Thesis for the Degree of Licentiate of Philosophy

Proving and Disproving in Dynamic Logic for Java

Philipp Rümmer

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, 2006

Proving and Disproving in Dynamic Logic for Java
Philipp Rümmer

Technical Report no. 26L
ISSN 1652-876X
Department of Computer Science and Engineering
Research Group: Formal Methods Group

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31–772 1000

Printed at Chalmers, Göteborg, 2006

Abstract. This thesis is about proving the functional correctness and incorrectness of imperative, object-oriented programs. One of the main approaches for the first item is deductive program verification, whereas the second item is traditionally handled by techniques like testing. In this thesis, we show how both correctness and incorrectness can be covered by dynamic logic for Java (a program logic) and be handled using similar techniques. The theorem prover KeY, which provides an implementation of dynamic logic for Java, was used for experiments and was extended for this purpose.

We introduce the concept of taclets, which is the rule language that is used to implement the calculus for Java dynamic logic in KeY. Apart from a detailed introduction of the language and complete definitions of the semantics of taclets, reasoning about the correctness of taclets is discussed. This part of the thesis is the most complete account on taclets so far.

The concept of updates is described, which is the central component for performing symbolic execution in Java dynamic logic. Updates are systematically developed as an imperative programming language that provides the following constructs: assignments, guards, sequential composition and bounded as well as unbounded parallel composition. The language is equipped both with a denotational semantics and a correct rewriting system for execution, whereby the latter is a generalisation of the syntactic application of substitutions. The normalisation of updates is discussed.

Acknowledgements

I am extremely grateful to my supervisor, Wolfgang Ahrendt, for many discussions about my work and the secrets of computing science and logic, fast feedback whenever it was required, and for always having time regardless of how many other commitments were waiting.

I want to thank the PhD students and all members of the department for creating such a nice and relaxed atmosphere, and for making it such a hard task to concentrate on work. Special thanks go to Tobias Gedell, Rogardt Heldal, Angela Wallenburg, Dennis Walter, and Marcin Zalewski.

Many thanks are also due to the people that are or were working in the KeY project in Gothenburg, Karlsruhe, and Koblenz during the last two years, and that were all to some degree affected by the contents of this thesis. The persons that were most affected and deserve to be mentioned are Richard Bubel and Muhammad Ali Shah, thank you guys!

Table of Contents

1 Program Analysis 1
1.1 Deductive Verification 3
1.2 Testing 9
2 First-Order Predicate Logic 10
2.1 Sequent and Tableaux Calculi 11
3 Dynamic Logic for Java 13
4 Implementation of Proof Assistants 14
5 Overview 16

Taclets — A Language for Sequent Calculi 23
Philipp Rümmer
1 Taclets by Example 27
2 Schema Variables 36
2.1 The Kinds of Schema Variables in Detail 38
2.2 Schematic Expressions 42
2.3 Instantiation of Schema Variables and Expressions 44
2.4 Substitutions Revisited 46
2.5 Schema Variable Modifiers 48
2.6 Schema Variable Conditions 49
2.7 Generic Types 49
3 Instantiations and Metavariables—A Taster 54
4 Systematic Introduction of Taclets 56
4.1 The Taclet Language 56
Context Assumptions: What has to be present in a sequent 56
Find Pattern: To which expressions a taclet can be applied 58
State Conditions: Where a taclet can be applied 58
Variable Conditions: How schema variables may be instantiated . 60
Goal Templates: The effect of the taclet application 60
Rule Sets: How taclets are applied automatically 61
4.2 Well-Formedness Conditions on Taclets 61
4.3 Implicit Bound Renaming and Avoidance of Collisions 63
4.4 Applicability of Taclets 66
4.5 The Effect of a Taclet 69
4.6 Taclets in Context: Taclet-Based Proofs 70
5 Reasoning about the Soundness of Taclets 72
5.1 Soundness in Sequent Calculi 73
5.2 A Basic Version of Meaning Formulae 74
5.3 Meaning Formulae for Rewriting Taclets 76
5.4 Meaning Formulae in the Presence of State Conditions 77
5.5 Meaning Formulae for Nested Taclets 79
5.6 Elimination of Schema Variables 81

Sequential, Parallel, and Quantified Updates of First-Order Structures ... 89
Philipp Rümmer
1 Introduction ... 89
2 Updates for Symbolic Execution in Dynamic Logic ... 90
3 Syntax of Terms, Formulas and Updates ... 91
4 Semantics of Terms, Formulas and Updates ... 92
5 Application of Updates by Rewriting ... 96
6 Application of Substitutions by Rewriting ... 99
7 Sequentiality and Application of Updates to Updates ... 99
8 Soundness and Completeness of Update Application ... 101
9 Modelling Stack and Heap Structures ... 102
Variables ... 102
Local Variables ... 102
Explicit Stack ... 103
Classes and Attributes ... 103
Object Allocation ... 104
Arrays ... 104
10 Symbolic Execution in Dynamic Logic Revisited ... 104
11 Laws for Update Simplification ... 105
12 Normalisation and Equivalence Modulo Definedness ... 107
Assignments vs. Modifications ... 108
Normalisation of Updates ... 109
13 Related Work ... 109
14 Conclusions and Future Work ... 110

Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic ... 115
Philipp Rümmer, Muhammad Ali Shah
1 Introduction ... 115
2 Formalisation of the Problem in Dynamic Logic ... 116
2.1 Heap Representation in Dynamic Logic for Java ... 118
2.2 Formalising the Violation of Post-Conditions ... 118
2.3 Quantification over Program States ... 119
3 Constructing Proofs for Program Incorrectness ... 120
3.1 Using a Ground Proof Procedure ... 122
3.2 Construction of Proofs using Metavariables and Backtracking ... 123
3.3 Construction of Proofs using Incremental Closure ... 124
3.4 A Hybrid Approach: Backtracking and Incremental Closure ... 126
4 Representation of Solutions: Constraint Languages ... 126
5 Related Work ... 127
6 Conclusions and Future Work ... 128

Introduction

This thesis is about proving the functional correctness and incorrectness of programs. One of the main approaches for the first item is *deductive program verification*, whereas the second item is traditionally handled by techniques like *testing*. In this thesis, we show how both correctness and incorrectness can be covered by dynamic logic for Java (a program logic) and be handled using similar techniques. The theorem prover KeY, which provides an implementation of dynamic logic for Java, was used for experiments and was extended for this purpose.

Sect. 1 gives a short introduction to deductive verification and to testing, and in particular the background to the third paper of this thesis (page 115). Sect. 2 and 3 introduce first-order predicate logic and dynamic logic. Sect. 1.1 and 3 motivate the second paper of the thesis (page 89). Finally, Sect. 4 is a short account on proof assistants and gives a background for the first paper of the thesis (page 23).

