk}} = \mathit{Stmt}(e_{\mathrm{Sk}})$, and the compatibility of the argument types of s_{Sk} has to be ensured:

- The map Stmt is an injection of $\mathit{Expression}_{\mathrm{Sk}}$ into $\mathit{Statement}^r_{\mathrm{Sk}}$

  $$\mathit{Stmt} : \mathit{Expression}_{\mathrm{Sk}} \to \mathit{Statement}^r_{\mathrm{Sk}}$$

  and satisfies for each $e_{\mathrm{Sk}} \in \mathit{Expression}_{\mathrm{Sk}}$:

  $$\begin{aligned} \mathit{PVArg}(\mathit{Stmt}(e_{\mathrm{Sk}})) &= \mathit{PVArg}(e_{\mathrm{Sk}}) \cdot (\mathit{RType}(e_{\mathrm{Sk}}), \mathbf{Throwable}, \mathbf{int})^{37} \\ \mathit{JTSize}(\mathit{Stmt}(e_{\mathrm{Sk}})) &= 1. \end{aligned}$$

[36] We are not considering the concrete rules of the KeY system in detail at this point, as these rules are currently the object of too frequent modifications to make such considerations useful.

[37] By $a \cdot b$ we denote the concatenation of two tuples a and b.

For each expression symbol $e_{\mathrm{Sk}} \in \mathit{Expression}_{\mathrm{Sk}}$, the decomposition rules $\mathfrak{D}^{\diamond}_{e_{\mathrm{Sk}}}$ and $\mathfrak{D}^{\square}_{e_{\mathrm{Sk}}}$ are defined; we will again show the definition of $\mathfrak{D}^{\diamond}_{e_{\mathrm{Sk}}}$ for diamond modalities:

$$\mathfrak{D}^{\diamond}_{e_{\mathrm{Sk}}} = \#ct\Big(\langle ..\ \#x = e_{\mathrm{Sk}}(\#v_1, \ldots, \#v_l);\ ..._{\#pct}\rangle \#p\Big) \leftrightarrow \#ct\Big(\langle ..\ sb\ ..._{\#pct}\rangle \#p\Big)$$

where sb is the statement block as in the example ($\mathrm{T} := \mathit{RType}(e_{\mathrm{Sk}})$ is the result type of e_{Sk})

```
{
  T          #r;
  Throwable  #t;
  int        #d;
  Stmt(eSk) ( #v1 , ... , #vl , #r , #t , #d; throw #t );
  #x = #r;
}
```

and the schema variables are defined as in the following table, where we use the abbreviation $\mathbf{V} := \{\#v_1, \ldots, \#v_l, \#x\}$:

Sym.	Type	javaType	u.Only	prefix	pvPrefix	unique	SUL
$\#ct$	ContextSV					*false*	*false*
$\#pct$	PContextSV						
$\#p$	FormulaSV			$\{\#ct\}$	$\mathbf{V}$		
$\#v_i$	PVariableSV	$\mathit{PVArg}(e_{\mathrm{Sk}})_i$	*false*				
$\#x$[38]	PVariableSV	$\mathit{RType}(e_{\mathrm{Sk}})$	*false*				
$\#r$	PVariableSV	$\mathit{RType}(e_{\mathrm{Sk}})$	*true*				
$\#t$	PVariableSV	**Throwable**	*true*				
$\#d$	PVariableSV	**int**	*true*				

Altogether, we have

$$\mathbf{C}_{\mathrm{Sk}} := \{\mathfrak{D}^{\diamond}_{s_{\mathrm{Sk}}}, \mathfrak{D}^{\square}_{s_{\mathrm{Sk}}} \mid s_{\mathrm{Sk}} \in \mathit{Statement}^{r}_{\mathrm{Sk}}\} \cup \{\mathfrak{D}^{\diamond}_{e_{\mathrm{Sk}}}, \mathfrak{D}^{\square}_{e_{\mathrm{Sk}}} \mid e_{\mathrm{Sk}} \in \mathit{Expression}_{\mathrm{Sk}}\}$$

4.6.4. Definition of the Proof Obligation

As in Sect. 4.4.1, we define the proof obligation by instantiating schema variables of $\mathfrak{P}$. Unlike the situation for FOL, however, we need more than one instance of $\mathfrak{P}$, to cover different possible instantiations of PVariableSV. Namely, it is possible that

- two PVariableSV (that do not have the unusedOnly-property set to true) are instantiated with the same program variable[39]

[38] We define $\#x$ to have exactly the result type of e_{Sk}; one could, more generally, also allow $\#x$ to have any type to which the type of e_{Sk} may be assigned statically. As in KeY PVariableSV are currently untyped, practically we even allow *any* type for $\#x$.

[39] This exceptional case is mainly critical for free PVariableSV, as the instantiations of bound ones can always be renamed. For most real applications of taclets, dealing with multiple program variables, it is probably necessary to require distinctness in any case, which however cannot be expressed using the taclet mechanism of KeY (currently).

- a PVariableSV is instantiated with a program variable of $PVar_e$,[40] or an explicit program variable occurring in $\mathfrak{P}$ (which can only be a free program variable).

4.21 Example: Instances $\iota(\mathfrak{P})$ of the schematic formula

$$\mathfrak{P} = \langle \#i = 0; \#j = 1; \rangle \#i \doteq 0$$

are valid for $\iota(\#i) \neq \iota(\#j)$, and invalid otherwise. $*$

4.22 Example: Instances $\iota(\mathfrak{Q})$ of the schematic formula

$$\mathfrak{Q} = \langle \#i = 0; j = 1; \rangle \#i \doteq 0$$

are valid for $\iota(\#i) \neq \mathrm{j}$, and invalid otherwise. $*$

Therefore we distinguish these cases, and create an instance $\varphi^i_{\mathrm{po}} = \iota_i(\mathfrak{P})$ of $\mathfrak{P}$ for each of them. We could assume that each instance has to be proved separately, i.e. that we are given a set $\{H_i \mid i \in \mathbf{I}\}$ of proofs, and would then even be able to treat situations in which there are infinitely many different cases. For practical reasons, though, this is of little importance, and we will assume $\mathbf{I}$ to be finite, and define the complete proof obligation by

$$\varphi_{\mathrm{po}} := \bigwedge_{i \in \mathbf{I}} \varphi^i_{\mathrm{po}}.$$

Therefore we start with the instantiation of PVariableSV of $\mathfrak{P}$, and each instantiation $\iota \in \mathbf{J}_{\mathrm{PVariableSV}}$ obtained in this first step will subsequently be continued to an instantiation of all schema variables.

Instantiation of PVariableSV: $\mathbf{J}_{\mathrm{PVariableSV}}$
Let $\#v_1, \ldots, \#v_k$ be the PVariableSV of $\mathfrak{P}$, and let

$$\mathbf{P} := \{\mathrm{v}_1, \ldots, \mathrm{v}_k\} \subset PVar_u \backslash PVar_e(\mathfrak{P})$$

be k distinct program variables, where each v_i shall have the same Java type as $\#v_i$. The instantiations $\mathbf{J}_{\mathrm{PVariableSV}}$ of the PVariableSV are then all instantiations of $\#v_1, \ldots, \#v_k$ with elements of $\mathbf{P} \cup PVar_e(\mathfrak{P})$ that are consistent with the definitions of PVariableSV (especially regarding the properties of $\#v_1, \ldots, \#v_k$).

This simple construction creates more instantiations than necessary; from two instantiations $\iota, \kappa \in \mathbf{J}_{\mathrm{PVariableSV}}$, one may be removed, provided that

- ι and κ instantiate the same PVariableSV with elements of $PVar_e(\mathfrak{P})$, and for $\iota(\#v_i) \in PVar_e(\mathfrak{P})$ we have $\iota(\#v_i) = \kappa(\#v_i)$
- ι and κ instantiate the same PVariableSV equally, i.e. we have

$$\iota(\#v_i) = \iota(\#v_j) \Longleftrightarrow \kappa(\#v_i) = \kappa(\#v_j).$$

[40] A realistic example for this situation would be the existence of a taclet describing the effect of a Java method, which could include the modification of a class attribute. The program variable representing this attribute would then require unusual treatment. This observation also has to be considered when dealing with meta operators for the expansion of method bodies.

From a theoretical point of view, there is no need to remove unnecessary elements of $\mathbf{J}_{\text{PVariableSV}}$ (as long as the set stays finite), whereas practically one would try to make $\mathbf{J}_{\text{PVariableSV}}$ as small as possible, and use a more intelligent (and complicated) construction.

4.23 Example (Example 4.21 continued): For the formula $\mathfrak{P}$ of example 4.21, and assuming that we have

$$PVar_e = PVar_e(\mathfrak{P}) = \{\mathrm{a}\}$$

and a, $\#i$, $\#j$ all have the same type, the chosen instantiations $\mathbf{J}_{\text{PVariableSV}}$ would be

	ι_1	ι_2	ι_3	ι_4	ι_5	ι_6	ι_7	ι_8	ι_9
$\#i$	v_1	v_1	v_1	v_2	v_2	v_2	a	a	a
$\#j$	v_1	v_2	a	v_1	v_2	a	v_1	v_2	a

To minimise $\mathbf{J}_{\text{PVariableSV}}$, it would be sufficient to consider the instantiations ι_1, ι_2, ι_3, ι_7, ι_9, as the others differ only insignificantly. $*$

Instantiation of Bound Schema Variables: $\mathbf{J}_2$

For each $\iota \in \mathbf{J}_{\text{PVariableSV}}$, in the next step we instantiate two further kinds of schema variables of $\mathfrak{P}$ and create the continuations $\kappa \in \mathbf{J}_2$:

- VariableSV $\#v$ are instantiated with arbitrary distinct logical variables that have the same sort as $\#v$ (as in Sect. 4.4.1 for FOL)
- Label schema variables are skolemized exactly like VariableSV (which follows from the fact that the definitions are almost identical): LabelSV are instantiated with distinct labels.

There cannot be any collision, as neither explicit logical variables nor explicit labels are allowed to occur in $\mathfrak{P}$.

Instantiation of the remaining Schema Variables: J

For each $\iota \in \mathbf{J}_2$, we choose the remaining instantiations, and obtain the final $\kappa \in \mathbf{J}$:

To skolemize the four schema variable types TermSV, FormulaSV, StatementSV and ExpressionSV, the skolem symbols introduced in Sect. 4.6.2 are used. Namely, for each schema variable $\#a$ of these types, we first define a set $\mathbf{P}_{\#a}$ of relevant program variables, which consists of

- the already chosen instantiations $\iota(\#v)$, for each PVariableSV $\#v$ of the pvPrefix of $\#a$
- the elements of $PVar_e(\mathfrak{P})$
- if $\#a$ is a StatementSV, we add two program variables $\mathrm{t}_{\#a} \in PVar_u$ of the Java type **Throwable** and $\mathrm{d}_{\#a} \in PVar_u$ of type **int**, not yet occurring anywhere. The variable $\mathrm{t}_{\#a}$ is necessary because the pvPrefix property of StatementSV lacks a **throw**-statement (see Sect. 3.2.2), and we therefore have to add one at this point, and need a suitable argument. The second variable $\mathrm{d}_{\#a}$ is needed to ensure that the signature of the skolem symbol to be introduced fulfils the assumption of Sect. 4.6.3, namely that the type of the last program variable argument has to be **int**.

To give the elements of $\mathbf{P}_{\#a}$ an ordering, we arrange them as a tuple $p_{\#a} = (\mathrm{v}_1, \ldots, \mathrm{v}_l)$, such that the variable $\mathrm{d}_{\#a}$, if existing, is the last component of $p_{\#a}$.

In any case, for the skolem symbol $s_{\mathrm{Sk}}^{\#a}$ for $\#a$ we will then choose

$$PVArg(s_{\mathrm{Sk}}^{\#a}) = (T_1, \ldots, T_l)$$

where $T_1, \ldots, T_l$ are the Java types of the components $\mathrm{v}_1, \ldots, \mathrm{v}_l$ of $p_{\#a}$. The final instantiation is determined as:

- If $\#a$ is a TermSV or FormulaSV, it is instantiated as in Sect. 4.4.1, except that we use a new (i.e. not already used as instantiation of another schema variable) element $s_{\mathrm{Sk}}^{\#a}$ of $\mathit{Func}_{\mathrm{Sk}}$ or $\mathit{Pred}_{\mathrm{Sk}}$ instead of a simple function or predicate symbol, and that it is given the program variables of $\mathbf{P}_{\#a}$ as additional arguments, which altogether looks like

  $$\kappa(\#a) = s_{\mathrm{Sk}}^{\#a}(x_1, \ldots, x_k; \mathrm{v}_1, \ldots, \mathrm{v}_l)$$

  $(x_1, \ldots, x_k$ as in 4.4.1)

- If $\#a$ is a StatementSV, we first define a set $\tilde{\mathbf{T}}_{\#a}$ of jump statements (the statements are mostly derived from the jumpPrefix of $\#a$), to be the smallest set with:

  - $\mathbf{T}_{\#a}(\iota) \subset \tilde{\mathbf{T}}_{\#a}$. Note that $\mathbf{T}_{\#a}(\iota)$, which is defined in Sect. 3.2.2, does only depend on the instantiations of PVariableSV and LabelSV[41]
  - **throw** $\mathrm{t}_{\#a} \in \tilde{\mathbf{T}}_{\#a}$, where $\mathrm{t}_{\#a}$ is the program variable defined above.

  $\#a$ is instantiated with the skolem symbol $s_{\mathrm{Sk}}^{\#a} \in \mathit{Statement}_{\mathrm{Sk}}^{r}$, which is again supposed to be new, and must not be an element of the image of the map Stmt, i.e. must not be a symbol that is used to transform expression skolem symbols to statement skolem symbols as defined in Sect. 4.6.3. For $s_{\mathrm{Sk}}^{\#a}$ we choose

  $$\mathit{JTSize}(s_{\mathrm{Sk}}^{\#a}) = |\tilde{\mathbf{T}}_{\#a}|.$$

  The instantiation of $\#a$ is

  $$\kappa(\#a) = s_{\mathrm{Sk}}^{\#a}(\mathrm{v}_1, \ldots, \mathrm{v}_l; s_1; \ldots; s_k)$$

  where $\{s_1, \ldots, s_k\} = \tilde{\mathbf{T}}_{\#a}$

- If $\#a$ is an ExpressionSV, $\#a$ is instantiated with a new expression skolem symbol $s_{\mathrm{Sk}}^{\#a} \in \mathit{Expression}_{\mathrm{Sk}}$. $\mathit{RType}(s_{\mathrm{Sk}}^{\#a})$ is the type given by the corresponding property of $\#a$. The instantiation of $\#a$ is

  $$\kappa(\#a) = s_{\mathrm{Sk}}^{\#a}(\mathrm{v}_1, \ldots, \mathrm{v}_l).$$

[41] Apart from PContextSV, but such schema variables are not allowed for $\mathfrak{P}$.

The Proof obligation $\varphi_{\mathbf{po}}$

As already mentioned above, the proof obligation φ_{po} is chosen referring to the determined set $\mathbf{J}$ of instantiations:

$$\varphi_{\mathrm{po}} = \bigwedge_{\iota \in \mathbf{J}} \iota(\mathfrak{P}).$$

The next two sections treat the extraction of concrete proofs (i.e. for arbitrary instances of $\mathfrak{P}$) from a proof of the proof obligation. Therefore, first we have to formulate two requirements that are needed to translate the decomposition rules of Sect. 4.6.3.

4.6.5. Assumptions about Java

In the previous sections, skolem symbols for Java code fragments as well as rules $\mathbf{C}_{\mathrm{Sk}}$ to deal with these symbols were introduced, without saying anything (substantial) about the semantics of the symbols or the soundness of the axioms. We could think of statement skolem symbols as some kind of anonymous program, to be interpreted by an input/output relation (and something additional, telling about abrupt completion), and see that the rules preserve consistency.

We will rather give an indirect argument for the soundness of the rules, namely by requiring the rules to be sound after having replaced the contained skolem symbols with suitable Java code fragments.

Statements

The intention of the following assumptions is that it shall be possible to transform Java statements into equivalent statements, in such a way that its behaviour regarding abrupt termination can be controlled. As an example, the statement

$$s = s_{\mathrm{Sk}}(\mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{m}; \mathbf{break})$$

should handle modifications that are applied to the jump statement **break** in a faithful way. The statement

$$s' = s_{\mathrm{Sk}}(\mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{m}; \mathbf{break}\ \mathrm{l})$$

is expected (informally) to behave like s, except that each time s completes abruptly by executing **break**, s' should execute **break** l. This postulation is inherited to statements by which s_{Sk} can be replaced. Within the following paragraphs, we justify the existence of statements satisfying this demand, which are thus possible substitutions for s_{Sk}, and outline an effective method of replacement. For that, we introduce three mappings *Norm*, $\mathit{Norm}_{\mathrm{Dec}}$ and $\mathit{Norm}_{\mathrm{c}}$, which “normalise” arbitrary Java statements to meet our requirements.

Let α be a JavaCard statement respecting scopes, and $\mathbf{T} = \{s_1, \ldots, s_l\}$ be a set of statements (all not containing skolem symbols or **method−frame**s), such that α is compliant to $\mathbf{T}$ regarding termination behaviour (following Def. 3.14). The free program variables of α and those of the elements of $\mathbf{T}$ shall be subsumed by $\mathrm{v}_1, \ldots, \mathrm{v}_k$. Further let $\mathrm{m}_1, \ldots, \mathrm{m}_{l+1}$ be $l+1$ distinct labels, d be a program variable of type **int**, and t a program variable of type **java.lang.Throwable**, all not occurring in α or $\mathbf{T}$.

We assume that two Java statements

$$Norm(\alpha) = Norm(\alpha, \mathrm{d}, \mathrm{t}, \mathrm{m}_1, \ldots, \mathrm{m}_{l+1}), \qquad Norm_{\mathrm{Dec}}(\alpha) = Norm_{\mathrm{Dec}}(\alpha, \mathrm{d}, \mathrm{t})$$

exist, such that

1. $Norm(\alpha)$ and $Norm_{\mathrm{Dec}}(\alpha)$ respect scopes, and do *not* contain

   - **method−frame**s
   - free symbols beside the symbols that also occur in α, except from the program variables d, t:

     $$FS(Norm(\alpha)) \subset FS(\alpha) \cup \{\mathrm{d}, \mathrm{t}\}, \qquad FS(Norm_{\mathrm{Dec}}(\alpha)) \subset FS(\alpha) \cup \{\mathrm{d}, \mathrm{t}\}$$

     (FS as introduced in Sect. A.2 for arbitrary JavaCardDL$_{\mathrm{Sk}}$ statements).

   Furthermore the sets $FS(Norm(\alpha))$ and $FS(Norm_{\mathrm{Dec}}(\alpha))$ of free symbols do *not* contain

   - labels
   - skolem symbols.

   And finally the statement $Norm_{\mathrm{Dec}}(\alpha)$ is termination compliant to $\emptyset$.[42]

2. For each formula φ that does not contain skolem symbols or the program variables d and t, and each formula φ' that is obtained from φ by replacing a number of occurrences of α by $\pi(Norm(\alpha)) =: Norm_{\mathrm{c}}(\alpha)$:

   $$\vdash_{\mathbf{HT_A}} \varphi \leftrightarrow \varphi'$$

   where π is the parameter substitution (these substitutions are defined in Sect. A.2)

   $$\pi = \{\mathrm{m}_1/s_1, \ldots, \mathrm{m}_l/s_l, \mathrm{m}_{l+1}/\mathbf{throw}\ \mathrm{t}\}.$$

   This means that the statements α and $Norm(\alpha)$ are treated as equivalent by the calculus $\mathbf{HT_A}$, if the original jump statements are inserted into $Norm(\alpha)$.

3. The statement decomposition axioms of Sect. 4.6.3 become derivable, if the contained skolem symbols are replaced with $Norm(\alpha)$ and $Norm_{\mathrm{Dec}}(\alpha)$. This means (very informally) that the equivalence

   $$\langle ..\ Norm(\alpha)\ ...\rangle\varphi \leftrightarrow \langle Norm_{\mathrm{Dec}}(\alpha)\rangle\langle ..\ ic\ ...\rangle\varphi$$

   (in which *ic* is the if-cascade as in Sect. 4.6.3) is valid. To decompose a program block, it is then possible to replace the statement $Norm(\alpha)$ with $Norm_{\mathrm{Dec}}(\alpha)$, and the assumption guarantees that no problems regarding abrupt termination arise.

[42] By Def. 3.14, this means that $Norm_{\mathrm{Dec}}(\alpha)$ can only complete abruptly through an exception.

- Let $e \in \{0, \ldots, k\}$ and $s_{\mathrm{Sk}} \in \mathit{Statement}^r_{\mathrm{Sk}}$ be a statement skolem symbol with $\mathit{JTSize}(s_{\mathrm{Sk}}) = l + 1$ and

$$\mathit{PVArg}(s_{\mathrm{Sk}}) = (T_1, \ldots, T_e, \mathbf{Throwable}, \mathbf{int})$$

  where $T_1, \ldots, T_e$ are the types of the program variables $\mathrm{v}_1, \ldots, \mathrm{v}_e$

- Suppose σ is the s-substitution that replaces the skolem symbols s_{Sk} and $\mathit{Dec}(s_{\mathrm{Sk}})$ (the symbols occurring in a decomposition axiom of Sect. 4.6.3) with $\mathit{Norm}(\alpha)$ and $\mathit{Norm}_{\mathrm{Dec}}(\alpha)$ respectively:

$$\sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (\mathit{Norm}(\alpha), \langle \mathrm{v}_1, \ldots, \mathrm{v}_e, \mathrm{t}, \mathrm{d}, \mathrm{m}_1, \ldots, \mathrm{m}_{l+1}\rangle)$$
$$\sigma_{\mathrm{cont}}(\mathit{Dec}(s_{\mathrm{Sk}})) = (\mathit{Norm}_{\mathrm{Dec}}(\alpha), \langle \mathrm{v}_1, \ldots, \mathrm{v}_e, \mathrm{t}, \mathrm{d}\rangle)$$

  (σ, σ_{cont} as in Def. A.21)

- Suppose $\varphi \in \mathfrak{D}_{s_{\mathrm{Sk}}}$ is an instance of the axiom $\mathfrak{D}_{s_{\mathrm{Sk}}}$ (for diamond or box modalities), which does not contain any skolem symbols except s_{Sk} and $\mathit{Dec}(s_{\mathrm{Sk}})$, and such that an s-substitution $\omega =_{\mathrm{br}} \sigma$ exists that can be applied to φ without collisions[43]

- Finally, our assumption regarding $\mathit{Norm}(\alpha)$, $\mathit{Norm}_{\mathrm{Dec}}(\alpha)$ is

$$\vdash_{\mathbf{HT_A}} \omega(\varphi).$$

4.24 *Example:* We illustrate the three items of the assumption, to demonstrate that the assumption is justified:

1. In the next paragraphs, one possible derivation of the statements $\mathit{Norm}(\alpha)$, $\mathit{Norm}_{\mathrm{Dec}}(\alpha)$ is performed for the following statement α, in which v is supposed to be a program variable of type **int**:

```
{     if ( v == 0 )
            return 3;
      else
            continue loop;
      idle : { break idle; }
      throw new Exception ();      }
```

   α is compliant to (as exceptions are not considered for compliance following Def. 3.14, and the target of the **break**-statement lies inside α):

$$\mathbf{T} = \{\mathbf{continue}\ \mathrm{loop}, \mathbf{return}\ \mathrm{r}\},$$

   where r is a program variable of type **int**. Thus altogether there are two program variables, i.e. $k = 2$ and $\mathrm{v}_1 = \mathrm{r}$, $\mathrm{v}_2 = \mathrm{v}$. The set $\mathbf{T}$ is regarded as ordered, namely we denote the elements by s_1, s_2 as above. We choose the $l + 1 = 3$ labels that are required above simply to be $\mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3$, and call the two additional program variables d and t. $\mathit{Norm}_{\mathrm{Dec}}(\alpha)$ is obtained by

[43] As $\mathrm{Cod}(\sigma)$ does not contain labels or logical variables, a sufficient condition for that would be that φ must not contain bound program variables which are also elements of $\mathrm{Cod}(\sigma)$.

- enclosing α by a **try**-block labelled with exit, catching everything
- replacing representatives of statements of the set **T** within α (according to Def. 3.14) by statements that trigger a "clean" (i.e. not abrupt) completion of α, and assign the variables d, t appropriately. For the statement **continue** loop, which is the first statement s_1 of **T**, this replacement would be

```
{ d = 1; break exit; }
```

The resulting statement $Norm_{\mathrm{Dec}}(\alpha)$ is

```
{   d = 0;
    exit : try {
      {   if ( v == 0 )
                { d = 2; r = 3; break exit; }
          else
                { d = 1;        break exit; }
          idle : { break idle; }
          throw new Exception ();   }
    } catch ( Throwable e ) {
      d = 3; t = e;
    }   }
```

Note that the target of the jump statements "**break** exit" that have been inserted lies inside of $Norm_{\mathrm{Dec}}(\alpha)$, and thus $Norm_{\mathrm{Dec}}(\alpha)$ is termination compliant to $\emptyset$.

To construct the statement $Norm(\alpha)$, the if-cascade which is defined in Sect. 4.6.3 for the statement decomposition rule $\mathfrak{D}_{s_{\mathrm{Sk}}}$ is added to $Norm_{\mathrm{Dec}}(\alpha)$. The schema variables of the if-cascade are instantiated with the variable d and the labels $\mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3$ (the instantiation of the schema variables reflects the definition of parameter substitutions in Def. A.19):

```
{         NormDec(α)
          if ( d == 1 )
                m1 : skip;
     else if ( d == 2 )
                m2 : skip;
     else if ( d == 3 )
                m3 : skip;     }
```

2. If the parameter substitution π is defined as in the assumption above, then the statement $\pi(Norm(\alpha))$ will be

```
{         NormDec(α)
          if ( d == 1 )
                continue loop;
     else if ( d == 2 )
                return r;
     else if ( d == 3 )
```

```
                  throw t;          }
```

which is (semantically) equivalent to the original statement α, except that the program variables d, t are modified.

3. To demonstrate the last assumption, we choose $e = 1$ (i.e. the second variable v will be removed) and consider the instance

$$\varphi = \Big(\langle s_{\mathrm{Sk}}(\mathrm{h}, \mathrm{t}, \mathrm{i}; \textbf{break}\ \mathrm{loop}; \textbf{return}\ \mathrm{h}; \textbf{throw}\ \mathrm{t});\rangle \mathit{true}\Big) \leftrightarrow \Big(\langle s'_{\mathrm{Sk}}(\mathrm{h}, \mathrm{t}, \mathrm{i});\rangle\langle ic\rangle \mathit{true}\Big)$$

of $\mathfrak{D}^{\diamond}_{s_{\mathrm{Sk}}}$. Compared to the statement $\pi(\mathit{Norm}(\alpha))$ of the last item, in this formula program variables have been renamed and jump statements have been modified:

$$\mathrm{v} \mapsto \mathrm{h}, \quad \mathrm{d} \mapsto \mathrm{i}, \quad \textbf{continue}\ \mathrm{loop} \mapsto \textbf{break}\ \mathrm{loop}, \quad \textbf{return}\ \mathrm{r} \mapsto \textbf{return}\ \mathrm{h}.$$

ic once again is the already well-known if-cascade, which is also instantiated with the new program variables and jump statements:

```
{          if  ( i == 1 )
                 break loop;
      else if  ( i == 2 )
                 return h;
      else if  ( i == 3 )
                 throw t;          }
```

If the s-substitution ω is defined as described and applied to φ, the statements containing s_{Sk} and s'_{Sk} are replaced with

$$\{\mathrm{r}/\mathrm{h}, \mathrm{t}/\mathrm{t}, \mathrm{d}/\mathrm{i}, \mathrm{m}_1/\textbf{continue}\ \mathrm{loop}, \mathrm{m}_2/\textbf{break}\ \mathrm{loop}, \mathrm{m}_3/\textbf{return}\ \mathrm{r}\}\mathit{Norm}(\alpha),$$
$$\{\mathrm{r}/\mathrm{h}, \mathrm{t}/\mathrm{t}, \mathrm{d}/\mathrm{i}\}\mathit{Norm}_{\mathrm{Dec}}(\alpha).$$

It can be observed that the replacement of the skolem symbols s_{Sk}, s'_{Sk} exactly matches the chosen if-cascade *ic* (regarding program variables and jump statements), which is obtained by the instantiation of schema variables. The reader can also ascertain that the JavaCardDL formula $\omega(\varphi)$ is valid. $*$

Expressions

As for statement skolem symbols, we show that there are suitable JavaCard expressions and statements that can be used to substitute expression skolem symbols. Therefore, we continue the maps *Norm* and $\mathit{Norm}_{\mathrm{Dec}}$ that are defined for statement skolem symbols to expressions skolem symbols.

Let γ be a JavaCard expression of type T_γ, which does not contain skolem symbols, and whose free program variables are subsumed by the variables $\mathrm{v}_1, \ldots, \mathrm{v}_k$. Moreover let r be a program variable not occurring in γ, which is also of type T_γ.

We assume that there is a JavaCard statement α, such that

1. α is termination compliant to the empty set, and does not contain free symbols beside those that occur in γ, except for r:

$$FS(\alpha) \subset FS(\gamma) \cup \{\mathrm{r}\}$$

Furthermore $FS(\alpha)$ does not contain any labels or skolem symbols.

2. According to the previous part of this section, the following two statements exist for α:

$$Norm(\gamma) := Norm(\alpha), \qquad Norm_{\text{sep}}(\gamma) := Norm_{\text{Dec}}(\alpha).$$

We assume that the expression "normalisation" axiom $\mathfrak{D}_{e_{\text{Sk}}}$ for an expression skolem symbol e_{Sk} becomes derivable, if we substitute the expression γ and the statement $Norm(\gamma)$ for the skolem symbols in this axiom. The following construction is very similar to the one already given for statement skolem symbols:

- Let $e \in \{0, \ldots, k\}$ and $e_{\text{Sk}} \in Expression_{\text{Sk}}$ be an expression skolem symbol with

$$PVArg(e_{\text{Sk}}) = (T_1, \ldots, T_e)$$

where $T_1, \ldots, T_e$ are the types of the program variables $\mathrm{v}_1, \ldots, \mathrm{v}_e$

- Suppose σ is the s-substitution that replaces the skolem symbols e_{Sk} and $Stmt(e_{\text{Sk}})$ (the symbols occurring in an expression axiom of Sect. 4.6.3) with γ and $Norm(\gamma)$ respectively:

$$\sigma_{\text{cont}}(e_{\text{Sk}}) = (\gamma, \langle \mathrm{v}_1, \ldots, \mathrm{v}_e \rangle)$$
$$\sigma_{\text{cont}}(Stmt(e_{\text{Sk}})) = (Norm(\gamma), \langle \mathrm{v}_1, \ldots, \mathrm{v}_e, \mathrm{r}, \mathrm{t}, \mathrm{d}, \mathrm{m} \rangle)$$

(σ, σ_{cont} as in Def. A.21, and t, d, m $= \mathrm{m}_1$ as in the previous section)

- Suppose $\varphi \in \mathfrak{D}_{e_{\text{Sk}}}$ is an instance of the axiom $\mathfrak{D}_{e_{\text{Sk}}}$ (for diamond or box modalities), which does not contain skolem symbols except e_{Sk} and $Stmt(e_{\text{Sk}})$, and such that an s-substitution $\omega =_{\text{br}} \sigma$ exists that can be applied to φ without collisions[44]

- Finally, our assumption is

$$\vdash_{\mathbf{HT_A}} \omega(\varphi).$$

4.25 Example: Again we illustrate the two items of the assumption by considering an example:

1. We show one possible construction of the statement α for the expression γ

```
a + 2*(o.b)
```

which contains the two program variables a, o (logically the attribute b is not treated as a program variable, but as a unary mapping of the class of o, see [Bec01]). The corresponding statement α is simply obtained by assigning the value of γ to the program variable r:

```
r = a + 2*(o.b)
```

2. The derivation of the statements $Norm(\gamma)$, $Norm_{\text{Dec}}(\gamma)$ as in example 4.24 results in

[44] Again, a sufficient condition would be that φ must not contain bound program variables which are also elements of $\mathrm{Cod}(\sigma)$.

```
{    d = 0;
     exit : try {
       r = a + 2*(o.b);
     } catch ( Throwable e ) {
       d = 1; t = e;
     }    }
```

as $Norm_{\text{Dec}}(\gamma)$, and analogously $Norm(\gamma)$ is the statement

```
{        NormDec(γ)
         if ( d == 1 )
             m : skip;       }
```

As an example we consider the following instance $\varphi \in \mathfrak{D}^{\Box}_{e_{\text{Sk}}}$ of an axiom as defined in Sect. 4.6.3:

$$\varphi = \Big([\, \mathrm{x} = e_{\text{Sk}}(\mathrm{a}, \mathrm{o});\,]\mathit{false}\Big) \leftrightarrow \Big([\, \mathit{sb}\,]\mathit{false}\Big)$$

where *sb* is the statement block

```
{
  int        r;
  Throwable  t;
  int        d;
  Stmt(eSk) ( a, o, r, t, d; throw t );
  x = r;
}
```

Substituting the original expression γ and the statement $Norm(\gamma)$ in φ leads to the formula

$$\varphi = \Big([\, \mathrm{x} = \mathrm{a} + 2{*}(\mathrm{o.b});\,]\mathit{false}\Big) \leftrightarrow \Big([\, \mathit{sb}\,]\mathit{false}\Big)$$

where *sb* is the statement

```
{
  int        r;
  Throwable  t;
  int        d;
  {        d = 0;
           exit : try {
             r = a + 2*(o.b);
           } catch ( Throwable e ) {
             d = 1; t = e;
           }
           if ( d == 1 )
               throw t;          }
  x = r;
}
```

*

4.6.6. Lifting Proofs

Remember that the proof obligation φ_{po} of the schematic formula $\mathfrak{P}$ is defined to be the following conjunction:

$$\varphi_{\mathrm{po}} = \bigwedge_{\iota \in \mathbf{J}} \iota(\mathfrak{P}).$$

As in Sect. 4.4.2, we are assuming to be provided a closed proof H of this proof obligation

$$\neg\varphi_{\mathrm{po}}, \psi_1, \ldots, \psi_k$$

using the calculus $\mathbf{HT}_{\mathbf{A}_{\mathrm{Sk}}}$, as well as an instance $\varphi \in \mathfrak{P}$ (which is supposed to be formulated using the original logic, i.e. not containing any of the skolem symbols of Sect. 4.6.2)[45] that is to be proved using the calculus $\mathbf{HT}_{\mathbf{A}}$. As before, we achieve this by replacing skolem symbols in H with the instantiations that define $\varphi = \mu(\mathfrak{P})$, followed by some further transformations, and showing that this modifications lead to a new proof still correct.