1 Program Analysis

The topic of this thesis is to reason about the behaviour of programs. We focus on a very specific area: the programs that we consider are *imperative* programs, which means that the semantics of a program is centred around the notion of *states*, and that the execution of a program consists of a series of state changes. While this kind of programs is often considered as too low-level, i.e., too similar to the technical details of a computer, it is the model of computation that is most frequently used to write programs.

As a second choice, the programs that we investigate will be *object-oriented*, which on the one hand means that programs can store data as a graph, the *heap*, and on the other hand that the language conceptionally attaches behaviour to pieces of data. For this thesis, the only important aspect of object-oriented languages is the handling of heap and of linked datastructures. The same effects as with heaps can already be observed when working with arrays: the number of involved locations is in general unbounded, and it is not decidable whether two program expressions denote the same or different locations (*aliasing*).

Although most parts of the thesis are independent of a particular programming language and are meaningful for all (object-oriented, imperative) languages, the language that is used throughout the thesis is Java [1]. We do not consider issues like concurrency, so that the treated fragment of Java mostly corresponds to the JavaCard language [2].

Language Semantics The behaviour of an imperative program can be investigated on different levels of detail. A denotational view will reduce a program to its *input-output-relation* (I/O-relation), i.e., to the binary relation between

pre-states and the post-states that can be reached by running the program. Because we only investigate *deterministic* programs, the I/O-relations are partial functions, i.e., map a pre-state to at most one post-state. In this thesis, the behaviour of programs is always specified by stating properties of the I/O-relation. The most common approach for such specifications are *pre- and post-conditions*, which is a concept that, for instance, is essential for *Design by Contract* [3].

A second view on the semantics of programs is *operational semantics*. Describing the operational meaning of an imperative programming language essentially means writing an interpreter for the language. Because this is a comparatively simple task even for complicated languages, it is—in different flavours, like for actual or for symbolic execution—often used as basis of program analysis. The execution of an imperative program consists of a sequence of state transitions. When looking at these transitions one at a time, we see the *small-step operational semantics* of the program. If all steps, from the beginning of the execution until the (possible) termination of the program, are combined, we are talking about the *big-step operational semantics*, which essentially coincides with the I/O-relation of a program.

Denotational and operational models are equally important in this thesis: while we specify programs by stating desired properties of their denotation, the actual analysis of the programs is performed using an operational definition of the language semantics. In this context, the second paper about updates (page 89) discusses the topic of capturing the operational semantics of an imperative language as rules of dynamic logic.

Specifications and Assertion Languages We need a language for describing properties of I/O-relations. In practice, often natural language is used, but in order to mechanically reason about a program it is necessary to provide a *formal* specification. The languages that this thesis concentrates on are based on *classical first-order logic* (see Sect. 2), extended with algebraic theories like natural numbers and lists. When used for specification, this language often appears in disguise and with an unusual syntax: specification languages that essentially coincide with first-order logic are, for instance, the Java Modelling Language (JML) [4] or the Object Constraint Language (OCL) [5]. For reasoning about programs and specifications, this is mostly irrelevant. How first-order logic is used in specifications is illustrated in Sect. 3.

It should be noted, that already the effort of creating a formal specification is usually significant, even though specification languages are designed to be easy to learn and to use. The lack of a tailor-made specification for a program does not necessarily mean, however, that the techniques discussed here are not applicable. It can be interesting to reason about standard properties that are often not stated explicitly, like about termination or exception-freeness. Such properties are one of the main application areas for deduction-based verification systems and software model checkers.

Compile-Time Analysis Typically, one distinguishes between dynamic and static analyses. The first approach works with information that is obtained by actu-

ally running a program, whereas the second one gains information by directly analysing program code (or just any suitable representation of a program). This distinction is misleading, because (i) it is not clear whether *symbolic* execution of a program should be regarded as dynamic or static, and (ii) the term "static analysis" is often used only for a very special kind of analysis, e.g. not for program verification.

In the present thesis, we rather distinguish between runtime and compile-time methods, i.e., between methods that need to be carried out each time a program is run and methods that work upfront. In this sense, all methods shown here are compile-time methods.

Alternatives and Related Approaches The semantics based on an I/O-relation is also called *partial correctness model* and has the property that it is not possible to distinguish between non-termination and erroneous termination of programs. A detailed discussion and further models are given in [6].

A third kind of programming language semantics is called *axiomatic semantics* and is often regarded almost as a synonym for Hoare-style program verification. Such semantics determine the meaning of programs indirectly through rules or axioms of a logic that allow to derive properties of the program.

There are, of course, numerous kinds of formal specifications that are not directly comparable with first-order logic: (i) A very popular method is to use the programming language itself also for specification, which has the advantage that specifications are executable. JML resembles this approach, but offers further concepts like quantifiers that are not present in Java. Because of the limited expressiveness of programming languages, a translation to first-order logic is always possible but requires the same techniques as the actual verification of programs. (ii) Languages like Z [7] or B [8] are based on set theory and are strictly more expressive than first-order logic. (iii) Types are a general means of specifying programs and range from simple datatypes to dependant types that allow functional specifications. A language supporting such specifications is [9]. Whether this kind of specifications can be reduced to first-order logic depends on the particular type system. (iv) Given an embedding of a programming language in an arbitrary logic, this logic can be used for specification (and verification) purposes. This is often done in higher-order proof systems like Isabelle/HOL [10], Coq [11], or PVS [12].

1.1 Deductive Verification

In the following, we assume that we are given a program, together with a formal specification that describes properties of the I/O-relation of the program. If the correctness of the program wrt. the specification is of great importance, then it can be necessary to *verify* the program, i.e., to find a mathematical/logical argument that entails that the program cannot violate the specification. Verification is an intricate problem: (i) it is well-known that, in general, the correctness of a program is not decidable, and furthermore (ii) for most kinds of specifications,

verification of real-world programs is currently beyond the capabilities of automated tools. Likewise, interactive verification is a difficult and time-consuming process.

This thesis concentrates on *deductive verification*, which is verification that uses a proof procedure for a logic as backend. Deductive verification is one of the main approaches to program verification. Using a logic raises the number of involved formal languages to three (although some or all of the languages can coincide): a programming language, a specification language and a logic in which deduction takes place.

When trying to verify a program, we implicitly make a positive assumption: the hypothesis is the correctness of the program, and through verification this claim is supposed to be substantiated. Deductive verification systems are primarily designed for this purpose. This does not mean that the failure to verify a program is not helpful for finding a possible error (in the program or in the specification). Unfortunately, not being able to verify a program does not entail the presence of a bug, which is a consequence of the undecidability of correctness (and the fact that soundness is usually considered as an important property).