Preliminaries

We start by selecting the part $\iota(\mathfrak{P})$ of the proof obligation we are interested in (or: that we have to use), which is the one that has

- the same PVariableSV instantiated with (the same) elements of $PVar_e(\mathfrak{P})$ as φ, i.e.

$$\begin{aligned} \iota(\#u) \in PVar_e(\mathfrak{P}) &\Longleftrightarrow \mu(\#u) \in PVar_e(\mathfrak{P}), \\ \iota(\#u) \in PVar_e(\mathfrak{P}) &\Longrightarrow \iota(\#u) = \mu(\#u) \end{aligned}$$

- the same PVariableSV instantiated equally, i.e. we have

$$\iota(\#u) = \iota(\#v) \Longleftrightarrow \mu(\#u) = \mu(\#v).$$

Such an instantiation ι must have been used to construct a part of φ_{po}, because μ is by assumption a valid instantiation (according to the schema variable definitions in Sect. 3.2), and in Sect. 4.6.4 we demanded to cover all possible combinations regarding the PVariableSV instantiations.

4.26 Example (Example 4.23 continued): We consider the instantiation $\mu_1 = \{\#i/\mathrm{a}, \#j/\mathrm{t}\}$ of the schema variables of example 4.23, where $\mathrm{a} \in PVar_e$ and $\mathrm{t} \in PVar_u$ is a further program variable. Of the instantiations ι_i of example 4.23, ι_7 and ι_8 would then be admissible choices for the following consideration. For a second instantiation $\mu_2 = \{\#i/\mathrm{t}, \#j/\mathrm{t}\}$, we would choose ι_1 or ι_5. *

We have

$$\neg\varphi_{\mathrm{po}} = \neg \bigwedge_{\pi \in \mathbf{J}} \pi(\mathfrak{P}) \equiv_0 \bigvee_{\pi \in \mathbf{J}} \neg\pi(\mathfrak{P}) \equiv_0 \neg\iota(\mathfrak{P}) \vee \Big(\bigvee_{\pi \in \mathbf{J}\setminus\iota} \neg\pi(\mathfrak{P}) \Big)$$

[45] It should not be difficult to remove this restriction, but we try to simplify the following proof.

and if the formulas of H

$$\neg\varphi_{\mathrm{po}}, \psi_1, \ldots, \psi_k$$

are propositionally unsatisfiable, then so are

$$\neg\iota(\mathfrak{P}), \psi_1, \ldots, \psi_k$$

which means that from H we can derive a proof $\tilde{H}$ of $\tilde{\varphi}_{\mathrm{po}} := \iota(\mathfrak{P})$.

To make the argumentation easier, we will make some assumptions that can always be established by minor modifications of $\tilde{H}$:

- We will treat *all* instantiations of PVariableSV regarding ι and μ as equal. Namely, if the instantiations $\iota(\#v) \neq \mu(\#v)$ are unequal for a PVariableSV $\#v$, then we have $\iota(\#v), \mu(\#v) \in PVar_u \backslash PVar_e(\mathfrak{P})$ (this is guaranteed by the choice of ι). Therefore $\iota(\#v)$ and $\mu(\#v)$ occur in $\tilde{\varphi}_{\mathrm{po}}$ only as instantiation of PVariableSV or as an argument of a skolem symbol, introduced for the proof obligation and another schema variable. By renaming program variables in the whole proof $\tilde{H}$ it is then possible to identify the instantiations.[46]
- As in Sect. 4.4.2 (because **HT** allows bound renaming), we can assume VariableSV to be equally instantiated.
- Additionally, LabelSV can be assumed to be instantiated equally (by the same considerations as for VariableSV).
- If the instantiation of a TermSV, FormulaSV, StatementSV or ExpressionSV $\#a$ in φ contains program variables which are not the instantiations of elements of the pvPrefix of $\#a$, and which are not elements of $PVar_e(\mathfrak{P})$ either (this can happen, following the definition of the pvPrefix in Sect. 3.2.2), then these program variables are supposed not to occur in the proof $\tilde{H}$ or in the instantiation ι. This is no restriction, as such variables have to be elements of $PVar_u \backslash PVar_e(\mathfrak{P})$, and occurrences can therefore be renamed.
- Constants $c \in \mathbf{C}_{(\mathrm{Ex})}$ introduced by the rule (Ex) in $\tilde{H}$ do not occur in φ; this can be established by renaming those constants in $\tilde{H}$, using Def. 4.8.

This means that the "only" schema variable types that need to be considered are the types TermSV, FormulaSV, StatementSV and ExpressionSV. As in Sect. 4.4.2, we will now define an s-substitution σ (such substitutions are introduced in Sect. A.2), that replaces those skolem symbols which are used to define $\iota(\mathfrak{P}) = \tilde{\varphi}_{\mathrm{po}}$ as part of the proof obligation with the concrete instantiations of μ.

To avoid collisions, subsequently we derive from σ and for each formula ξ of the proof $\tilde{H}$ an s-substitution $\sigma_\xi =_{\mathrm{br}} \sigma$. Applying these particular substitutions to the formulas of $\tilde{H}$ will give us the sequence of formulas

$$\neg\varphi'_{\mathrm{po}}, \psi'_1, \ldots, \psi'_k$$

[46] More exactly, we are able to permute $\iota(\#v)$ and $\mu(\#v)$ in the whole proof $\tilde{H}$ (also using Def. 4.8), and after finitely many transpositions all PVariableSV instantiations of ι and μ are equal.

with $\xi' := \sigma_\xi(\xi)$. In general this sequence is *not* yet an $\mathbf{HT_A}$-proof, as the formulas will usually not be the results of valid rule applications. Within the remaining part of this section, we will show that nevertheless the following propositions hold:

1. The set $\{\neg\varphi'_{\mathrm{po}}, \psi'_1, \ldots, \psi'_k\}$ is propositionally unsatisfiable
2. There is an (usually not closed) $\mathbf{HT_A}$-proof H_φ
$$\neg\varphi, \zeta'_1, \ldots, \zeta'_n$$
that entails $\neg\varphi'_{\mathrm{po}}$ propositionally
$$\{\neg\varphi, \zeta'_1, \ldots, \zeta'_n\} \vDash_0 \neg\varphi'_{\mathrm{po}}$$
3. Each formula ψ'_i can either be introduced by a $\mathbf{HT_A}$-rule application, or there is a closed $\mathbf{HT_A}$-proof $H_{\psi'_i}$ of ψ'_i.

Using Lem. 4.14 it is then possible to combine the proofs H_φ and the formulas ψ'_i (or the corresponding proof $H_{\psi'_i}$), preserving the unsatisfiability of the original sequence of formulas, and obtain a closed $\mathbf{HT_A}$-proof H' of φ. Namely, without loss of generality the rule applications by which the formulas ψ'_i (or the proofs H_φ) have been created stay valid:

- Applications of the rule (Ax) are always valid, as the set $\mathbf{A}$ of axioms is not modified
- Applications of (Ex) can always be made valid by renaming introduced constants, referring to Def. 4.8 and the definition of $\mathbf{A}$ in Sect. 4.6.1.

Definition of the s-Substitutions σ_ξ

As in Def. A.21 and for FOL, we will define σ through a map σ_{cont}. For StatementSV and ExpressionSV, not only the skolem symbols of the proof obligation itself will be replaced with the concrete instantiations from φ, but also the symbols used by the rules $\mathbf{C}_{\mathrm{Sk}}$ (that are given by the maps *Dec*, *Stmt*). Additionally, the instantiations from φ will in this case be "normalised", referring to Sect. 4.6.5.

We are aiming at an application of Lem. A.36 about the application of s-substitutions to instances of schematic formulas, which will provide the s-substitutions $\sigma_\xi =_{\mathrm{br}} \sigma$, if σ is compatible with the formulas ξ (and compatible with the instantiations each formula ξ is originating from, see Def. A.34). As a premise of compatibility, σ must in particular be collision preventing regarding these formulas (Def. A.28), and thus we will consider this feature for each of the following items.

- For a TermSV or FormulaSV $\#a$, instantiated with $T = \mu(\#a)$ in φ, and with
$$\iota(\#a) = s^{\#a}_{\mathrm{Sk}}(x_1, \ldots, x_r; \mathrm{v}_1, \ldots, \mathrm{v}_l)$$
in $\tilde{\varphi}_{\mathrm{po}}$:
$$\sigma_{\mathrm{cont}}(s^{\#a}_{\mathrm{Sk}}) := (T, \langle x_1, \ldots, x_r, \mathrm{v}_1, \ldots, \mathrm{v}_l\rangle)$$
Note that, because of the definitions of schema variable types in Sect. 3.2, T cannot contain free logical variables except $x_1, \ldots, x_r$. Moreover, free program variables of T except $\mathrm{v}_1, \ldots, \mathrm{v}_l$ do not occur in any formula of $\tilde{H}$ (by assumption), which is the premise of Def. A.28.

- For a StatementSV $\#a$, instantiated with $\alpha = \mu(\#a)$ in φ, and with

$$\iota(\#a) = s_{\mathrm{Sk}}^{\#a}(\mathrm{v}_1, \ldots, \mathrm{v}_r, \mathrm{t}_{\#a}, \mathrm{d}_{\#a}; s_1; \ldots; s_l; \textbf{throw}\ \mathrm{t}_{\#a})$$

in $\tilde{\varphi}_{\mathrm{po}}$ (w.l.o.g. the program variables and statements appear in this order within $\iota(\#a)$): Because α is a valid instantiation of $\#a$, it respects scopes and is termination compliant to the set $\mathbf{T}_{\#a}(\mu) = \{s_1, \ldots, s_l\}$ of statements. Namely, in any case this set is also the set derived from the jumpPrefix of $\#a$ as described in Sect. 3.2.2, because ι and μ are equal restricted to VariableSV and LabelSV.

α may again contain free program variables beside $\mathrm{v}_1, \ldots, \mathrm{v}_r$, that do however not occur in any formula of $\tilde{H}$. By assumption, in particular the program variables $\mathrm{t}_{\#a}$, $\mathrm{d}_{\#a}$ do not occur in α. Furthermore variables of the statements $s_1, \ldots, s_l$ have to occur within $\mathrm{v}_1, \ldots, \mathrm{v}_r$ by Rem. 3.17.

Providing labels $\mathrm{m}_1, \ldots, \mathrm{m}_{l+1}$ (and the variables $\mathrm{t}_{\#a}, \mathrm{d}_{\#a}$ already existing), the assumption regarding Java statements in Sect. 4.6.5 then furnishes us with statements $\mathit{Norm}(\alpha)$ and $\mathit{Norm}_{\mathrm{Dec}}(\alpha)$ that we use for σ_{cont}, and that make σ collision preventing, regarding each formula of $\tilde{H}$:

$$\begin{aligned}\sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}^{\#a}) &= (\mathit{Norm}(\alpha), \langle \mathrm{v}_1, \ldots, \mathrm{v}_r, \mathrm{t}_{\#a}, \mathrm{d}_{\#a}, \mathrm{m}_1, \ldots, \mathrm{m}_{l+1}\rangle)\\ \sigma_{\mathrm{cont}}(\mathit{Dec}(s_{\mathrm{Sk}}^{\#a})) &= (\mathit{Norm}_{\mathrm{Dec}}(\alpha), \langle \mathrm{v}_1, \ldots, \mathrm{v}_r, \mathrm{t}_{\#a}, \mathrm{d}_{\#a}\rangle)\end{aligned}$$

Note that this definition implies

$$\sigma(\iota(\#a)) = \mathit{Norm}_{\mathrm{c}}(\alpha)$$

where $\mathit{Norm}_{\mathrm{c}}(\alpha)$ is the statement defined in Sect. 4.6.5 (we are going to refer to this property later).

- For a ExpressionSV $\#a$, instantiated with $\gamma = \mu(\#a)$ in φ, and with

$$\iota(\#a) = e_{\mathrm{Sk}}^{\#a}(\mathrm{v}_1, \ldots, \mathrm{v}_r)$$

in $\tilde{\varphi}_{\mathrm{po}}$: As always, γ may contain further program variables, not occurring in any formula of $\tilde{H}$.

Providing program variables $\mathrm{r}_{\#a}, \mathrm{t}_{\#a}, \mathrm{d}_{\#a}$ and a label $\mathrm{m} = \mathrm{m}_1$, again Sect. 4.6.5 guarantees the existence of statements $\mathit{Norm}(\gamma)$, $\mathit{Norm}_{\mathrm{sep}}(\gamma)$ (and again σ is collision preserving, particularly as a Java expression cannot contain labels):

$$\begin{aligned}\sigma_{\mathrm{cont}}(e_{\mathrm{Sk}}^{\#a}) &= (\gamma, \langle \mathrm{v}_1, \ldots, \mathrm{v}_r\rangle)\\ \sigma_{\mathrm{cont}}(\mathit{Stmt}(e_{\mathrm{Sk}}^{\#a})) &= (\mathit{Norm}(\gamma), \langle \mathrm{v}_1, \ldots, \mathrm{v}_r, \mathrm{r}_{\#a}, \mathrm{t}_{\#a}, \mathrm{d}_{\#a}, \mathrm{m}\rangle)\\ \sigma_{\mathrm{cont}}(\mathit{Dec}(\mathit{Stmt}(e_{\mathrm{Sk}}^{\#a}))) &= (\mathit{Norm}_{\mathrm{sep}}(\gamma), \langle \mathrm{v}_1, \ldots, \mathrm{v}_r, \mathrm{r}_{\#a}, \mathrm{t}_{\#a}, \mathrm{d}_{\#a}\rangle).\end{aligned}$$

By the definition of the proof obligation φ_{po}, the s-substitution σ is well-defined (i.e. no skolem symbol occurs more than once within the items above), and it is also collision preserving regarding each formula of $\tilde{H}$. There may however be further skolem symbols $s_{\mathrm{Sk}} \in \mathit{Sym}_{\mathrm{Sk}}$, possibly also occurring within formulas of $\tilde{H}$, e.g. symbols that are not introduced by the instantiation ι (or indirectly through the maps *Dec*, *Stmt*), but by another instantiation used to define φ_{po}. These symbols cannot occur in $\tilde{\varphi}_{\text{po}} = \iota(\mathfrak{P})$ (as they show up neither in $\mathfrak{P}$ nor in ι), and we can choose (almost) arbitrary substitutions to get rid of them:

- Function or predicate skolem symbols can be replaced with constants (which exist, as they are needed by the rule (Ex)) or atoms like *true*
- Expression skolem symbols e_{Sk}, and the symbols $\mathit{Stmt}(e_{\text{Sk}})$, $\mathit{Dec}(\mathit{Stmt}(e_{\text{Sk}}))$ that are associated with e_{Sk} can be replaced with suitable literals γ, and the statements $\mathit{Norm}(\gamma)$, $\mathit{Norm}_{\text{sep}}(\gamma)$ provided by Sect. 4.6.5
- Analogously it is possible to replace statement skolem symbols s_{Sk}, $\mathit{Dec}(s_{\text{Sk}})$ with the statements $\mathit{Norm}(\alpha)$, $\mathit{Norm}_{\text{Dec}}(\alpha)$ that are provided in Sect. 4.6.5 for the neutral statement $\alpha = \textbf{skip}$.

The s-substitution σ is collision preventing regarding the formulas of $\tilde{H}$, in particular program variables of $\text{Cod}(\sigma)$ do not occur in $\tilde{H}$. It is possible and still necessary to rename bound symbols of σ suitably, to make the applications of σ collision free. This is achieved by defining an s-substitution σ_ξ for each formula ξ of $\tilde{H}$, mostly using Lem. A.36.[47]

- For formulas ξ that are introduced by the rule (Ax) and a formula $\mathfrak{Q} \in \mathbf{A}_{\text{Sk}} \backslash \mathbf{C}_{\text{Sk}}$ containing schema variables, i.e. $\xi = \eta(\mathfrak{Q})$, where

  $$\eta = \{\#s_1/\alpha_1, \ldots, \#s_k/\alpha_k\}$$

  is an instantiation of the schema variables of $\mathfrak{Q}$: We use Lem. A.36. Unfortunately, it could happen that σ is not collision preventing regarding an instantiation α_i (which is a premise of Lem. A.36), namely $\text{Cod}(\sigma)$ could contain a program variable v occurring in α_i, but not in ξ.[48] In this situation, we can observe that $\text{v} \in \mathit{PVar}_u \backslash \mathit{PVar}_e(\mathfrak{P})$ (this holds for all program variables of $\text{Cod}(\sigma)$), and that it is therefore possible to replace v by a "new" variable u, creating a new instantiation η' with $\eta'(\mathfrak{Q}) = \eta(\mathfrak{Q}) = \xi$ (as v did not occur in ξ, after all).

  We can thus assume that σ is collision preventing regarding $\mathbf{F} := \{\xi, \alpha_1, \ldots, \alpha_k\}$. By the definition of the statements $\mathit{Norm}(\alpha)$, $\mathit{Norm}_{\text{Dec}}(\alpha)$ that are substituted by σ for statement skolem symbols (or indirectly for expression symbols), σ is even compatible with $\mathbf{F}$ (see Def. A.34 for this term). The remaining premises of Lem. A.36 are implied by the definition of $\mathbf{A}$ ($\mathfrak{Q}$ does not contain skolem symbols of Sym_{Sk}), and Rem. 3.5.

  Lem. A.36 does then provide the s-substitution $\sigma_\xi := \omega$ we will use for ξ.

- For the first formula $\neg\tilde{\varphi}_{\text{po}} = \neg\iota(\mathfrak{P})$ of $\tilde{H}$, we construct $\sigma_{\neg\tilde{\varphi}_{\text{po}}}$ exactly the same way as in the first case, replacing $\mathfrak{Q}$ by $\mathfrak{P}$ and η by ι. This time, the conditions of Lem. A.36 hold because of the restrictions made regarding $\mathfrak{P}$.

[47] It would as well be possible to choose a single substitution ω for the whole proof, but renaming per formula seems to be the easier way.

[48] This case can be constructed using a ContextSV without the unique-property.

- For all other formulas ξ (e.g. created by the rule (Ex)), we use Lem. A.29 to obtain an s-substitution $\sigma_\xi := \omega$ that can be applied without collisions to ξ.

As it has been expected, we are now able to use the substitutions σ_ξ to define the sequence

$$\neg\varphi'_{\text{po}}, \psi'_1, \ldots, \psi'_k$$

by $\psi'_i := \sigma_{\psi_i}(\psi_i)$ and $\neg\varphi'_{\text{po}} := \sigma_{\neg\tilde{\varphi}_{\text{po}}}(\neg\tilde{\varphi}_{\text{po}})$ (as for FOL).

The Set $\{\neg\varphi'_{\text{po}}, \psi'_1, \ldots, \psi'_k\}$ is propositionally unsatisfiable
This follows exactly as for FOL, only using Lem. A.30 instead of Lem. A.14.

Construction of the Proof H_φ
Contrary to the situation in FOL, the formula φ'_{po} is actually quite different from the formula φ we are really interested in. This is our own fault, as we didn't just use the instantiations $\alpha = \mu(\#a)$ of StatementSV $\#a$ from φ to define the substitution σ, but "normalised" statements $\mathit{Norm}(\alpha)$ instead. Fortunately, in Sect. 4.6.5, as part of the definition of $\mathit{Norm}(\alpha)$, the possibility to replace α with $\mathit{Norm}(\alpha)$ was demanded. We will use this knowledge to construct the proof H_φ, satisfying

$$\{\neg\varphi, \zeta'_1, \ldots, \zeta'_n\} \vDash_0 \neg\varphi'_{\text{po}}.$$

To exchange the statements, let $\#s_1, \ldots, \#s_k$ be the StatementSV of $\mathfrak{P}$, and let $\alpha_i := \mu(\#s_i)$ for $i \in \{1, \ldots, k\}$ be their instantiations in φ. We construct a list $\mu_0, \ldots, \mu_k$ of instantiations with $\mu = \mu_0$ to replace α_i with $\mathit{Norm}(\alpha_i)$ successively, such that

$$\mu_0(\mathfrak{P}) = \varphi, \qquad \mu_k(\mathfrak{P}) =_{\text{br}} \varphi'_{\text{po}}$$

Therefore we refer to the statements $\mathit{Norm}_{\text{c}}(\alpha_1), \ldots, \mathit{Norm}_{\text{c}}(\alpha_k)$ that are derived in Sect. 4.6.5 from the statements $\mathit{Norm}(\alpha_1), \ldots, \mathit{Norm}(\alpha_k)$. The instantiations $\mu_0, \ldots, \mu_k$ are defined by

$$\begin{aligned} \mu_i(\#s_j) &= \mathit{Norm}_{\text{c}}(\alpha_j) && \text{for } j \le i \\ \mu_i(\#a) &= \mu(\#a) && \text{for all other } \#a, \end{aligned}$$

i.e. each μ_i instantiates the first i schema variables $\#s_1, \ldots, \#s_i$ with normalised statements, and all other schema variables with the original instantiations determined by μ.

The first requirement $\mu_0(\mathfrak{P}) = \varphi$ follows directly from $\mu_0 = \mu$. For the second one, $\mu_k(\mathfrak{P}) =_{\text{br}} \varphi'_{\text{po}}$, we will again use Lem. A.36. By the construction of $\sigma_{\neg\tilde{\varphi}_{\text{po}}} =: \omega$ and the lemma, the formula φ'_{po} is an instance of $\mathfrak{P}$:

$$\varphi'_{\text{po}} = \omega(\tilde{\varphi}_{\text{po}}) = \omega(\iota(\mathfrak{P})) \overset{(*)}{=} \iota_\omega(\mathfrak{P}), \qquad \text{with } \iota_\omega(\#a) := \omega(\iota(\#a)),$$

where $(*)$ uses Lem. A.36. To show that the two instances $\mu_k(\mathfrak{P})$ and $\iota_\omega(\mathfrak{P})$ are equal modulo bound renaming, we can thus consider the instantiations μ_k and ι_ω. Namely, it can be observed that the instantiations $\mu_k(\#a)$ and $\iota_\omega(\#a)$ of each a schema variable $\#a$ are equal modulo renaming:[49]

[49] We are implicitly using Lem. A.30.

- For a TermSV or FormulaSV $\#a$:

$$\omega(\iota(\#a)) = \omega(s^{\#a}_{\mathrm{Sk}}(x_1, \ldots; \mathrm{v}_1, \ldots)) =_{\mathrm{br}} \mu(\#a) = \mu_k(\#a)$$

- For a StatementSV $\#a = \#s_i$, using the observation $\sigma(\iota(\#s_i)) = \mathit{Norm}_{\mathrm{c}}(\alpha_i)$ that was made when defining σ:

$$\omega(\iota(\#s_i)) =_{\mathrm{br}} \mathit{Norm}_{\mathrm{c}}(\alpha_i) = \mu_k(\#s_i)$$

- For an ExpressionSV $\#a$:

$$\omega(\iota(\#a)) = \omega(s^{\#a}_{\mathrm{Sk}}(\mathrm{v}_1, \ldots)) =_{\mathrm{br}} \mu(\#a) = \mu_k(\#a)$$

- For other kinds of schema variables, by assumption and construction μ, μ_k and ι are equal.

By Lem. 2.3, and because instantiations of StatementSV respect scopes, this entails that the formulas $\mu_k(\mathfrak{P})$ and $\omega(\iota(\mathfrak{P})) = \varphi'_{\mathrm{po}}$ are also equal modulo renaming:

$$\big(\text{for each } \#a\text{:} \quad \mu_k(\#a) =_{\mathrm{br}} \iota_\omega(\#a)\big) \quad \Longrightarrow \quad \mu_k(\mathfrak{P}) =_{\mathrm{br}} \iota_\omega(\mathfrak{P}).$$

Finally, we refer to the assumption of Sect. 4.6.5, which gives us for $i \in \{1, \ldots, k\}$

$$\vdash_{\mathbf{HT_A}} \mu_{i-1}(\mathfrak{P}) \leftrightarrow \mu_i(\mathfrak{P}),$$

because each instance $\mu_i(\mathfrak{P})$ can be obtained by replacing occurrences of α_i within $\mu_{i-1}(\mathfrak{P})$ with $\mathit{Norm}(\alpha_i)$. Hence for suitable $\zeta'_1, \ldots, \zeta'_n$, that can be created by applications of $\mathbf{HT_A}$-rules, we have

$$\{\zeta'_{j_i}, \ldots, \zeta'_{j_{i+1}-1}\} \vDash_0 \mu_{i-1}(\mathfrak{P}) \leftrightarrow \mu_i(\mathfrak{P}).$$

Obviously the following derivation holds

$$\Big\{\neg\varphi, \Big(\mu_0(\mathfrak{P}) \leftrightarrow \mu_1(\mathfrak{P})\Big), \Big(\mu_1(\mathfrak{P}) \leftrightarrow \mu_2(\mathfrak{P})\Big), \ldots, \Big(\mu_{k-1}(\mathfrak{P}) \leftrightarrow \mu_k(\mathfrak{P})\Big)\Big\} \vDash_0 \neg\varphi'_{\mathrm{po}}$$

and by replacing each equivalence with its particular $\mathbf{HT_A}$-proof (using Lem. 4.14), renaming constants introduced by the rule (Ex) upon collisions, the proof H_φ emerges

$$\neg\varphi, \zeta'_1, \ldots, \zeta'_n.$$

The Formulas $\psi'_1, \ldots, \psi'_k$

The remaining proposition we need to show is that each formula $\psi'_i = \sigma_{\psi_i}(\psi_i)$ either results from a valid $\mathbf{HT_A}$-rule application, or can at least be proved using $\mathbf{HT_A}$ (the first case is actually subsumed by the second one, except for the rule (Ex)). Different situations may occur, again depending on the way a formula ψ_i was introduced in the original proof H:

- The rule (Ax) has been used, together with a formula $\mathfrak{Q}_{\mathrm{Sk}} \in \mathbf{A}_{\mathrm{Sk}} \backslash \mathbf{C}_{\mathrm{Sk}}$ containing schema variables, i.e. $\psi_i \in \mathfrak{Q}_{\mathrm{Sk}}$. Then Lem. A.36 tells us that $\psi_i' = \sigma_{\psi_i}(\psi_i)$ is also an instance of $\mathfrak{Q}_{\mathrm{Sk}}$, and as σ has been defined to replace all skolem symbols by constructs not containing skolem symbols, $\psi_i' \in \mathfrak{Q}$ is even an instance of the axiom $\mathfrak{Q} \in \mathbf{A}$ corresponding to $\mathfrak{Q}_{\mathrm{Sk}}$

- The rule (Ax) has been used, together with a formula $\mathfrak{Q} \in \mathbf{C}_{\mathrm{Sk}}$, which is a decomposition axiom either for statement symbols $\mathit{Statement}_{\mathrm{Sk}}$ or for expression symbols $\mathit{Expression}_{\mathrm{Sk}}$. Contrary to the situation in the first case, thereby we cannot apply Lem. A.36 directly, as $\mathfrak{Q}$ contains symbols of $\mathrm{Dom}(\sigma)$. Both possibilities (i.e. statement and expressions symbols) can be treated simultaneously: In any case we will construct an $\mathbf{HT_A}$-proof of ψ_i' by referring to the assumptions made in Sect. 4.6.5:

  Let $\mathfrak{Q} = \mathfrak{D}^{\diamond}_{s_{\mathrm{Sk}}}$ or $\mathfrak{Q} = \mathfrak{D}^{\square}_{s_{\mathrm{Sk}}}$, as defined in Sect. 4.6.3. The axiom $\mathfrak{Q}$ contains two distinguished skolem symbols, namely the symbol s_{Sk} itself, and the symbol to which s_{Sk} is "normalised" by $\mathfrak{Q}$. We define a set $\mathbf{S}$ that consists of these two symbols:

  – If $s_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ is a statement symbol, we have $\mathbf{S} := \{s_{\mathrm{Sk}}, \mathit{Dec}(s_{\mathrm{Sk}})\}$
  – Otherwise $s_{\mathrm{Sk}} \in \mathit{Expression}_{\mathrm{Sk}}$ is an expression symbol, and then we choose $\mathbf{S} := \{s_{\mathrm{Sk}}, \mathit{Stmt}(s_{\mathrm{Sk}})\}$.

  The set $\mathbf{T}$ is defined to consist of the remaining symbols of the domain of the s-substitution $\omega := \sigma_{\psi_i}$, which are $\mathbf{T} := \mathrm{Dom}(\omega) \backslash \mathbf{S}$. From Lem. A.27 and the definition of ω, it follows that ω can be decomposed into

  $$\omega = \omega|_{\mathbf{S}} \circ \omega|_{\mathbf{T}}.$$

  Contrary to σ, for $\omega|_{\mathbf{T}}$ Lem. A.36 is applicable, because the major obstacle, namely the symbols $\mathbf{S}$, have been removed. By this lemma, for a particular s-substitution $\pi =_{\mathrm{br}} \omega|_{\mathbf{T}}$ we therefore have $\pi(\psi_i) \in \mathfrak{Q}$.

  The only skolem symbols remaining in $\pi(\psi_i) =: \theta$ are the symbols of $\mathbf{S}$, and the s-substitution $\omega|_{\mathbf{S}}$ is exactly the substitution σ defined in Sect. 4.6.5 (either for statement or expression skolem symbols). By renaming bound program variables of $PVar_u$ in θ (see footnote 43 on page 70) and arbitrary bound symbols in $\omega|_{\mathbf{S}}$, it is then possible to choose a formula $\theta' =_{\mathrm{br}} \theta$ with $\theta' \in \mathfrak{Q}$, and an appropriate s-substitution $\nu =_{\mathrm{br}} \omega|_{\mathbf{S}}$, such that ν can be applied to θ' without collisions. Finally, by the assumption of Sect. 4.6.5, we have

  $$\vdash_{\mathbf{HT_A}} \nu(\theta').$$

  Repeated applications of Lem. A.30 reveal that $\nu(\theta')$ and $\psi_i' = \omega(\psi_i)$ are equal modulo bound renaming, and are therefore not distinguished by $\mathbf{HT_A}$:

  $$\vdash_{\mathbf{HT_A}} \psi_i'$$

- The rule (Ex) has been used: Then we can use exactly the reasoning as for FOL (in Sect. 4.4.2), if Lem. A.16 in this reasoning is replaced with Lem. A.32.

5. Proof Obligations that Treat Further Aspects of JavaCardDL

In Sect. 4.6, the only axioms $\mathbf{A}_{\mathrm{Sk}}$, $\mathbf{A}$ that were considered as elements of the proofs H (of the proof obligation φ_{po}) and H' (of an instance $\varphi \in \mathfrak{P}$ of the schematic formula $\mathfrak{P}$ to be proved) were those that can be formulated using schema variables. Axioms defined this way are insufficient even for FOL (where it is necessary to have a supplementary rule to apply object-level substitution operators), and for JavaCardDL several further rules, e.g. for updates are needed. We will give a concise discussion of some of these rules.

5.1. Further Kinds of Rules

First we describe the general method that is used to introduce new axioms $\mathfrak{R}$, for which the reasoning of Sect. 4.6 needs to be adapted. We assume that $\mathbf{A}$ and $\mathbf{A}_{\mathrm{Sk}}$ are sets of axioms for the logics JavaCardDL and JavaCardDL$_{\mathrm{Sk}}$ respectively, for which the lifting process of Sect. 4.6.6 can be performed. To add new schematic formulas $\mathfrak{R}$ to the sets $\mathbf{A}$, $\mathbf{A}_{\mathrm{Sk}}$, we are requiring in any case

- The sets $\mathbf{C}_{(\mathrm{Ex})}$ of constants for the $\mathbf{HT}$-rule (Ex) and $PVar_u$ of program variables fulfil the condition formulated in Def. 4.8 regarding $\{\mathfrak{R}\}$, i.e. the elements of $\mathbf{C}_{(\mathrm{Ex})}$ and $PVar_u$ can still be regarded as unused symbols.

After having extended the set $\mathbf{A}$ to a larger set $\tilde{\mathbf{A}} \supset \mathbf{A}$, it is necessary (at least usually) also to add new axioms to $\mathbf{A}_{\mathrm{Sk}}$ to achieve completeness. Beside the condition regarding skolem symbols for each schematic formula $\mathfrak{R}_{\mathrm{Sk}} \in \tilde{\mathbf{A}}_{\mathrm{Sk}} \backslash \mathbf{A}_{\mathrm{Sk}}$ that has already been given, we are demanding for the new set $\tilde{\mathbf{A}}_{\mathrm{Sk}} \supset \mathbf{A}_{\mathrm{Sk}}$:

- For a formula $\gamma \in \mathfrak{R}_{\mathrm{Sk}} \in \tilde{\mathbf{A}}_{\mathrm{Sk}} \backslash \mathbf{A}_{\mathrm{Sk}}$ and an s-substitution σ, which is collision preventing regarding γ and can also be applied to γ without collisions,[50] such that $\sigma(\gamma)$ does not contain skolem symbols of Sym_{Sk}:

  $$\vdash_{\mathbf{HT}_{\tilde{\mathbf{A}}}} \sigma(\gamma).$$

  A sufficient condition for this is $\sigma(\gamma) \in \mathfrak{R} \in \tilde{\mathbf{A}}$.