Embeddings In order to verify a program deductively, it is necessary to draw a connection between the programming language, the specification language and the logic in which deduction takes place: translations have to be defined that turn both the program and the specification into an expression of the logic. We concentrate on the first case, the creation of an *embedding* of an object-oriented, imperative programming language into a logic.

There are two main approaches for embedding a formal language into a logic, which differ in the way in which the *semantics* of the language is represented:

- Creating a *deep embedding* means to formalise both the syntax and the semantics of the language *within* the target logic. As an example, a deep embedding of a programming language and its operational semantics would essentially be an interpreter that is written in the target logic. Deep embeddings are mostly used to reason about the properties of programming languages ("meta-reasoning" about programs), and are in most cases written in higher-order frameworks that are expressive enough for capturing the semantics of a language in a natural way. For the actual verification of individual programs wrt. a specification, deep embeddings are rather a disadvantage: the effort of creating a deep embedding is big, and using the formalisation of a language semantics itself to determine the meaning of a program is usually not very efficient. Cases in which deep embeddings *are* used for verification are the deep embedding of the Java virtual machine in ACL2 [13], the LOOP tool [14], and the EVT tool for verifying Erlang programs [15] (although the deep embedding is here also used to derive more efficient proof rules).
- A *shallow embedding* is established by defining a translation from the language in question to the target logic *outside* of the target logic. For a programming language, this translation would map programs to a representation of the meaning of the program within the target logic, e.g. to a formula

describing the I/O-relation of the program. This means that the embedding function knows about the semantics of the source language. A shallow embedding is usually easier to realise than a deep embedding, and can be more efficient for the actual verification. The downside is that a shallow embedding cannot directly be used for meta-reasoning.

Again, in this thesis we focus on the case of *shallow embedding*. We find this paradigm in a number of verification systems for imperative programming languages (probably in most of them), although in very different flavours and often somewhat hidden:

Verification Condition Generators Many tools, in particular automated ones, contain a component called the *verification condition generator* (VCG), which is a translator that takes a program and a specification and produces a formula that can consequently be tackled using a theorem prover or an interactive proof assistant. From a technical point of view, this means that the translation of the programming language into a formula and the actual reasoning are strictly separated. The essential correctness property of a VCG is that the produced formula must only be valid if the program is correct wrt. the given specification. We can prove a program correct by showing that the formula produced by a correct VCG is valid. In this architecture, this is mostly done using automated theorem provers, because the formulae that a VCG produces usually have only little structure in common with the original program, and are, therefore, hard to comprehend.

The analysis of a program when computing verification conditions is in most cases very similar to the actual execution of the program, i.e., resembles the operational semantics. A primary distinction that can be drawn is the direction of the analysis, which can be either forwards or backwards. One of the most popular approaches is the classical *weakest-precondition calculus* (wp-calculus) [16], which is a backwards analysis but still very near to the operational semantics.[1] The wp-calculus is known for its surprising simplicity (at least for simple, academic languages), which can intuitively be explained with the facts that (i) when starting with a post-condition and trying to derive the corresponding weakest pre-condition, it is natural to start with the last statement of a program, and (ii) when looking at a post-condition, substituting a term for a variable is equivalent to assigning the value of the term to the variable (the *substitution theorem*), which can be exploited in backwards reasoning. Examples of verification systems for imperative languages (in particular for Java) that use wp-calculus are ESC/Java2 [17], Boogie [18], Jack [19], and Why [20] (which is used as backend for the Krakatoa tool [21]).

Symbolic Execution An approach for creating verification conditions that uses forward-reasoning—but that is otherwise very similar to wp-calculus—is *symbolic execution* (SE) [22]. SE is one of the main topics of this thesis: the second

[1] Initially, the wp-calculus is in fact introduced as *predicate transformer semantics*, i.e., as an independent means of defining the semantics of a programming language.

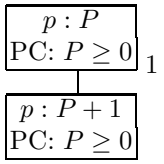
paper about updates (page 89) develops a formalism for carrying out SE in dynamic logic. SE works by executing an imperative program with symbolic initial inputs. The values of variables during the execution are represented as terms over the program inputs (in the original paper [22], as polynomials). The SE of a program is in general not linear, because the values of branch predicates cannot be decided, but can be visualised as a *symbolic execution tree*. Each node in the tree represents a path from the program entry to one of the program statements and shows the values of variables as well as a *path condition* (PC), which is a predicate of the program inputs and determines whether an actual program execution would follow the represented path. Fig. 1 shows, as an example, the SE tree for the following program:

```
1  if (p >= 0)
2    p = p+1;
3  else
4    p = -p;
5
6  if (p == 0)
7    r = 0;
8  else
9    r = r/p;
```

While the wp-calculus works by modifying the post-condition and gradually turns it into a weakest pre-condition, we can imagine that SE operates on the pre-condition (which corresponds to the initial path condition and is simply *true* in Fig. 1) and finally produces a strongest post-condition.[2] This procedure is, however, somewhat hidden in SE, because the modified pre-conditions are stored in two parts: the path conditions and the symbolic variable assignment. The separation avoids the inversion of right-hand sides of assignments during the execution, which is, in principle, necessary to describe strongest post-conditions. In order to illustrate this, we symbolically execute the simple program

```
1  p = p+1;
```

with the pre-condition (initial path condition) $P \geq 0$:



The actual strongest post-condition is obtained by inverting the term $P + 1$:

$$\exists P'.\, P = P' + 1 \wedge P' \geq 0 \quad \equiv \quad P - 1 \geq 0 \quad \equiv \quad P \geq 1$$

[2] Because the values of variables are stored explicitly during SE, it is also possible to use symbolic execution for deriving weakest pre-conditions.