  This second requirement guarantees that an application of the rule $\mathfrak{R}_{\mathrm{Sk}} \in \tilde{\mathbf{A}}_{\mathrm{Sk}} \backslash \mathbf{A}_{\mathrm{Sk}}$ within the proof of a proof obligation φ_{po} can be lifted to a proof step of an instance $\varphi \in \mathfrak{P}$.

If these two requirements hold for the sets $\tilde{\mathbf{A}} \supset \mathbf{A}$, $\tilde{\mathbf{A}}_{\mathrm{Sk}} \backslash \mathbf{A}_{\mathrm{Sk}}$, then the argumentation of Sect. 4.6.6 can as well be performed, i.e. we also have

$$\vdash_{\tilde{\mathbf{A}}_{\mathrm{Sk}}} \varphi_{\mathrm{po}} \Longrightarrow \vdash_{\tilde{\mathbf{A}}} \mathfrak{P}$$

for a schematic formula $\mathfrak{P}$ as in Sect. 4.6 and the proof obligation φ_{po} of $\mathfrak{P}$.

[50] Collision prevention is introduced in Def. A.28, and is *not* implied by the fact that the application of σ is collision free.

5.1.1. Substitutions

It is necessary to have rules for applying substitutions which are expressed by the object-level substitution operator

$$\{x\ \ t\}T.$$

The formula introduced in Sect. 4.4.3 for FOL is hardly appropriate for JavaCardDL, as it does not prohibit non-rigid terms to pass modalities within T like updates and program blocks, semantically spoken. In the KeY system, there is in fact a second (object-level) substitution operator $\{\cdot\ \ \cdot\}_w$, which resolves the problem by identifying modalities and non-rigid terms, and uses an alternative method of application for this critical case ($\{\cdot\ \ \cdot\}_w$ can only be applied to formulas φ):

$$\{x\ \ t\}_w\varphi \leftrightarrow \exists x.(x \doteq t \wedge \varphi).$$

Like the application of the normal substitution operator, this method can be described by an axiom $\mathfrak{S}^w \in \tilde{\mathbf{A}}$ for JavaCardDL. To define a corresponding schematic formula $\mathfrak{S}^w_{\mathrm{Sk}}$ that can be added to $\mathbf{A}_{\mathrm{Sk}}$, and which satisfies the requirements formulated in the beginning of 5.1, it is necessary also to regard function and predicate skolem symbols $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}}$ as modalities. Namely, by an s-substitution σ these symbols could be replaced with formulas or terms containing modalities, e.g.

$$s_{\mathrm{Sk}}(t) \quad \stackrel{\sigma}{\mapsto} \quad \langle\alpha\rangle t \doteq 0, \qquad \text{with} \quad \sigma(s_{\mathrm{Sk}}(x)) = \langle\alpha\rangle x \doteq 0.$$

As the term t turns up within the scope of the modal operator $\langle\alpha\rangle$ in the resulting formula, the symbol s_{Sk} also has to be regarded as a modality (in the example affecting the term t).

5.1.2. Updates

As it has been done for substitutions, it is necessary to provide an axiom $\mathfrak{U}$ defining admissible transformations of updates

$$(\{\mathrm{v} := e\}\varphi) \leftrightarrow \varphi' \qquad \text{or} \qquad (\{\mathrm{v} := e\}t) \doteq t'$$

and simultaneous updates (which will not explicitly discussed in the following paragraphs, but to which the same arguments apply). To extend these transformations of updates to terms and formulas containing any of the skolem symbols of $\mathit{Sym}_{\mathrm{Sk}}$, it is again (as for substitutions) essential to consider the situations that can arise from the replacement of these symbols, as it is performed by an s-substitution σ:

- The substitution of function and predicate skolem symbols can introduce modalities, e.g.

  $$s_{\mathrm{Sk}}(t;\mathrm{v}) \qquad \stackrel{\sigma}{\mapsto} \qquad \langle\alpha\rangle t \doteq \mathrm{v}$$

  It is therefore not sound to let updates pass below any skolem symbol, only applying the update to the arguments of the symbol:

  $$\nvdash_{\tilde{\mathbf{A}}} \ \sigma\Big(\{\mathrm{v} := e\}s_{\mathrm{Sk}}(t;\mathrm{v}) \leftrightarrow s_{\mathrm{Sk}}(\{\mathrm{v} := e\}t; \{\mathrm{v} := e\}\mathrm{v})\Big)$$

- An s-substitution σ that replaces a skolem symbol s_{Sk} within a formula φ can introduce (free) program variables w that did originally not occur as arguments of s_{Sk}. As σ is collision preventing, these variables w do not occur in φ, however (see Def. A.28). As an example, we consider the formula

$$\varphi = \{\mathrm{w}\ \texttt{:=}\ e\}s_{\mathrm{Sk}}(\mathrm{v}).$$

Because φ contains the program variable w, we have $\mathrm{w} \notin \mathrm{Cod}(\sigma)$ for all relevant s-substitutions σ. Therefore it can be guaranteed that the program variable w is not introduced (as a free variable) upon the replacement of s_{Sk}. A suitable rule for update simplification could thus remove the update because of

$$\vdash_{\tilde{\mathbf{A}}} \ \sigma(\{\mathrm{w}\ \texttt{:=}\ e\}s_{\mathrm{Sk}}(\mathrm{v})) \leftrightarrow \sigma(s_{\mathrm{Sk}}(\mathrm{v}))$$

where σ is a collision preventing s-substitution.

5.2. Further Types of Schema Variables

The kinds of schema variables described in Sect. 3.2 are not sufficient for a complete treatment of JavaCardDL: Practically and in the KeY system several additional types are defined, usually to be used for very specific tasks. The procedure to allow axioms which are defined referring to such additional types within the sets $\mathbf{A}$ and $\mathbf{A}_{\mathrm{Sk}}$ essentially consists of an according enhancement of Lem. A.36, which is the most important tool in Sect. 4.6.6 for modification of proofs.

5.3. Contexts

The attentive reader has noticed that discrepancies exist between the definition of meaning formulas for taclets (Sect. 3.3.2) and the requirements made regarding those schematic formulas that can be proved, and those that may be used as axioms within those proofs. Namely, in Sect. 4.6 we demand for ContextSV that either the unique-flag is false, or that there are no such variables at all. We will give some simple lemmas to resolve these problems, showing that the restrictions of Sect. 4.6 pose no loss of generality. All propositions of this section refer to the schema variables defined in 3.2.

The first lemma shows that it is possible to remove the unique-flag from ContextSV in many cases, in particular for rewrite taclets that only have one `replacewith`-statement:

5.1 *Lemma (Non-unique Contexts):* Let $\mathbf{A}$ be a set of schematic formulas, and

$$\mathfrak{P} := \quad \#ct(T_1) \rightarrow \#ct(T_2) \quad \in \mathbf{A}$$

where $\#ct$ is a ContextSV with the unique-flag set to true, and T_1, T_2 do not contain free PVariableSV for which the unusedOnly-flag is true. If $\#ct'$ is a new ContextSV with the same properties as $\#ct$ except for the unique-flag (which is false for $\#ct'$), and $\tilde{\mathfrak{P}}$ is obtained from $\mathfrak{P}$ by replacing $\#ct$ with $\#ct'$, then

$$\vdash_{\mathbf{HT_A}} \tilde{\mathfrak{P}}. \qquad *$$

Proof: By applying $\mathfrak{P}$ repeatedly with different contexts. Suppose that ι is an instantiation of $\tilde{\mathfrak{P}}$, and we have $\psi' := \iota(\#ct')$ (the formula ψ' can contain multiple insertion points $a_{\#ct'}$). If $T_1^\iota := \iota(T_1)$, $T_2^\iota := \iota(T_2)$ are the instances of T_1, T_2 respectively, and we write

$$\psi(T^\iota) := J_{\mathrm{ContextSV}}(\psi, T^\iota)$$

for the instantiation of a ContextSV, then we have

$$\{\psi_1(T_1^\iota) \to \psi_1(T_2^\iota),\ \psi_2(T_1^\iota) \to \psi_2(T_2^\iota),\ \ldots,\ \psi_l(T_1^\iota) \to \psi_l(T_2^\iota)\} \vDash_0 \psi'(T_1^\iota) \to \psi'(T_2^\iota)$$

where the formulas $\psi_1, \ldots, \psi_l$ are chosen such that

$$\psi'(T_1^\iota) =_{\mathrm{br}} \psi_1(T_1^\iota),\ \psi_1(T_2^\iota) =_{\mathrm{br}} \psi_2(T_1^\iota),\ \ldots,\ \psi_l(T_2^\iota) =_{\mathrm{br}} \psi'(T_2^\iota).$$

This can be achieved by an appropriate (successive) replacement of insertion points $a_{\#ct'}$ within ψ' either with T_1^ι or with T_2^ι. This replacement can in general only be performed if T_1, T_2 do not contain free PVariableSV for which the unusedOnly-flag is true, because (by the definition of PVariableSV) instantiations of such schema variables must not occur within the instantiation of the ContextSV $\#ct$. □

5.2 *Remark:* The lemma is also applicable for formulas $\mathfrak{P} = \#ct(T_1) \leftrightarrow \#ct(T_2)$, and the proposition of the lemma does also hold for formulas $\mathfrak{P}'$ containing $\mathfrak{P}$ as a sub-formula with positive polarity (such that propositional junctors are the only operators occurring above $\mathfrak{P}$), provided that $\#ct$ only occurs in $\mathfrak{P}$. ∗

Presumed that **A** contains axioms to handle equations (and equivalences, which can be treated exactly the same way, but are not mentioned explicitly in the next paragraphs), namely the formulas

$$\begin{aligned}\mathfrak{E}_1 &= \#ct(\#t \doteq \#t) \leftrightarrow \#ct(true)\\ \mathfrak{E}_2 &= \#t_1 \doteq \#t_2 \to (\#ct(\#t_1) \leftrightarrow \#ct(\#t_2))\end{aligned}$$

(where at least for $\mathfrak{E}_2$ the SUL-flag of the variable $\#ct$ has to be true) then it is possible to generalise Lem. 5.1, making it applicable also for meaning formulas of taclets with multiple `replacewith`-statements. For that, we can first observe that Lem. 5.1 can be applied to $\mathfrak{E}_1$, $\mathfrak{E}_2$ (by Rem. 5.2). Hence the variable $\#ct$ in these formulas does not need to have the unique-flag being true.

5.3 *Lemma (Non-unique Contexts II):* Let **A** be a set of schematic formulas that contains $\mathfrak{E}_1, \mathfrak{E}_2$, and $\mathfrak{P}$ a schematic formula, defined through schema variables, in which the subformulas $\mathfrak{Q}_1, \ldots, \mathfrak{Q}_k$ occur with positive polarity and only below propositional junctors. Suppose that these formulas are given by

$$\mathfrak{Q}_i = \#ct(t) \leftrightarrow \#ct(t_i)$$

such that

- $\#ct$ is a ContextSV, for which the unique- and SUL-flags are true, and that does only occur in the formulas $\mathfrak{Q}_1, \ldots, \mathfrak{Q}_k$ (and not within $t, t_1, \ldots, t_k$)

- t does not contain free PVariableSV, for which the unusedOnly-flag is true
- $\#ct$ does not occur within the value of any prefix-property of the schema variables of $\mathfrak{P}$.

If $\#ct'$ (as in Lem. 5.1) is a new ContextSV with the same properties as $\#ct$ except for the unique-flag (which is false for $\#ct'$), and $\tilde{\mathfrak{P}}$ is obtained from $\mathfrak{P}$ by replacing $\#ct$ with $\#ct'$, then

$$\vdash_{\mathbf{HT_A}} \tilde{\mathfrak{P}}. \qquad *$$

Proof: We consider an arbitrary instantiation ι of the schema variables of $\mathfrak{P}'$, and construct an instantiation κ for $\mathfrak{P}$ by

$$\kappa(\#s) := \begin{cases} t' \doteq a_{\#ct} & \text{for } \#s = \#ct \\ \iota(\#s) & \text{otherwise} \end{cases}$$

where $t' =_{\text{br}} \iota(t)$ is chosen such that κ is a valid instantiation (i.e. bound symbols of t' do not occur as instantiations of other schema variables; this can be enforced by bound renaming).

Then we have

$$\kappa(\mathfrak{Q}_i) = (t' \doteq \iota(t)) \leftrightarrow (t' \doteq \iota(t_i))$$

and using instances of $\mathfrak{E}_1$ and $\mathfrak{E}_2$ (we write $\psi(T) := J_{\text{ContextSV}}(\iota(\#ct'), T)$)

$$\{\kappa(\mathfrak{Q}_i),\ \underbrace{t' \doteq t' \leftrightarrow \mathit{true}}_{\in \mathfrak{E}_1},\ \underbrace{t' \doteq \iota(t_i) \to \big(\psi(t') \leftrightarrow \psi(\iota(t_i))\big)}_{\in \mathfrak{E}_2}\} \vDash_0 \underbrace{\psi(\iota(t)) \leftrightarrow \psi(\iota(t_i))}_{=\iota(\mathfrak{Q}_i)}.$$

As the formulas $\mathfrak{Q}_i$ have positive polarity within $\mathfrak{P}$ and occur only below propositional junctors, it is then possible to "replace" $\kappa(\mathfrak{Q}_i)$ with $\iota(\mathfrak{Q}_i)$ within $\kappa(\mathfrak{P})$ by propositional transformations, and thus

$$\vdash_{\mathbf{HT_A}} \iota(\tilde{\mathfrak{P}}) \qquad \square$$

5.4 *Example:* Consider the taclet t_6

$$\begin{array}{lll} \texttt{find}(\#t) & \texttt{replacewith}(0) & \texttt{add}(\#t \doteq 0 \vdash\ ); \\ & \texttt{replacewith}(\#t) & \texttt{add}(\ \vdash \#t \doteq 0) \end{array}$$

which has the meaning formula (Sect. 3.3.2)

$$\mathfrak{M}(t_6) = \Big(\big(\#ct(\#t) \leftrightarrow \#ct(0)\big) \land \#t \doteq 0\Big) \lor \Big(\big(\#ct(\#t) \leftrightarrow \#ct(\#t)\big) \land \neg\#t \doteq 0\Big).$$

By Lem. 5.3, the value of the unique-flag of the variable $\#ct$ in this formula has no influence on the validity of the formula. $*$

The axiom $\mathfrak{E}_2$ can also be used to eliminate ContextSV $\#ct$ from meaning formulas of taclets, provided that the SUL-flag of these variables is true, and that $\#ct$ does not occur within prefix-properties.[51] By the same reasoning as in the proof of Lem. 5.3, it follows that in this case sub-formulas

$$\#ct(A) \leftrightarrow \#ct(B_i)$$

(as they occur in meaning formulas) can be replaced with simple equations or equivalences

$$A \doteq B_i, \qquad A \leftrightarrow B_i.$$

Finally, for contexts $\#ct$ for which the SUL-flag is false, and for meaning formulas of taclets that only possess a `find`- and one `replacewith`-statement, the following equivalence can be used:

$$\models\ A \doteq B \quad \Longleftrightarrow \quad \models\ \#ct(A) \leftrightarrow \#ct(B).$$

[51] Otherwise it is nevertheless possible to use a semantic argument, namely that free variables can always be replaced with constants, to remove $\#ct$ in such situations.

6. Conclusion

In Sect. 4 we have described two construction procedures for proof obligations of taclets, which treat first-order logic and JavaCardDL, respectively. We have shown that for both of the methods the derivability of the proof obligation implies the reproducibility of the corresponding taclet.

In any case, the procedure which ensures the soundness of a taclet essentially consists of the following three steps:

1. The meaning formula $\mathfrak{P}$ of a taclet t is instantiated with skolem constants, functions, predicates and programs. The resulting formula φ_{po} is the *proof obligation* of the taclet t. This step is computable and can be performed automatically.

2. A proof of the formula φ_{po} has to be furnished using a certain calculus. This step has to be performed either interactively by the user, or by an automated prover.

3. The reproducibility of the taclet t is shown, by constructing sequences of rule applications (using rules that are known to be sound) that have the same effect as applications of t. This step has to be performed only once for a given procedure for the creation of proof obligations φ_{po}, and the necessary reasoning for the two procedures that are introduced in this thesis are contained by Sect. 4.

As part of the minor thesis, the method to compute proof obligations (Step 1) has been implemented for JavaCardDL and for the prover of the KeY system. This implementation differs from the considerations in this document regarding some details, however:

- In the KeY system, the definitions of the schema variables are not completely equivalent to the definitions given in this document

- The KeY system provides (a lot) more kinds of schema variables than treated in this thesis. While also not handling all available schema variable types, the implementation still addresses more types than this document.

Both implementation and document cannot be regarded as complete treatments of taclets, as they exist in the KeY system. It has therefore been tried to make the reasoning of Sect. 4, as well as the implementation as modular as possible to allow extensions that cover further aspects.

A. Substitutions

A.1. Substitution of Functions and Predicates

In this section, we define a family of substitutions that do not map variables to terms, as usually, but instead function and predicate symbols to terms and formulas, replacing formal parameters with terms given as arguments (i.e. "call-by-name"). The substitutions can be seen as higher-order substitutions that replace function and predicate variables (the arguments of symbols that are replaced are only terms, i.e. only first-order, however). It is always possible to reduce normal substitutions of (free) logical variables to the substitutions defined in this section, by replacing free variables with constants.

A.1 Example: To the formula

$$\forall z.p(f(a, h(z)))$$

we apply a substitution that replaces the binary function f with the term $g(h(y), x)$, whereby x and y are be the formal parameters and represent the first and second argument of f respectively. The result is the formula

$$\forall z.p(g(h(h(z)), a)).$$

A.2 Definition (Free Symbols): For a term or formula T, with $FS(T)$ we denote the set of function or predicate symbols as well as free variables occurring in T. $*$

A.3 Example: For the term from the example A.1 we have

$$FS(g(h(y), x)) = \{x, y, g, h\}.$$

To distinguish ordinary substitutions of logical variables from the new kind of substitution we are going to introduce, we will call the former ones *v-Substitutions* within the whole appendix A (substitutions of variables).

A.4 Definition (f-Substitution): An *f-substitution* σ (substitution of functions) is a map

$$\sigma : \mathit{Term} \cup \mathit{For} \to \mathit{Term} \cup \mathit{For}$$

such that there is a map[52]

$$\sigma_{\text{cont}} : \mathit{Func} \cup \mathit{Pred} \to (\mathit{Term} \cup \mathit{For}) \times \mathit{Var}^*$$

(the "content" of σ) satisfying

- each function symbol $f \in \mathit{Func}$ with signature $(S_1, \ldots, S_k) \to S$ is mapped onto a pair $(t, \langle x_1, \ldots, x_k\rangle)$, where t is a term with sort S and $x_1, \ldots, x_k$ are distinct variables (the formal parameters), whose sorts are $S_1, \ldots, S_k$ respectively
- analogously, each predicate symbol $p \in \mathit{Pred}$ is mapped onto a pair $(\varphi, \langle x_1, \ldots, x_k\rangle)$, where φ is a formula

[52]By Var^* we denote the set of finite sequences over Var.

- for almost all symbols s, $\sigma_{\text{cont}}(s) = (s(x_1, \ldots, x_k), \langle x_1, \ldots, x_k \rangle)$ (i.e. essentially s is mapped onto itself)
- σ is the application of σ_{cont} to terms and formulas:
  - For function or predicate symbols s:
  $$\sigma(s(r_1, \ldots, r_k)) = \{x_1/\sigma(r_1), \ldots, x_k/\sigma(r_k)\}T \quad \text{where} \quad \sigma_{\text{cont}}(s) = (T, \langle x_1, \ldots, x_k \rangle)$$
  (the substitution in this formula is a v-substitution of logical variables).
  - All other constructs (like variables, quantifiers, ...) are handled faithfully, i.e. as a homomorphism.

Conversely, for each map σ_{cont} satisfying the conditions of Def. A.4, a corresponding map σ can be constructed inductively.

A.5 Example (Example A.1 continued): For the f-substitution σ applied in example A.1, we have
$$\sigma_{\text{cont}}(f) = (g(h(y), x), \langle x, y \rangle).$$

A.6 Remark (Notation): For nullary function or predicate symbols $s_1, \ldots, s_l$ we will use the same notation for f-substitutions as for v-substitutions:
$$\{s_1/t_1, \ldots, s_l/t_l\}$$

A.7 Definition (Domain and Codomain): Let σ, σ_{cont} be as in Def. A.4. As usually (see for example [SA94]) we define the *domain* $\text{Dom}(\sigma)$ of σ to be the set of function or predicate symbols not mapped onto themselves, and the *codomain* $\text{Cod}(\sigma)$ of σ by
$$\text{Cod}(\sigma) = \bigcup_{\substack{s \in \text{Dom}(\sigma) \\ (a, \langle x_1, \ldots, x_k \rangle) = \sigma_{\text{cont}}(s)}} FS(a) \setminus \{x_1, \ldots, x_k\}.$$

A.8 Lemma (Free Variables and f-Substitution): Let σ, σ_{cont} be as in Def. A.4, and let them satisfy the following equivalent conditions:

1. For each symbol s with $\sigma_{\text{cont}}(s) = (a, \langle x_1, \ldots, x_k \rangle)$: $FV(a) \subset \{x_1, \ldots, x_k\}$
2. $\text{Cod}(\sigma) \cap \mathit{Var} = \emptyset$

Then for each formula or term T: $FV(\sigma(T)) \subset FV(T)$. $*$

A.9 Definition (Bound Renaming of f-Substitutions): Let σ_{cont} and ω_{cont} be contents as in Def. A.4. σ_{cont} and ω_{cont} are called *equal modulo bound renaming*, if for each symbol s with $\sigma_{\text{cont}}(s) = (a, \langle x_1, \ldots, x_k \rangle)$, $\omega_{\text{cont}}(s) = (b, \langle y_1, \ldots, y_k \rangle)$ the formulas (or terms)
$$\{x_1/z_1, \ldots, x_k/z_k\}a \quad \text{and} \quad \{y_1/z_1, \ldots, y_k/z_k\}b$$
are equal modulo bound renaming. Thereby $z_1, \ldots, z_k$ are distinct variables not occurring in a or b, which have the same sorts as $x_1, \ldots, x_k$, and the applied substitutions are v-substitutions.

f-Substitutions σ and ω are called *equal modulo bound renaming*, if there are contents σ_{cont} and ω_{cont} as in Def. A.4 which are equal modulo bound renaming. $*$

A.10 Definition (Collisions): Let σ, σ_{cont} be as in Def. A.4. The application of σ to a term or formula T is called *collision free*, if for each symbol s with $\sigma_{\text{cont}}(s) = (a, \langle x_1, \ldots, x_k \rangle)$

- No element of $FV(a) \backslash \{x_1, \ldots, x_k\}$ is bound at an occurrence of s in T
- The v-substitution of the variables $x_1, \ldots, x_k$ in Def. A.4 is collision free for any occurrence of s in T.

A.11 Example: We demonstrate the two kinds of collisions for the f-substitution σ, which is determined by

$$\sigma_{\text{cont}}(p) = (\forall y.r(x, z), \langle x \rangle), \qquad \text{Dom}(\sigma) = \{p\}, \ \ \text{Cod}(\sigma) = \{r, z\}$$

where p (resp. r) is a unary (resp. binary) predicate, and x, y, z are logical variables. An example for the first type of collision in Def. A.10 is the application

$$\sigma(\exists z.p(a)) = \exists z.\forall y.r(a, z)$$

in which the binding of the variable z is altered. The second kind of collision occurs in

$$\sigma(p(y)) = \forall y.r(y, z)$$

as the free variable y of $p(y)$ is moved into the scope of $\forall y$.

It can be observed that if the application of an f-substitution σ to T is collision free, then the application of σ to any sub-term or sub-formula of T will not cause collisions either. $*$

A.12 Lemma (Collision Free f-Substitutions): Let σ be an f-substitution for which the conditions of Lem. A.8 hold, and T a term or formula. Then there exists an f-substitution ω, that is equal to σ modulo bound renaming, such that the application of ω to T is collision free. $*$

A.13 Remark: In general (i.e. if the conditions of Lem. A.8 are not fulfilled) Lem. A.12 is wrong, as it is usually also necessary to rename bound variables within T. A simple example is the following application of a f-substitution:

$$\{a/x\}\forall x.p(a).$$

A.14 Lemma (Renaming Everything): Let S and T be terms or formulas which are equal modulo bound renaming, and σ and ω be f-substitutions which are also equal modulo bound renaming. If the applications of σ to S and of ω to T are collision free, then $\sigma(S)$ and $\omega(T)$ are equal modulo bound renaming. $*$

f-Substitutions that only replace *nullary* function and predicate symbols mostly behave like v-substitutions of logical variables. In fact the only difference is that the elements of the domain of an f-substitution cannot be bound by operators, contrary to logical variables. For the following two lemmas we will call an f-substitution of nullary symbols, i.e. an f-substitution σ with

$$\text{for each } s \in \text{Dom}(\sigma): \quad s \text{ is nullary}$$

a *nullary f-substitution*. Because of the similarity of nullary f-substitutions and v-substitutions, the next lemmas hold for both likewise.

A.15 Lemma (Concatenation I): Let σ be a v-substitution or a nullary f-substitution, T a term or formula, $y_1, \ldots, y_l$ either distinct variables or distinct nullary function or predicate symbols, and $\omega = \{y_1/t_1, \ldots, y_l/t_l\}$ a v-substitution or a nullary f-substitution. Provided that

- the application of ω to T does not cause collisions and
- $FS(T) \cap \mathrm{Dom}(\sigma) \subset \{y_1, \ldots, y_l\}$

we have

$$\sigma(\omega(T)) = \nu(T) \quad \text{where} \quad \nu = \{y_1/\sigma(t_1), \ldots, y_l/\sigma(t_l)\}.$$

Lem. A.15 does not hold if the application of ω leads to collisions, in general. A counter example is

$$\{x/c\}\{y/x\}\forall x.p(y) \neq \{y/c\}\forall x.p(y).$$

Proof: First we can observe that both for v-substitutions and nullary f-substitutions κ_1, κ_2 we have

$$\kappa_1(\varphi) = \kappa_2(\varphi) \quad \Longleftrightarrow \quad \text{for each } s \in FS(\varphi): \ \ \kappa_1(s) = \kappa_2(s).^{53} \tag{8}$$

This equivalence does not hold for f-substitutions that also replace symbols which are not nullary, however. Information about the property for v-substitutions, which also applies to nullary f-substitutions can be found in [Fit96].

Suppose σ and ω are as in Lem. A.15. We show the conjecture by structural induction over $T = op(r_1, \ldots, r_n)$:

- For $op = y_i \in \{y_1, \ldots, y_l\}$:

  $$\sigma(\omega(T)) = \sigma(t_i) = \nu(y_i) = \nu(T)$$

- For $op \notin \{y_1, \ldots, y_l\} \supset \mathrm{Dom}(\omega)$: Then we also have $op \notin \mathrm{Dom}(\sigma)$, because otherwise $op \in FS(T)$ would violate the premise of the lemma.

  $$\begin{aligned} \sigma(\omega(T)) &= \sigma(\omega(op(r_1, \ldots, r_n))) \\ &= \sigma(op(\omega_1(r_1), \ldots, \omega_n(r_n))) && op \text{ may bind variables} \\ &= op(\sigma_1(\omega_1(r_1)), \ldots, \sigma_n(\omega_n(r_n))) \\ &= op(\nu_1(r_1), \ldots, \nu_n(r_n)) && \text{Claim (i)} \\ &= \nu(op(r_1, \ldots, r_n)) \end{aligned}$$

  In this derivation the substitutions $\sigma_i, \omega_i, \nu_i$ arise by the restriction of σ, ω, ν to symbols not bound by op within the ith sub-term or sub-formula.

  To prove claim (i), we show the equation

  $$\sigma_i(\omega_i(r_i)) = \nu_i(r_i)$$

[53] We assume that for symbols $s \in FS(\varphi)$, for which κ_1, κ_2 are not defined as maps, we have $\kappa_1(s) = \kappa_2(s) = \perp$. This applies to function and predicate symbols occurring in φ that are not nullary, but that are also contained by $FS(\varphi)$ (see Def. A.2).

for each $i \in \{1, \ldots, n\}$. $\mathbf{B}$ shall be the set of symbols that are bound by *op* at the ith argument. Without loss of generality we have

$$\mathbf{B} \cap \{y_1, \ldots, y_l\} = \{y_1, \ldots, y_{e-1}\}, \quad \text{with} \quad e \in \{1, \ldots, l+1\},$$

i.e. *op* binds (exactly) the first $e-1$ variables of $\{y_1, \ldots, y_n\} \supset \mathrm{Dom}(\omega)$ (if ω is an f-substitution, than e will be 1 as *op* does not bind any constants).

We can apply the induction hypothesis at this point, as the presumptions of Lem. A.15 are fulfilled: The application of ω_i to r_i is collision free, and

$$FS(r_i) \cap \mathrm{Dom}(\sigma_i) \quad \subset \quad \big(FS(T) \cup \mathbf{B}\big) \cap \big(\mathrm{Dom}(\sigma) \backslash \mathbf{B}\big) \quad \subset \quad \{y_e, \ldots, y_l\}.$$

This leads to

$$\begin{aligned}
\sigma_i(\omega_i(r_i)) &= \sigma_i(\{y_e/t_e, \ldots, y_l/t_l\}r_i) \\
&= \{y_e/\sigma_i(t_e), \ldots, y_l/\sigma_i(t_l)\}r_i && \text{by IH} \\
&= \{y_e/\sigma(t_e), \ldots, y_l/\sigma(t_l)\}r_i && \text{Claim (ii)} \\
&= \nu_i(r_i)
\end{aligned}$$

To show claim (ii), we apply equivalence (8), i.e. we have to prove that for each $y_j \in FS(r_i)$ with $j \in \{e, \ldots, l\}$: $\sigma_i(t_j) = \sigma(t_j)$. A second application of (8) reveals that this equation is equivalent to

$$\text{for each } s \in FS(t_j): \quad \sigma_i(s) = \sigma(s).$$

This proposition finally is fulfilled, because σ_i, σ are equal except for the elements of $\mathbf{B}$, and we have $\mathbf{B} \cap FS(t_j) = \emptyset$. Namely, $y_j \in FS(r_i)$ occurs free in r_i. The existence of $s \in \mathbf{B} \cap FS(t_j)$ would then cause a collision upon the application of ω to T, which is precluded by assumption.

A.16 Lemma (Concatenation II): Let σ be an f-substitution, T a term or formula, $y_1, \ldots, y_l$ either distinct variables or distinct nullary function or predicate symbols with

$$(\mathrm{Dom}(\sigma) \cup \mathrm{Cod}(\sigma)) \cap \{y_1, \ldots, y_l\} = \emptyset$$

and $\omega = \{y_1/t_1, \ldots, y_l/t_l\}$ a v-substitution or a nullary f-substitution. If the application of σ to T is collision free, then the following holds:

$$\sigma(\omega(T)) = \nu(\sigma(T)) \quad \text{where} \quad \nu = \{y_1/\sigma(t_1), \ldots, y_l/\sigma(t_l)\}.$$

Proof: This is proved by structural induction over $T = op(r_1, \ldots, r_n)$. For that, let σ, σ_{cont} as in Def. A.4.

- If $op \in \mathrm{Dom}(\sigma)$: Let $(b, \langle x_1, \ldots, x_n \rangle) = \sigma_{\mathrm{cont}}(op)$.

$$\begin{aligned}
\nu(\sigma(T)) &= \nu(\{x_1/\sigma(r_1), \ldots, x_n/\sigma(r_n)\}b) \\
&= \{x_1/\nu(\sigma(r_1)), \ldots, x_n/\nu(\sigma(r_n))\}b && \text{by Lem. A.15} \\
&= \{x_1/\sigma(\omega(r_1)), \ldots, x_n/\sigma(\omega(r_n))\}b && \text{by IH} \\
&= \sigma(op(\omega(r_1), \ldots, \omega(r_n))) \\
&= \sigma(\omega(T)) && \text{as } op \notin \{y_1, \ldots, y_l\}
\end{aligned}$$

It is possible to apply Lem. A.15 in this derivation, as by assumption we have

$$\begin{aligned} &(FS(b)\backslash\{x_1,\ldots,x_n\}) \cap \{y_1,\ldots,y_l\} = \emptyset \\ \iff \quad &FS(b) \cap \{y_1,\ldots,y_l\} \subset \{x_1,\ldots,x_n\} \end{aligned}$$

and furthermore the application of $\{x_1/\sigma(r_1),\ldots,x_n/\sigma(r_n)\}$ to b is collision free.