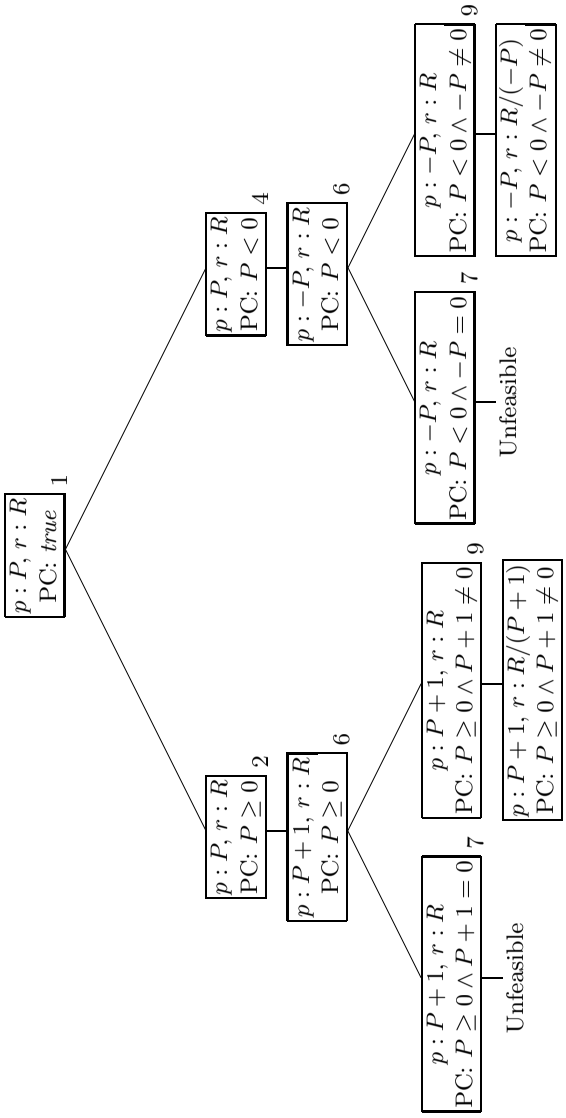


Fig. 1. Symbolic Execution Tree. The numbers next to nodes refer to the line numbers of the program on page 6.

Like the wp-calculus, SE itself can handle loops in programs only through unrolling and needs to be supported by further techniques like induction or invariants in general. This aspect is also discussed in the paper about updates (page 89), where it is proposed to use a more general representation of the symbolic program state in order to handle certain kinds of loops (more details are given in [23]).

For the implementation of verification condition generators, SE is by far less often used than the wp-calculus, although there are no striking reasons to prefer one of the two techniques in this area. In contrast, some of the techniques used in program logics like Hoare logics or dynamic logic can be identified as SE. SE is also popular in the area of software model checking (e.g., [24–27]) or test data generation (Sect. 1.2). One reason for this is the flexibility of only analysing parts of a SE tree, and the possibility to detect unfeasible paths.

Program Logics Instead of separating the generation of verification conditions and the actual reasoning, it is also possible to combine both aspects in one logic. The calculus of such a logic contains both the VCG and a calculus for the underlying logic. The most well-known examples are Hoare-style logics [28], which exist for many imperative languages. Examples of verification systems that are based on Hoare logics for Java are Jive [29] and the system developed as part of Bali [30]. A further program logic is dynamic logic [31], which strongly resembles Hoare logics and is described in more detail in Sect. 3. Strictly speaking, Hoare logics and dynamic logic are examples for a shallow embedding of a programming language, because the semantics of the language is not formalised on the object level of the target logic. The practical difference to an architecture with a separate VCG is that the translation of the program into the logic can be performed lazily, it is not necessary to translate the whole program in one go. This is advantageous for interactive verification, because the structure of a program can be preserved as long as possible.

Program analysis in Hoare logics can be performed both in forward and backward direction, and can to a certain degree be seen as a simulation of either symbolic execution or the wp-calculus. A difference to both techniques[3] is that the usage of intermediate assertions in Hoare proofs (*annotated programs*) allows to reduce proof branching, because the splitting that is necessary to handle conditional statements in a program can be localised.

Heap Representation Both wp-calculus and SE as well as many program logics were initially only formulated for programs without heap or arrays, i.e., for programs whose state is completely determined by the values of the program variables. Program variables can comparatively simply be carried over to a logic and be handled using logical variables or constants, as it is illustrated in the paragraph about symbolic execution above. Handling the heap of a program, which can be seen as a mapping from addresses to values, is more intricate. Two main approaches for representing heap in a first-order logic are:

[3] An optimisation of the wp-calculus that leads to a similar effect is described in [32].

- Because a heap has to property of being unbounded, but finite, it can be modelled through algebraic datatypes like lists, arrays [33, 34], or through more specialised types. This approach is used in ESC/Java2 [17], Boogie [18], Krakatoa [21], and KIV [35].
- The heap can directly be represented as a first-order structure, i.e., by choosing an appropriate vocabulary that represents arrays as functions mapping indexes to values, etc. This approach is chosen in Jack [19] and KeY [36], but also the memory model of separation logic [37] falls into this category. The paper about updates (page 89) in this thesis introduces the main formalism that is used in KeY for modifying the heap.

This distinction resembles the earlier categorisation in deep embeddings and shallow embeddings. The second approach has the disadvantages that arbitrary quantification of program states is not directly possible (in first-order logic), and that additional effort is needed to express well-formedness properties like the existence of only finitely many objects. As an advantage of the second approach, on the other hand, heap accesses can be translated more directly to logical expressions (examples are given in the paper about updates, page 89), which is convenient for interactive verification.

A method for working around the limitation of not being able to quantify over program states is described in the third paper about incorrectness proofs (page 115). Conceptually, the paper shows how the first and the second approach to heap representation can be related using updates.

1.2 Testing

As a second approach to program analysis, we shortly describe methods for generating test data in order to analyse the behaviour of programs. Given a program and/or a specification, such methods produce concrete program inputs on which the program can be run. By observing the output of the program, one then decides whether the behaviour is correct or not. Although testing is also used to examine whether a program is correct, the premisses are different from those of deductive verification. Testing is a search for program inputs for which a program behaves wrongly, which means that it is an attempt to substantiate the hypothesis that the program is *incorrect*. At the same time, testing can (apart from special cases) not prove that programs are correct. In this sense, testing is the opposite of program verification.

The notion of testing as a whole is not directly comparable to deductive verification, it is more general: test data can also be produced by hand or in cases where no formal specification of a program exists. In this regard, we can see testing as a complementary method to verification that can, for instance, also help to validate a specification. In this thesis, however, we concentrate on methods for *automatically* creating test data. Traditionally, two approaches are distinguished:

Specification-Based Testing Following this approach, the generation of test data is driven by an analysis of the specification of a program. In its purest form,

specification-based testing does not analyse the actual program and is therefore also called *black-box testing*. Instead, a specification (or model) of the program, for instance pre- and post-conditions, are used to guess program inputs and to evaluate whether the corresponding program outputs are correct. The program inputs can, for instance, be generated so that all classes of program inputs (up to a suitable notion of isomorphism) that are allowed by the pre-condition are covered (e.g. [38, 39]). Also the generation of random program inputs is common (e.g. [40]).

Implementation-Based Testing The other extreme is to generate test data by analysing the program and ignoring the specification, which is also known as *white-box testing*. Such techniques select test data with the goal of optimising coverage criteria, like that all statements of the program are executed by some test case (statement coverage) or that all branches of conditional statements are taken (branch coverage). This is achieved, besides others, by means of symbolic execution and constraint solving. A survey of coverage criteria and methods is given in [41].