- If $op = y_i$: Then $n = 0$ and

$$\begin{aligned} \nu(\sigma(T)) &= \nu(y_i) && \text{as } y_i \notin \mathrm{Dom}(\sigma) \\ &= \sigma(t_i) = \sigma(\omega(y_i)) \end{aligned}$$

- Otherwise (op is something else)

$$\begin{aligned} \nu(\sigma(T)) &= \nu(op(\sigma(r_1),\ldots,\sigma(r_n))) \\ &= op(\nu_1(\sigma(r_1)),\ldots,\nu_n(\sigma(r_n))) && op \text{ may bind variables} \\ &= op(\sigma(\omega_1(r_1)),\ldots,\sigma(\omega_n(r_n))) && \text{by IH} \\ &= \sigma(\omega(T)) \end{aligned}$$

A.17 *Example:* In Sect. 4.4.2, the previous lemma is used to prove that applications of the **HT**-rule (Ex) stay valid upon the replacement of skolem symbols, which are used to formulate proof obligations for taclets, with concrete terms or formulas. This replacement is performed by an f-substitution. For an illustration, consider the f-substitution σ given by

$$\sigma_{\text{cont}}(p_{\text{Sk}}) = (\forall y.q(z,y), \langle z\rangle), \qquad \sigma_{\text{cont}}(f_{\text{Sk}}) = (\{y \quad d\}g(z), \langle z\rangle)$$

that also contains an object-level substitution operator that binds the logical variable y. In Sect. 4.4.2, the following application of σ could arise, in which $\{x/c\}$ is the (meta-level) v-substitution of the logical variable x:

$$\sigma\big(\{x/c\}p_{\text{Sk}}(f_{\text{Sk}}(x))\big).$$

Because the application of σ to $p_{\text{Sk}}(f_{\text{Sk}}(x))$ is collision free, and we also have

$$c \notin \mathrm{Dom}(\sigma) = \{p_{\text{Sk}}, f_{\text{Sk}}\}, \qquad \mathrm{Cod}(\sigma) = \{q,d,g\},$$

Lem. A.16 tells us that

$$\sigma\big(\{x/c\}p_{\text{Sk}}(f_{\text{Sk}}(x))\big) = \{x/c\}\sigma\big(p_{\text{Sk}}(f_{\text{Sk}}(x))\big)$$

which is verified by

$$\begin{aligned} \sigma\big(\{x/c\}p_{\text{Sk}}(f_{\text{Sk}}(x))\big) &= \sigma\big(p_{\text{Sk}}(f_{\text{Sk}}(c))\big) = \forall y.q\big(\{y \quad d\}g(c), y\big) \\ \{x/c\}\sigma\big(p_{\text{Sk}}(f_{\text{Sk}}(x))\big) &= \{x/c\}\forall y.q\big(\{y \quad d\}g(x), y\big) = \forall y.q\big(\{y \quad d\}g(c), y\big). \end{aligned}$$

A.1.1. f-Substitutions and Schema Variables

The following lemma associates schema variable instantiations for FOL (as in Sect. 3.2.1) and f-substitutions. We will show that for certain schema variable instantiations ι and f-substitutions ω, there is a second instantiation κ making the following diagram commutative:

$$\begin{tikzcd}
& \varphi \arrow[dl, "\iota"'] \arrow[dr, "\kappa"] & \\
\iota(\varphi) \arrow[rr, "\omega"'] & & \omega(\iota(\varphi)) = \kappa(\varphi)
\end{tikzcd}$$

The lemma depends on the possibility to have contexts $\#ct$ that contain more than one "insertion point" $a_{\#ct}$ (i.e. the unique-flag is false). Namely, if the symbol $a_{\#ct}$ occurs as argument of a function or predicate symbol f in an instantiation $\iota(\#ct)$ (in $\iota(\varphi)$), replacing f by a term or a formula could lead to multiple occurrences of $a_{\#ct}$ (or none). To make the set $\mathbf{R}$ of valid instantiations closed under f-substitutions, it is thus necessary to reset the unique-flag.

A.18 *Lemma (Schema Variables and f-Substitutions):* Let φ be a formula or term containing VariableSV, TermSV, FormulaSV and ContextSV $\#s_1, \dots, \#s_k$, such that for each occurring ContextSV the unique-flag is false. Let $\iota = \{\#s_1/\alpha_1, \dots, \#s_k/\alpha_k\}$ be a valid instantiation, and σ be an f-substitution

- satisfying the conditions of Lem. A.8
- for each function or predicate symbol s of φ: $s \notin \mathrm{Dom}(\sigma)$
- for each ContextSV $\#s_i = \#ct_i$: $a_{\#ct_i} \notin \mathrm{Dom}(\sigma) \cup \mathrm{Cod}(\sigma)$.

Then there is an f-substitution ω, equal to σ modulo bound renaming, with

1. $\kappa = \{\#s_1/\omega(\alpha_1), \dots, \#s_k/\omega(\alpha_k)\}$ is a valid instantiation, in which the applications of ω are collision free
2. The following equation, in which the application of ω is collision free, holds:

$$\kappa(\varphi) = \omega(\iota(\varphi))$$

Proof: We replace all bound variables within σ with distinct ones not occurring in σ, α_i or φ and get the f-substitution ω. Then collisions may not occur as ω and α_i, φ do not contain common variables, and both claims are fulfilled:

1. - Instantiations of VariableSV are not modified and do not occur bound within any instantiation $\omega(\alpha_i)$, as they do not occur bound in α_i and cannot be introduced (bound) by ω

- Lem. A.8 guarantees that $FV(\omega(\alpha_i)) \subset FV(\alpha_i)$ for TermSV and FormulaSV instantiations. Moreover, for ContextSV $\#s_j = \#ct_j$ we have

$$\mathrm{Bound}(\alpha_j) \subset \mathrm{Bound}(\omega(\alpha_j))$$

as the arguments of functions or predicates do not leave the scopes of operators binding logical variables when applying ω. Altogether, prefix conditions are still fulfilled with κ.

2. The equation is proved by a structural induction:

- For a VariableSV, TermSV or FormulaSV $\#s_i$:

$$\kappa(\#s_i) = \omega(\alpha_i) = \omega(\iota(\#s_i))$$

- For a ContextSV $\#s_i = \#ct_i$:

$$\kappa(\#ct_i(r)) = \{a_{\#ct_i}/\kappa(r)\}\omega(\alpha_i) \stackrel{\text{IH}}{=} \{a_{\#ct_i}/\omega(\iota(r))\}\omega(\alpha_i) \stackrel{(*)}{=} \omega(\iota(\#ct_i(r)))$$

where $(*)$ uses Lem. A.16 and the premise.

- For any other operator *op*:

$$\begin{aligned}\kappa(op(r_1, \ldots, r_n)) &= op(\kappa(r_1), \ldots, \kappa(r_n)) \\ &\stackrel{\text{IH}}{=} op(\omega(\iota(r_1)), \ldots, \omega(\iota(r_n))) \stackrel{(*)}{=} \omega(\iota(op(r_1, \ldots, r_n)))\end{aligned}$$

where $(*)$ uses the premise.

A.2. Substitution of Skolem Symbols for JavaCardDL

Analogously to the f-substitutions of the previous section, in this one we define a family of substitutions that replace the skolem symbols Sym_{Sk}, introduced in 4.6.2. These so-called s-substitutions (substitutions of skolem symbols) follow exactly the same intention as f-substitutions, and most properties we have formulated for the latter ones can be translated with minor modifications only. For the following, we denote the set of all formulas, terms and programs of JavaCardDL with *Syn* (as introduced in Sect. 2.1), and with Syn_{Sk} we denote the set of all formulas, terms and programs of JavaCardDL$_{\mathrm{Sk}}$ (i.e. of JavaCardDL, enriched with the skolem symbols Sym_{Sk}).

While for f-substitutions the arguments of symbols to be replaced are terms (and the formal parameters are logical variables, so that normal v-substitutions could be used to achieve call-by-name parameter passing), for skolem symbols of Sym_{Sk} we need to treat terms, program variables and jump statements (these kinds of arguments can occur for skolem symbols). As formal parameters for these kinds of constructs we use

- logical variables $x \in Var$ for terms (when replacing function and predicate skolem symbols), exactly as for f-substitutions

- program variables $v \in PVar$ for program variables (the formal parameters are renamed to the call-parameters)

- for jump statements we use labels $\mathrm{l} \in \mathit{Label}$ (when replacing statement skolem symbols).

We denote the set of these possible formal parameters (depending on a given vocabulary) by *Par*:

$$\mathit{Par} := \mathit{Var} \cup \mathit{PVar} \cup \mathit{Label}.$$

A.19 Definition (Parameter Substitution): A *parameter substitution*

$$\omega = \{x_1/t_1, \ldots, x_k/t_k\}$$

is a map $\mathit{Syn}_{\mathrm{Sk}} \to \mathit{Syn}_{\mathrm{Sk}}$, defined by

$$\omega(x_i) := t_i, \qquad \text{or for } x_i \text{ a label:} \quad \omega(x_i\text{: }\mathbf{skip}) := t_i$$

(i.e. in the latter case the neutral statement **skip**[54] labelled with x_i is replaced with t_i). Each pair (x_i, t_i) ($i \in \{1, \ldots, k\}$) has to be of one of the following kinds:

- $x_i \in \mathit{Var}$ is a logical variable and $t_i \in \mathit{Term}$ is a term, both having the same sort
- $x_i, t_i \in \mathit{PVar}$ are both program variables, having the same type
- $x_i \in \mathit{Label}$ is a label and t_i is a Java statement.

ω is continued as a homomorphism to elements $T \in \mathit{Syn}_{\mathrm{Sk}}$ respecting operators binding symbols, which can be

- operators binding logical variables, like quantifiers and object-level substitution operators (for substitution of logical variables)
- declarations of local (stack) program variables (for renaming program variables).

A.20 Example (Parameter Substitution): We substitute a term, a program variable and a statement:

$$\{x/f(a), \mathrm{v}/\mathrm{w}, l/\mathbf{throw}\ \mathrm{t}\}\Big(\langle \mathbf{int}\ \mathrm{v};\ \mathrm{v} = 0;\ \mathrm{l}\!:\!\mathbf{skip};\rangle p(x, \mathrm{v})\Big)$$
$$= \langle \mathbf{int}\ \mathrm{v};\ \mathrm{v} = 0;\ \mathbf{throw}\ \mathrm{t};\rangle p(f(a), \mathrm{w})$$

The following definition is almost identical to Def. A.4, except that other kinds of symbols are replaced, and that we use the parameter substitution of Def. A.19 instead of v-substitutions:

A.21 Definition (s-Substitution): An *s-substitution* σ is a map

$$\sigma : \mathit{Syn}_{\mathrm{Sk}} \to \mathit{Syn}_{\mathrm{Sk}}$$

such that there is a map

$$\sigma_{\mathrm{cont}} : \mathit{Sym}_{\mathrm{Sk}} \to \mathit{Syn}_{\mathrm{Sk}} \times \mathit{Par}^*$$

satisfying

[54] We only allow the replacement of the neutral statement **skip** to avoid problems and more complicated definitions caused by free symbols (like program variables or labels) that could otherwise occur within the statement.

- each function symbol $s_{\mathrm{Sk}} \in \mathit{Sym}_{\mathrm{Sk}}$ is mapped onto a pair $(T, \langle x_1, \ldots, x_k \rangle)$, where T is
  - a term that has the same sort as s_{Sk} for $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}}$
  - a formula for $s_{\mathrm{Sk}} \in \mathit{Pred}_{\mathrm{Sk}}$
  - a JavaCard statement for $s_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$
  - a JavaCard expression that has the same type as s_{Sk} for $s_{\mathrm{Sk}} \in \mathit{Expression}_{\mathrm{Sk}}$

  and $x_1, \ldots, x_k \in \mathit{Par}$ are distinct formal parameters, compatible to the signature of s_{Sk} (and in particular having correct sorts and types):
  - logical variables $x_1, \ldots, x_l \in \mathit{Var}$, for $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}}$
  - program variables $x_{l+1}, \ldots, x_m \in \mathit{PVar}$ (in any case)
  - labels $x_{m+1}, \ldots, x_k \in \mathit{Label}$, for $s_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$
- for almost all symbols s_{Sk}, $\sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (s_{\mathrm{Sk}}(x'_1, \ldots, x'_k), \langle x_1, \ldots, x_k \rangle)$, with

  $$x'_i := \begin{cases} x_i\text{: }\mathbf{skip} & \text{for } x_i \text{ a label} \\ x_i & \text{otherwise} \end{cases}$$

  (i.e. essentially s_{Sk} is mapped onto itself)
- σ is the application of σ_{cont} to terms, formulas and programs:
  - For skolem symbols s_{Sk}:

    $$\sigma(s_{\mathrm{Sk}}(r_1, \ldots, r_k)) = \{x_1/\sigma(r_1), \ldots, x_k/\sigma(r_k)\}T \quad \text{where} \quad \sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (T, \langle x_1, \ldots, x_k \rangle)$$

    (we use the parameter substitution of Def. A.19)
  - All other constructs are handled faithfully.

A.22 Example (s-Substitution): For the statement skolem symbol s_{Sk} and the s-substitution σ described by

$$\sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (\alpha, \langle \mathrm{v}, \mathrm{l} \rangle),$$

and the statement α being

```
{   if ( v == 0 ) l:skip; else break m;   }
```

we consider the application

$$\sigma\Big(\langle \mathrm{l}\ :\{\ \mathbf{int}\ \mathrm{w};\ s_{\mathrm{Sk}}(\mathrm{w}; \mathbf{continue}\ \mathrm{m});\ \}\rangle\varphi\Big).$$

To perform this application, first the parameters of s_{Sk} have to be substituted in α:

$$\begin{aligned} \sigma\big(s_{\mathrm{Sk}}(\mathrm{w}; \mathbf{continue}\ \mathrm{m})\big) &= \{\mathrm{v}/\mathrm{w}, \mathrm{l}/\mathbf{continue}\ \mathrm{m}\}\alpha \\ &= \{\ \mathbf{if}\ (\ \mathrm{w} == 0\ )\ \mathbf{continue}\ \mathrm{m};\ \mathbf{else\ break}\ \mathrm{m};\ \}. \end{aligned}$$

The result of the parameter substitution is then inserted in the enclosing formula, which leads to

$$\langle \mathrm{l}\ :\{\ \mathbf{int}\ \mathrm{w};\ \{\ \mathbf{if}\ (\ \mathrm{w} == 0\ )\ \mathbf{continue}\ \mathrm{m};\ \mathbf{else\ break}\ \mathrm{m};\ \}\ \}\rangle\sigma(\varphi).$$

In the previous section we denote the set of free symbols within a term or formula T by $FS(T)$. To treat issues of dynamic logic, in this section we also allow T to be a program (fragment), and define $FS(T)$ to contain the following symbols of T:

- free logical variables
- function and predicate symbols
- free (i.e. not locally declared) program variables
- skolem symbols of Sym_{Sk}
- free labels within programs, i.e. labels occurring as arguments of a **break** or **continue**-statement, that are not bound by the labelling of an enclosing statement.

Referring to this generalised meaning of FS, the domain $\mathrm{Dom}(\sigma)$ and codomain $\mathrm{Cod}(\sigma)$ of an s-substitution σ are defined similar as for f-substitutions (Def. A.7), namely by

A.23 Definition (Domain and Codomain): Let σ, σ_{cont} be as in Def. A.21. As usually we define the *domain* $\mathrm{Dom}(\sigma)$ of σ to be the set of skolem symbols not mapped onto themselves (as in Def. A.21), and the *codomain* $\mathrm{Cod}(\sigma)$ of σ by

$$\mathrm{Cod}(\sigma) = \bigcup_{\substack{s\in\mathrm{Dom}(\sigma)\\ (a,\langle x_1,\ldots,x_k\rangle)=\sigma_{\mathrm{cont}}(s)}} FS(a)\backslash\Big(\{x_1,\ldots,x_k\}\backslash \mathit{Label}\Big)^{55}$$

We reformulate Lem. A.8, which determines the free variables that can be introduced by an f-substitution, in a slightly more general version that is needed for a subsequent proof:

A.24 Lemma (Free Symbols and s-Substitutions): Let σ be an s-substitution and $T \in Syn_{\mathrm{Sk}}$ an element of $\mathrm{JavaCardDL}_{\mathrm{Sk}}$. Then we have

$$FS(\sigma(T)) \subset FS(T) \cup \mathrm{Cod}(\sigma).$$

Renaming of bound symbols can be performed for s-substitutions as for f-substitutions:

A.25 Definition (Bound Renaming of s-Substitutions): Let σ_{cont} and ω_{cont} be contents as in Def. A.21. σ_{cont} and ω_{cont} and called *equal modulo bound renaming*, if for each symbol $s_{\mathrm{Sk}} \in Sym_{\mathrm{Sk}}$ with $\sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (a, \langle x_1,\ldots,x_k\rangle)$, $\omega_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (b, \langle y_1,\ldots,y_k\rangle)$ the elements

$$\{x_1/z_1,\ldots,x_k/z_k\}a \quad \text{and} \quad \{y_1/z_1,\ldots,y_k/z_k\}b$$

are equal modulo bound renaming, where $z_1,\ldots,z_k$ are distinct. The substitutions used are parameter substitutions, and the particular replacements z_i are chosen accordingly to the kind of x_i (or y_i):

[55] We have to remove labels x_i from the list of formal parameters, as there may be free occurrences of x_i within a which are ignored by a parameter substitution (parameter substitutions only replace distinguished statements labelled with x_i).

- For a logical variable $x_i \in \mathit{Var}$, z_i also is a logical variable, which has the same sort as x_i, and that does not occur in a or b
- For a program variable $x_i \in \mathit{PVar}$, z_i is a program variable, which has the same type as x_i, and that does not occur in a or b
- For a label $x_i \in \mathit{Label}$, z_i is the empty statement labelled with a new label z'_i: z'_i: **skip**.

s-Substitutions σ and ω are called *equal modulo bound renaming*, if there are contents σ_{cont} and ω_{cont} as in Def. A.21 which are equal modulo bound renaming. $\ast$

A.26 Definition (Restriction of s-Substitutions): Let σ and σ_{cont} be as in Def. A.21, and suppose $\mathbf{S} \subset \mathrm{Dom}(\sigma)$ is a set. By

$$\sigma|_{\mathbf{S}} := \omega$$

we denote the *restriction* of σ to $\mathbf{S}$, which is defined by

$$\omega_{\text{cont}}(s_{\text{Sk}}) = \sigma_{\text{cont}}(s_{\text{Sk}}) \qquad \text{for } s_{\text{Sk}} \in \mathbf{S},$$

where ω, ω_{cont} are as in Def. A.21, and ω_{cont} maps all other symbols onto themselves (again as in Def. A.21). $\ast$

A.27 Lemma (Decomposition of s-Substitutions): Let σ and σ_{cont} be as in Def. A.21, and suppose $\mathbf{S} \subset \mathrm{Dom}(\sigma)$ and $\mathbf{T} := \mathrm{Dom}(\sigma) \backslash \mathbf{S}$ are sets. If

$$\mathbf{T} \cap \mathrm{Cod}(\sigma|_{\mathbf{S}}) = \emptyset$$

then it is possible to decompose σ:[56]

$$\sigma = \sigma|_{\mathbf{T}} \circ \sigma|_{\mathbf{S}}$$

A.2.1. Collisions

Collisions that may occur when applying an s-substitution σ are defined similar as for f-substitutions (Def. A.10), but we refer to more kinds of symbols:

- Logical variables
- Program variables
- Labels.

To be more exact, let σ and σ_{cont} be as in Def. A.21 and $\sigma_{\text{cont}}(s_{\text{Sk}}) = (T, \langle x_1, \ldots, x_k \rangle)$. The parameter substitution

$$\{x_1/\sigma(r_1), \ldots, x_k/\sigma(r_k)\}T$$

that is performed upon the replacement of an occurrence of s_{Sk} when applying σ can

[56]The performed concatenation $\circ$ is the concatenation of maps.

- Insert terms $\sigma(r_i)$ containing free logical variables within the scope of operators binding logical variables. For program variables or labels within $\sigma(r_i)$ this cannot happen, as we assume that scopes of these symbols are restricted to program blocks, which on the other hand cannot contain logical variables (and terms $\sigma(r_i)$ are only substituted for logical variables x_i).

- Rename occurrences of program variables $\sigma(r_i)$ within the scope of declarations of other program variables (within program blocks).

- Insert statements $\sigma(r_i)$
  - that contain free program variables within the scope of declarations of program variables
  - that contain jump statements with labels within labelled statements.[57]

Substituting the result r of the parameter substitution for the skolem symbol s_{Sk} may cause further collisions:

- For $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}}$, T is a term or formula and may contain free logical variables (which are not formal parameters), and can be inserted within the scope of an operator that binds variables (again, problems with program variables or labels cannot occur at this point).

- For $s_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}} \cup \mathit{Expression}_{\mathrm{Sk}}$, T is a statement or expression, which may contain free program variables (again not formal parameters) or labels, and can be inserted below/after declarations of program variables or within labelled statements (T cannot contain logical variables, and thus no collisions involving logical variables can occur)

- A symbol $s_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ may be replaced with a statement not respecting scopes (following Def. 2.1), which can alter the binding of trailing program variable occurrences.

To prevent the last three kinds of collisions, we will usually formulate conditions to the codomain $\mathrm{Cod}(\sigma)$ of σ (note that in general it is not possible to avoid these collisions by bound renaming of the s-substitution, it is additionally necessary to rename symbols of the formula the s-substitution is applied to):

A.28 Definition (Collision Preventing s-Substitutions): An s-substitution σ is called *collision preventing* regarding an element $T \in \mathit{Syn}_{\mathrm{Sk}}$, iff

- The codomain $\mathrm{Cod}(\sigma)$ does not contain logical variables[58] or labels

$$\mathrm{Cod}(\sigma) \cap \mathit{Var} = \emptyset, \qquad \mathrm{Cod}(\sigma) \cap \mathit{Label} = \emptyset$$

[57] More generally, one could always call the modification of the target of a jump statement by an insertion a collision; this would also apply to unlabelled **break**- and **continue**-statements or **throw**-statements. However, as we are mainly interested in the notion "collision" as a premise for Lem. A.30, and are not formulating a semantical substitution lemma, this is not necessary.

[58] This is also the premise of Lem. A.8, and is observed in Sect. 4.4.2 to hold for the f-substitution used for FOL proof lifting.

- Program variables of the codomain $\mathrm{Cod}(\sigma)$ do not occur in T.

In this definition, the second condition regarding program variables is weaker than the corresponding condition on occurrences of logical variables, because free occurrences of program variables are allowed for the JavaCardDL sequent calculus. Thus it is not possible to forbid free program variables that are introduced by s-substitutions completely. This observation is also made in the definition of the pvPrefix-property of schema variables (Sect. 3.2.2).

As for f-substitutions, collisions caused by the parameter substitution and statements not respecting scopes can be avoided by bound renaming:

A.29 Lemma (Collision Free s-Substitutions): Let σ be an s-substitution and $T \in Syn_{\mathrm{Sk}}$ an element of $\mathrm{JavaCardDL}_{\mathrm{Sk}}$, such that σ is collision preventing regarding T. Then there exists an s-substitution ω, that is equal to σ modulo bound renaming, such that the application of ω to T is collision free. $*$

In matters of this lemma, the requirement that σ is collision preventing could actually be weakened. In particular could the second item of Def. A.28 be replaced with the condition that program variables of the codomain must not occur *bound* in T.

A.30 Lemma (Renaming Everything): Let $T, S \in Syn_{\mathrm{Sk}}$ be elements of $\mathrm{JavaCardDL}_{\mathrm{Sk}}$ which are equal modulo bound renaming, and σ and ω be s-substitutions which are also equal modulo bound renaming. If the applications of σ to T and of ω to S are collision free, then $\sigma(T)$ and $\omega(S)$ are equal modulo bound renaming. $*$

A.2.2. Concatenation of s-Substitutions

The following two lemmas are adaptions of lemmas from Sect. A.1 to s-substitutions. For that, we assume that f-substitutions are continued to programs by the identity map.

A.31 Lemma (Analogue to Lem. A.15): Let σ be a v-substitution or a nullary f-substitution, $T \in Syn_{\mathrm{Sk}}$ an element of $\mathrm{JavaCardDL}_{\mathrm{Sk}}$, and $\omega = \{y_1/t_1, \dots, y_l/t_l\}$ a parameter substitution. Provided that

- the application of ω to T does not cause collisions and
- $FS(T) \cap \mathrm{Dom}(\sigma) \subset \{y_1, \dots, y_l\}$

we have

$$\sigma(\omega(T)) = \nu(T) \quad \text{where} \quad \nu = \{y_1/\sigma(t_1), \dots, y_l/\sigma(t_l)\}.$$

A.32 Lemma (Analogue to Lem. A.16): Let σ be an s-substitution, $T \in Syn_{\mathrm{Sk}}$ an element of $\mathrm{JavaCardDL}_{\mathrm{Sk}}$, $y_1, \dots, y_l$ either distinct variables or distinct nullary function or predicate symbols with

$$(\mathrm{Dom}(\sigma) \cup \mathrm{Cod}(\sigma)) \cap \{y_1, \dots, y_l\} = \emptyset$$

and $\omega = \{y_1/t_1, \dots, y_l/t_l\}$ a v-substitution or a nullary f-substitution. If the application of σ to T is collision free, then the following holds:

$$\sigma(\omega(T)) = \nu(\sigma(T)) \quad \text{where} \quad \nu = \{y_1/\sigma(t_1), \dots, y_l/\sigma(t_l)\}.$$

A.2.3. s-Substitutions and Schema Variables

The following lemma is essential for the proof transformation performed in Sect. 4.6.6 treating JavaCardDL, and essentially contains the same proposition as Lem. A.18: The instantiation of a schematic formula and the application of an f-substitution to the resulting instance can be commuted. The lemma in this section is however not restricted to schema variables for FOL, but discusses all variables defined in Sect. 3.2, and it refers to s-substitutions instead of f-substitutions. As for Lem. A.18, to preserve the validity of an instantiation α of a StatementSV when an s-substitutions σ is applied, it is necessary formulate a number of conditions on σ. These restrictions are embodied in the next few definitions.

For the following, we assume that a set $\mathbf{S}$ of schema variables is fixed, and denote the set of schematic elements of JavaCardDL (resp. of JavaCardDL$_{\mathrm{Sk}}$) with $\mathit{Syn}_{\mathrm{SV}}$ (resp. with $\mathit{Syn}_{\mathrm{Sk,SV}}$).

First we continue the notion of statements respecting scopes, which is introduced in Def. 2.1, to s-substitutions:

A.33 *Definition (s-Substitutions respecting Scopes):* Let σ and σ_{cont} be as in Def. A.21. We say that σ *respects scopes*, if for each statement skolem symbol $s_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ with $\sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (T, \langle \ldots \rangle)$ the statement T respects scopes. $*$

The next definition is the central premise of the commutation lemma, and it can be regarded as an adaption of the first premise of Lem. A.18 for f-substitutions:

- The codomain $\mathrm{Cod}(\sigma)$ of the f-substitution that is considered must not contain logical variables.

The corresponding requirements for s-substitutions are a bit more complicated, however:

A.34 *Definition (Compatible s-Substitutions):* Let σ be an s-substitution, and $\mathbf{F} \subset \mathit{Syn}_{\mathrm{Sk}}$ be a set of elements of JavaCardDL$_{\mathrm{Sk}}$. We say that σ is *compatible* with $\mathbf{F}$, iff:

- For each $T \in \mathbf{F}$: σ is collision preventing regarding T (by Def. A.28)
- For $s_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ and $\sigma_{\mathrm{cont}}(s_{\mathrm{Sk}}) = (a, \langle x_1, \ldots, x_k \rangle)$ (σ, σ_{cont} as in Def. A.21):
  - a does not contain any method-frames
  - there are no **return**/**break**/**continue**-statements (with or without argument) within a whose target is outside of a
- σ respects scopes.

The second item could be formulated as: the replacement of s_{Sk} may complete abruptly only through statements which are arguments of s_{Sk} (and by exceptions).

Instantiations of StatementSV have to respect scopes, hence it is necessary that this property is preserved by s-substitutions. Otherwise the schematic formulas we are considering could not be closed under s-substitutions:

A.35 *Lemma (Statements and s-Substitutions respecting Scopes):* Let α be a statement or a list of statements, and σ an s-substitution, both respecting scopes. Then $\sigma(\alpha)$ also respects scopes. $*$

Finally, we can formulate the announced lemma:

A.36 *Lemma (Schema Variables and s-Substitutions):* Let φ be a formula, term or program containing the schema variables $\#s_1, \ldots, \#s_k$ (which may be of any type we have defined), such that for each occurring ContextSV the unique-flag is false. Moreover let $\iota = \{\#s_1/\alpha_1, \ldots, \#s_k/\alpha_k\}$ be a valid instantiation which may contain skolem symbols (following Def. 4.19), and σ be an s-substitution

- that is compatible with $\{\iota(\varphi), \alpha_1, \ldots, \alpha_k\}$ (by Def. A.34)
- for each (skolem) symbol s of φ: $s \notin \mathrm{Dom}(\sigma)$
- for each ContextSV or PContextSV $\#s_i$: $a_{\#s_i} \notin \mathrm{Dom}(\sigma) \cup \mathrm{Cod}(\sigma)$.

Then there is an s-substitution ω, equal to σ modulo bound renaming, with

1. $\kappa = \{\#s_1/\omega(\alpha_1), \ldots, \#s_k/\omega(\alpha_k)\}$ is a valid instantiation, in which the applications of ω are collision free
2. The following equation, in which the application of ω is collision free, holds:

$$\kappa(\varphi) = \omega(\iota(\varphi))$$

Proof: To avoid collisions, we replace all bound logical variables, program variables and labels of σ by new, distinct symbols (not already occurring in σ, φ or ι). The resulting s-substitution ω can be applied without collisions to $\iota(\varphi)$ and $\alpha_1, \ldots, \alpha_k$.

1. To show the validity of the instantiation κ, we consider the definitions of each schema variable type:
   - Instantiations of VariableSV are not modified and do not occur bound within any instantiation $\omega(\alpha_i)$, as they do not occur bound in α_i and cannot be introduced (bound) by ω; the same holds for LabelSV
   - By Lem. A.24, for instantiations of TermSV and FormulaSV α_i we have

     $$FS(\omega(\alpha_i)) \subset FS(\alpha_i) \cup \mathrm{Cod}(\omega),$$

     and as ω prevents collisions even $FV(\omega(\alpha_i)) \subset FV(\alpha_i)$; as in the proof of Lem. A.18 for ContextSV $\#s_j = \#ct_j$ we have $\mathrm{Bound}(\alpha_j) \subset \mathrm{Bound}(\omega(\alpha_j))$ and the condition given by the prefix property of TermSV and FormulaSV holds for κ
   - For a ContextSV $\#s_i = \#ct_i$ having the SUL-flag set to true, in α_i skolem symbols cannot occur above the symbol $a_{\#ct_i}$ (by Def. 4.19); therefore the application of ω cannot introduce modalities above $a_{\#ct_i}$ either

- Instantiations of PVariableSV $\#s_i$ are not modified by ω, and cannot occur bound within the instantiation of any other schema variable (which follows from the same arguments as for VariableSV). We can further observe that for the instantiation $\alpha_i = \mathrm{v}_i$ and another schema variable $\#s_j$

$$\mathrm{v}_i \notin FS(\alpha_j) \implies \mathrm{v}_i \notin FS(\omega(\alpha_j)),$$

again because of

$$FS(\omega(\alpha_j)) \subset FS(\alpha_j) \cup \mathrm{Cod}(\omega),$$

and $\mathrm{v}_i \notin \mathrm{Cod}(\omega)$ by premise. Therefore the conditions given by the properties unusedOnly and pvPrefix (for TermSV, FormulaSV, StatementSV and ExpressionSV) are fulfilled

- By Def. A.34, instantiations $\omega(\alpha_i)$ of StatementSV $\#s_i$ do not contain method-frames (provided that α_i does not contain any); by Lem. A.35 $\omega(\alpha_i)$ respects scopes
- Considering the jumpPrefix property of StatementSV $\#s_i$, we are again using that PVariableSV and LabelSV instantiations are not altered by ω, and for a PContextSV $\#s_j = \#pct_j$ we have

$$\mathrm{Jumps}(\omega(\alpha_j)) = \mathrm{Jumps}(\alpha_j)$$

as the blocks within α_j relevant for $\mathrm{Jumps}(\alpha_j)$ are not modified by an s-substitution (and therefore the symbol $a_{\#pct}$ occurs in $\omega(\alpha_j)$ exactly once at a valid position). The set $\mathbf{T}$ of the StatementSV definition thus remains untouched for κ:

$$\mathbf{T}_{\#s_i}(\iota) = \mathbf{T}_{\#s_i}(\kappa).$$

As ω is a compatible s-substitution, $\omega(\alpha_i)$ will on the other hand not contain more (significant) jump statements than α_i, regarding Def. 3.14.

2. The equation is proved by a structural induction:

- For a ContextSV $\#s_i$, we use the same reasoning as in the proof of Lem. A.18, replacing the reference to Lem. A.16 by Lem. A.32
- For a PContextSV $\#s_i = \#pct_i$, we can observe again that $a_{\#pct}$ occurs in α_i only below operators that ω treats as a homomorphism, which immediately entails

$$\kappa(\#pct_i(\beta)) = \omega(\iota(\#pct_i(\beta)))$$

- For other schema variable types (for which the instantiation map is a simple replacement, as we have already discussed the more complicated ones) and other operators, the claim follows as in the proof of Lem. A.18.

References

[ABB+02] 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, *Fundamental Approaches to Software Engineering. 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

[ABB+03] 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. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, February 2003.

[Bec01] 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.

[BGH+03] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias (Serie A, Matemáticas)*, to appear, 2003.

[BS01] Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.

[Fit96] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

[Gie02] Martin Giese. STSR mit Schemavariablen und lokale Deklaration. Arbeitspapier, Fakultät für Informatik, Universität Karlsruhe, October 2002.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.

[Hab] Elmar Habermalz. STSRs für Metavariablen und Java-DL. Arbeitspapier, Fakultät für Informatik, Universität Karlsruhe.

[Hab00] Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln*. PhD thesis, Universität Karlsruhe, 2000.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[Kel02] Uwe Keller. Übersetzung von OCL-Constraints in Formeln einer Dynamischen Logik für Java. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2002. In German.

[OMG01] Object Modeling Group. *Unified Modelling Language Specification, version 1.4*, September 2001.

[SA94] Rolf Socher-Ambrosius. *Deduktionssysteme*. Bibliographisches Institut & F.A. Brockhaus, 1994.

[Sch02] Steffen Schlager. Behandlung von Integer Arithmetik bei der Verifikation von Java-Programmen. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, May 2002.

[Sun02] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Platform Specification*, September 2002.