Although implementation-based testing does, in its purest form, not refer to an explicit specification of a program (like pre- and post-conditions), it still has the purpose of ensuring that the program behaves correctly: by testing whether a program terminates properly, raises executions or reaches violated assertions, a specification is reintroduced through the back door.

The third paper of this thesis about incorrectness proofs (page 115) discusses how deductive verification based on dynamic logic can be used to find bugs in programs. Depending on the proof strategy that is used, this method can simulate both specification-based and implementation-based testing, or can combine both methods. The method is also able to find classes of program inputs that reveal bugs instead of only concrete program inputs.

2 First-Order Predicate Logic

The base logic for verifying programs in this thesis is always *classical first-order predicate logic* (FOL). A general introduction to this kind of logic is [42]. FOL goes beyond propositional logic by introducing the notion of *individuals* or *objects*, which are explicitly described using functions and terms and implicitly using predicates and formulae. Quantifiers allow to state properties that are supposed to hold for all or for some individuals. FOL is strictly more expressive than propositional logic: the validity of a first-order formula is only semi-decidable. On the other hand, FOL does not allow quantification over functions or sets of individuals (higher-order quantification), which entails that its expressiveness is strictly less than that of higher-order logics. As a consequence, FOL allows comparatively efficient automated reasoning: the majority of automated proving in deductive verification tools is carried out in FOL.

For deductive program verification, the expressiveness of pure FOL is not sufficient: Turing-complete programming languages inherently contain fixed-point

principles (loops or recursion), which do not exist in FOL. Furthermore, algebraic datatypes (like arithmetic, lists or arrays) that are common in programming languages cannot be specified in FOL. This entails that an adequate and correct embedding of programming languages in pure FOL is not possible. Fortunately, FOL does not lack "much" for handling programming languages: it is enough to add principles like Peano arithmetic to make FOL sufficiently powerful (but still by far less expressive that higher-order logic). For the simple while-language, related completeness results for Hoare-style logics and for dynamic logic are given in [43, 31]. Completeness results for objects-oriented languages are presented in [30, 44].

Proving and Disproving Compared to many other logics, reasoning about formulae in FOL has an asymmetric character: while the validity problem in FOL is semi-decidable (the set of valid formulae is recursively enumerable), the satisfiability problem is more difficult and not even semi-decidable. A different view on this phenomenon is as follows: in a FOL formula, the considered universe as well as predicate and function symbols are implicitly universally quantified. In order to show that a formula is *not* valid (to show that the negation of the formula is satisfiable), this implicit quantification would have to be turned into an explicit existential second-order quantification, which is not possible in FOL.

Consequently, for *proving* (showing the validity of a formula) and *disproving* (showing that a formula is not valid) different approaches are common. Approaches to proving are often based on calculi like resolution or tableaux [45], or (in case of provers that are used for deductive verification) on a combination of propositional calculi, decision procedures for theories and heuristics for handling quantifiers (the most well-known example is Simplify [46]). For disproving, the most successful methods are based on the construction of finite models (e.g. [47]), or on heuristics that predict when the proof search of resolution- or tableaux-based provers has failed and can be stopped.

The situation is somewhat different when FOL is combined with algebraic datatypes, which is exploited in the third paper of the thesis about showing the incorrectness of programs (page 115). Generally, disproving is more difficult than proving due to the presence of loose function or predicate symbols (functions or predicates whose meaning is not uniquely defined by axioms), which represent implicit second-order quantification. When algebraic datatypes are available, however, it is often not necessary to introduce such functions or predicates: algebraic datatypes can be used to represent *finite* mappings or *finite* sets, which suffice in many situations. In Sect. 1.1, for instance, it is discussed that the heap of object-oriented programs can either be represented with datatypes or using loose functions. The datatype solution suits disproving better, because explicit (existential) quantification can be used. In this context, disproving can be as easy or difficult as proving.

2.1 Sequent and Tableaux Calculi

One of the main classes of calculi for classical first-order logics are Gentzen-style sequent or tableaux calculi [48, 49]. The two kinds of calculi are in most regards

equivalent. While the tableaux representation is more popular in the area of automated theorem proving, interactive proof assistants are more often based on sequent calculi, and in this thesis we will only use sequent-notation. Sequent calculi are also more common when working with non-classical logics that are based on FOL, like with dynamic logic (Sect. 3).

In a sequent calculus, the validity of a formula is shown by systematically turning the formula into conjunctive normalform and by showing that each of the conjuncts (which are written as *sequents* $\Gamma \vdash \Delta$) is valid. The third paper of the thesis (page 115) gives examples for sequent calculus rules for FOL.

Metavariables The main issue when searching for proofs in sequent calculi for FOL is the construction of the right instances of quantified formulae, i.e., of the right terms t that have to be substituted for the variable in a formula $\forall x.\ \phi(x)$. A standard technique for finding the required instances are *metavariables* (which are in the tableaux community called free variables), place-holders X that are inserted instead of concrete terms t and that can—at a later point—be replaced (*substituted*) with t. This effectively postpones the problem of choosing the term t and turns it into the problem of deciding when and which substitutions should be applied.

The standard technique for solving this new problem is to use unification for finding substitution candidates, and backtracking (as a part of depth-first search) in order to undo substitutions that appear misleading at a later point. An example is the following proof (attempt), in which the metavariable X is used as a place-holder for the witness that is needed to prove the existentially quantified formula:

$$\dfrac{\dfrac{\dfrac{\vdash\ X = c,\ X = d}{\vdash\ X = c \vee X = d}\ \vee\mathrm{R} \qquad \vdash\ f(c) = f(X)}{\vdash\ (X = c \vee X = d) \wedge f(c) = f(X),\ \ldots}\ \wedge\mathrm{R}}{\vdash\ \exists x.\ ((x = c \vee x = d) \wedge f(c) = f(x))}\ \exists\mathrm{R}$$

At this point, it can be read off from the two top-most sequents that the proof can be closed by applying the substitution $\{X \mapsto c\}$. It can also be seen, however, that finding the right substitution is not always a simple task. When trying to use the equation $X = d$ for closing the left branch, applying the substitution $\{X \mapsto d\}$, a dead end would be reached and it would be necessary to backtrack or to introduce further metavariables and instances of the quantified formula.

Incremental Closure In [42, 50], an alternative to the destructive application of substitutions is discussed, which removes the need for backtracking. The method works by collecting substitution candidates for the individual proof branches, without immediately applying the substitutions. The avoidance of backtracking is, in particular, advantageous for proof systems that can be used both automatically and interactively. Empirical results [50] show that it can also be a basis for realising automated state-of-the-art theorem provers. Incremental closure is extensively used in the third paper of the thesis about showing the incorrectness

of programs (page 115), where constraints describe pre-states (program inputs) that reveal bugs in a program.

For the left branch in the previous example, here two *unification constraints* are derived as substitution candidates and stored for this branch. Analogously, one constraint is created for the right branch:

$$\dfrac{\dfrac{\dfrac{[\,X \equiv c\,],\ [\,X \equiv d\,]}{\vdash\ X = c,\ X = d}}{\vdash\ X = c \vee X = d}\ \vee\mathrm{R} \qquad \dfrac{[\,f(c) \equiv f(X)\,]}{\vdash\ f(c) = f(X)}}{\dfrac{\vdash\ (X = c \vee X = d) \wedge f(c) = f(X),\ \ldots}{\vdash\ \exists x.\ ((x = c \vee x = d) \wedge f(c) = f(x))}\ \exists\mathrm{R}}\ \wedge\mathrm{R}$$

In order to close the whole proof, it is now necessary to find constraints for all open branches that are compatible, which in this case are the two constraints $X \equiv c$ and $f(c) \equiv f(X)$. The constraint $X \equiv c \wedge f(c) \equiv f(X)$ is consistent and is solved by the substitution (the *unifier*) $\{X \mapsto c\}$ that is already known.

3 Dynamic Logic for Java

First-order dynamic logic (DL) [31] is a multi-modal extension of classical first-order predicate logic that is designed for reasoning about the I/O-relation of programs. The logic is comparable to Hoare-style logics and can be used for program verification in a very similar manner. DL contains two classes of modal operators: diamond formulae $\langle\alpha\rangle\phi$ express that the formula ϕ holds in at least one final state of program α. This means that $\langle\alpha\rangle\phi$ can only be true if α terminates. Box formulae can be regarded as abbreviations $[\alpha]\phi \equiv \neg\langle\alpha\rangle\neg\phi$, and express that the formula ϕ holds in all final states of the program α. The approach of considering programs as modal operators makes DL more flexible than Hoare-style logics, because modal operators can be nested arbitrarily with propositional connectives or quantifiers. It is also possible to make statements that involve more than one program, for instance can the pre- or post-conditions of a specification again contain programs. Examples for verification systems for Java that are based on DL are KIV [51] and KeY [36].

DL can be used for specifying programs in a manner similar to Hoare logics. A formula $\phi \rightarrow \langle\alpha\rangle\psi$ expresses the total correctness of the program α wrt. the pre-condition ϕ and the post-condition ψ (provided that α is deterministic). Likewise, $\phi \rightarrow [\alpha]\psi$ states the partial correctness of α: it is not required that α terminates. The partial correctness judgement $\phi \rightarrow [\alpha]\psi$ corresponds to a Hoare triple $\{\phi\}\alpha\{\psi\}$. In both DL formulae, the pre- and post-conditions ϕ and ψ can be arbitrary formulae of FOL, or can again contain programs.

Symbolic Execution On an intuitive level (and for deterministic programs), there is a strong correspondence between DL and the wp-calculus (Sect. 1.1): the formula $\langle\alpha\rangle\phi$ is the weakest pre-condition for the program α with post-condition ϕ. This analogy shows that the wp-calculus can directly be used as a calculus for

DL. Indeed, the Hilbert-style calculi for DL that are given in [31] can be identified as wp-calculus.

Nevertheless, the more common approach to reason in DL is symbolic execution (SE) and program analysis in forward direction (Sect. 1.1). When using a sequent calculus for DL to symbolically execute a program, path conditions can naturally be stored as side-formulae in the sequents. There are, however, different possibilities for representing the symbolic variable assignments that have to be maintained during SE. One way is to store variable assignments as equations, which means to implicitly carry out the inversion of right-hand sides of assignments that is shown in Sect. 1.1. This leads to the following assignment rule:

$$\frac{\Gamma[v/v'],\ v = E[v/v'] \ \vdash\ \phi,\ \Delta[v/v']}{\Gamma \ \vdash\ \langle v = E\rangle\phi, \Delta} \qquad v' \text{ fresh}$$

The article [35] presents a calculus for DL for JavaCard (based on the verification system KIV) that uses an assignment rule like this.

The second paper (page 89) of this thesis discusses the alternative approach that is used in KeY for handling variable assignments. *Updates* describe a set of assignments to program variables or to function symbols, with the result that the proof trees in a sequent calculus for DL almost directly correspond to SE trees as in Fig. 1. The assignment rule in a calculus with updates looks as follows:

$$\frac{\Gamma \ \vdash\ \{u\,;\ v := E\}\ \phi, \Delta}{\Gamma \ \vdash\ \{u\}\ \langle v = E\rangle\phi, \Delta}$$

In this setting, the actual program is preceded by an update u that determines the values of variables (or of the heap). An assignment $v = E$ is carried out by sequentially composing it with u. As an example, Fig. 2 shows how the program on page 6 can be symbolically executing in the resulting sequent calculus with updates. Generally, the first assignment rule (using equations) leads to a calculus the produces strongest post-conditions for given pre-conditions, whereas the second assignment rule with updates leads to a calculus that computes weakest pre-conditions for the given post-conditions of programs.

4 Implementation of Proof Assistants

Program logics like dynamic logic are typically implemented as *proof assistants*, in a similar manner as it is done for higher-order logics. In contrast to automated theorem provers, such systems are mainly used interactively, but usually offer also some degree of automation. Proof assistants also tend to provide a large number of rules (which can be derived rules or axioms) that need to be made accessible for the user.

"Pure" Proof Assistants There is a long history of rigorously implementing proof assistants based on a small number of axioms (like higher-order logic, set theory or type theory) and a meta-languages [52]. The first system to be designed

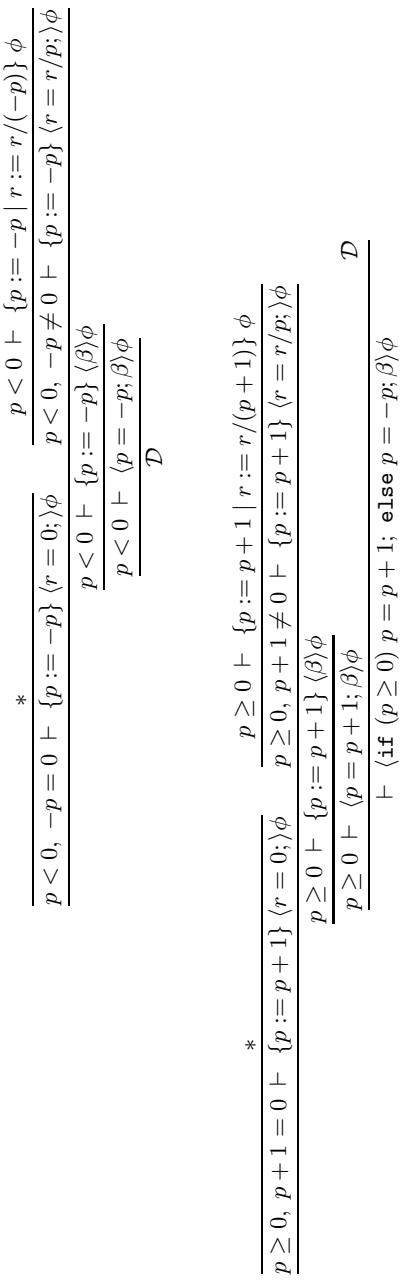


Fig. 2. Proof tree in a sequent calculus for dynamic logic using updates. The derivation resembles the SE tree of Fig. 1, the program is the one shown on page 6. We use the abbreviation β for the program `if` $(p = 0)$ $r = 0$; `else` $r = r/p$;. The post-condition ϕ can be an arbitrary formula.

like this was Edinburgh LCF, for which the functional programming language ML was invented as meta-language. *Tactics*, in this context, are ML functions operating on proof goals that can be invoked for goal-directed proof construction. Because it is possible to combine tactics arbitrarily (using ML) in order to create new tactics that realise larger proof steps or proof search, tactical systems are very flexible, but also non-trivial to use. Furthermore, the concept of creating a large number of composed and complex rules from a small initial set of rules prohibits unsoundness at at early stage and has proven to be extremely reliable. Further examples for pure proof assistants are Isabelle/HOL [10], Coq [11], and Nuprl [53].

"Pragmatic" Proof Assistants There are also proof assistants that follow a less rigorous and more pragmatic approach, in which it is possible to rather freely introduce new axioms instead of deducing everything from first principles. The most well-known example for such assistants is PVS [12], but also KeY [36] follows this approach. Such provers allow to introduce and define new domains typically faster than "pure" proof assistants, but it can be difficult to ensure the consistency of introduced rules.

The first paper (page 23) introduces *taclets*, which form the rule language that is used in the prover KeY. Taclets can be used both to introduce new axioms and to define lemmas that are derivable from existing rules. The paper also describes how it is possible to reason about the soundness of taclets, which can be employed to cross-validate taclets by comparing them with alternative axiomatisations of domains (like the semantics of a programming language), or to derive lemmas from other taclets.

5 Overview

Paper 1: Taclets — A Language for Sequent Calculi

We introduce the concept of *taclets*, which is the rule language that is used to implement the calculus for Java dynamic logic [54] in KeY [36]. Apart from a detailed introduction of the language and complete definitions of the semantics of taclets, reasoning about the correctness of taclets is discussed. The paper is the most complete account on taclets so far and examines and documents features that have been developed and implemented over the last years by the KeY project, in parts by the author.

The paper is in parts based on the journal article [55] and the workshop paper [56], which are coauthored by this author.

This paper is going to appear as the chapter "Construction of Proofs" in the forthcoming book "Verification of Object-Oriented Software: The KeY Approach", edited by Bernhard Beckert, Reiner Hähnle and Peter H. Schmitt, Springer LNCS.

Paper 2: Sequential, Parallel, and Quantified Updates of First-Order Structures

This paper describes the concept of *updates*, which is the central component for performing symbolic execution in Java dynamic logic [54]. Updates are systematically developed as an imperative programming language that provides the following constructs: assignments, guards, sequential composition and bounded as well as unbounded parallel composition. The language is equipped both with a denotational semantics and a correct rewriting system for execution, whereby the latter is a generalisation of the syntactic application of substitutions. The normalisation of updates is discussed. All results and the complete theory of updates have been formalised and proven using the Isabelle/HOL proof assistant [10].

This paper is an extended version of the paper accepted at the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Phnom Penh, Cambodia, which will appear in Springer LNCS.

Paper 3: Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic

In this paper, we use Java dynamic logic to prove the *incorrectness* of programs. In order to carry out disproving in Java dynamic logic, we use the concept of quantified updates together with existential quantification over algebraic datatypes. We show that this approach, carried out in a sequent calculus for dynamic logic, creates a connection between calculi and proof procedures for program verification and test data generation procedures. In comparison, starting with a program logic enables to find more general and more complicated counterexamples for the correctness of programs.

This paper has been written together with Muhammad Ali Shah, and is in parts based on the Master's thesis of Muhammad Ali Shah [57], who was supervised by the author, and on the workshop paper [58] by the author. The paper has been submitted to the International Conference on Tests And Proofs (TAP), ETH Zürich, Switzerland.

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. 2nd edn. Addison Wesley (2000)
2. Sun Microsystems, Inc. Palo Alto/CA, USA: Java Card 2.2 Platform Specification. (2002)
3. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
4. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C.: JML Reference Manual. (2002)
5. Object Modeling Group: Object Constraint Language Specification, version 1.1. (1997)

6. Nelson, G.: A generalization of Dijkstra's calculus. ACM Transactions on Programming Languages and Systems **11** (1989) 517–561
7. Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. Prentice Hall (1992)
8. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (1996)
9. Augustsson, L.: Cayenne—a language with dependent types. In: Proceedings, ICFP, New York, NY, USA, ACM Press (1998) 239–250
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
11. Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., Werner, B.: The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France (1993) Version 5.8.
12. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In Alur, R., Henzinger, T.A., eds.: Proceedings, CAV. Volume 1102 of LNCS., Springer (1996) 411–414
13. Hanbing Liu, J Strother Moore: Java program verification via a JVM deep embedding in ACL2. In Konrad Slind, Annette Bunker, Ganesh Gopalakrishnan, eds.: Proceedings, TPHOLs. Volume 3223 of LNCS., Springer (2004) 184–200
14. Jacobs, B., van den Berg, J., Huisman, M., van Berkum, M., Hensel, U., Tews, H.: Reasoning about Java classes: Preliminary report. In: Proceedings, OOPSLA '98, ACM Press (1998) 329–340
15. Arts, T., Chugunov, G., Dam, M., Fredlund, L., Gurov, D., Noll, T.: A tool for verifying software written in Erlang. Int. Journal of Software Tools for Technology Transfer **4** (2003) 405–420
16. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
17. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings, PLDI. (2002) 234–245
18. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: Post Conference Proceedings, CASSIS, Marseille. Volume 3362 of LNCS., Springer (2005) 49–69
19. Burdy, L., Requet, A., Lanet, J.: Java applet correctness: a developer-oriented approach. In: Proceedings, FME. Volume 2805 of LNCS., Springer (2003) 422–439
20. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (2003)
21. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/Javacard programs annotated in JML. Journal of Logic and Algebraic Programming **58** (2004) 89–106
22. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
23. Gedell, T., Hähnle, R.: Automating verification of loops by parallelization. In: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNAI, Springer (2006) To appear.
24. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. International Journal on Software Tools for Technology Transfer **2** (2000) 410–425
25. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10** (2003) 203–232
26. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings, PLDI. (2001) 203–213

27. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: Proceedings, TACAS. Volume 2619 of LNCS., Springer (2003) 553–568
28. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580
29. Meyer, J., Poetzsch-Heffter, A.: An architecture for interactive program provers. In: Proceedings, TACAS, Springer (2000) 63–77
30. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience **13** (2001) 1173–1214
31. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
32. Leino, K.R.M.: Efficient weakest preconditions. Information Processing Letters **93** (2005) 281–288
33. Wirsing, M.: Algebraic specification. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). (1990) 675–788
34. McCarthy, J.: Towards a mathematical theory of computation. In: Proceedings, IFIP Congress 62, Amsterdam, North-Holland (1963) 21–28
35. Stenzel, K.: A formally verified calculus for full JavaCard. In Rattray, C., Maharaj, S., Shankland, C., eds.: Proceedings, Algebraic Methodology and Software Technology (AMAST) 2004. Volume 3116 of LNCS., Springer (2004)
36. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY Tool. Software and System Modeling **4** (2005) 32–54
37. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings, LICS, IEEE Computer Society (2002) 55–74
38. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ISSTA. (2002) 123–133
39. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (2001) 22
40. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35** (2000) 268–279
41. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the Second Conference on Computer Science and Engineering in Linkping, ECSEL (1999) 21–28
42. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
43. Apt, K.R.: Ten years of Hoare's logic: A survey — part 1. ACM Transactions on Programming Languages and Systems **3** (1981) 431–483
44. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In Furbach, U., Shankar, N., eds.: Proceedings, IJCAR, Seattle, USA. LNCS, Springer (2006) To appear.
45. Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier Science B.V. (2001)
46. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52** (2005) 365–473
47. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In Baumgartner, P., Fermueller, C., eds.: Model Computation - Principles, Algorithms, Applications, Miami, Florida, CADE-19 Workshop (2003)
48. Gentzen, G.: Untersuchungen über das Logische Schliessen. Mathematische Zeitschrift **39** (1935) 176–210, 405–431 English translation in [59].

49. Hähnle, R.: Tableaux and related methods. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier Science B.V. (2001) 101–178
50. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, IJCAR, Siena, Italy. Volume 2083 of LNAI., Springer (2001) 545–560
51. Heisel, M., Reif, W., Stephan, W.: Program verification by symbolic execution and induction. In Morik, K., ed.: Proceedings, 11th German Workshop on Artifical Intelligence. Volume 152 of Informatik Fachberichte., Springer (1987)
52. Gordon, M.: From LCF to HOL: a short history. Proof, language, and interaction: essays in honour of Robin Milner (2000) 169–185
53. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Development System. Prentice-Hall, NJ (1986)
54. Beckert, B.: A dynamic logic for the formal verification of JavaCard programs. In Attali, I., Jensen, T., eds.: Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France. Volume 2041 of LNCS., Springer (2001) 6–24
55. Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Roth, A., Rümmer, P., Schlager, S.: Taclets: a new paradigm for constructing interactive theorem provers. Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas **98** (2004) Special Issue on Symbolic Computation in Logic and Artificial Intelligence.
56. Bubel, R., Roth, A., Rümmer, P.: Ensuring correctness of lightweight tactics for JavaCard dynamic logic. In: Informal Proceedings of Workshop on Logical Frameworks and Meta-Languages (LFM) at IJCAR 2004. (2004) 84–105
57. Shah, M.A.: Generating counterexamples for Java dynamic logic. Master's thesis (2005)
58. Rümmer, P.: Generating counterexamples for Java Dynamic Logic. In: Informal Proceedings of Workshop on Disproving at CADE 20. (2005)
59. Szabo, M.E., ed.: The Collected Papers of Gerhard Gentzen. North-Holland, Amsterdam (1969)

Paper 1

Taclets — A Language for Sequent Calculi

Philipp Rümmer

Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden
philipp@cs.chalmers.se

Abstract. We introduce the concept of *taclets*, which is the rule language that is used to implement the calculus for JAVA dynamic logic [1] in KeY. Apart from a detailed introduction of the language and complete definitions of the semantics of taclets, reasoning about the correctness of taclets is discussed. The paper is the most complete account on taclets in KeY so far.

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

Creating a proof assistant requires formalising the rules that the implemented calculus consists of. In our setting—in particular looking at calculi for dynamic logic [2]—such a formalisation is subject to a number of requirements:

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

only using mouse clicks and drag and drop. This puts a limit on the complexity that a single rule should have for keeping the required user interaction clear and simple, and it requires that rules also contain "pragmatic" information that describes how the rules are supposed to be applied. An accounts on the user interface in KeY is [3].
- The formalisation also has to enable the automation of as many proof tasks as possible. This covers the simplification of formulae and proof goals, the symbolic execution of programs (which usually does not require user interaction) as well as automated proof or decision procedures for simpler fragments of the logic and for theories. The approach followed in KeY is to have global *strategies* that give priorities to the different applicable rules and automatically apply the rule that is considered most suitable. This concept is powerful enough to implement complete proof procedures for first-order logic[1] and to handle theories like linear integer arithmetic or polynomial rings mostly automatically.

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

Historically, taclets have first been devised in [5, 6] as "Schematic Theory Specific Rules", with the main purpose of capturing the axioms of theories and algebraic specifications as rules. The language is general enough, however, to also cover all rules of a first-order sequent calculus and most rules of calculi for dynamic logic. The development of taclets as a way to build interactive provers was influenced to a large degree by the theorem prover InterACT [7], but also has strong roots in more traditional methods like tactics and derived rules that are commonly used for higher-order logics (examples for such systems are Isabelle/HOL, see [8], Coq, see [9], or PVS, see [10]). Compared to tactics, the expressiveness of taclets is very limited, for the reasons mentioned above. A further difference is that taclets do not (explicitly) build on a small and fixed set of primitive rules, as tactics do in (foundational) higher-order frameworks like Isabelle, but that a rather large number of taclets is considered as *axioms* that are simply assumed.

A recent conceptual introduction to taclets is given by [11]. The article lacks, however, many details of how taclets currently are used in KeY, because the taclet concept has constantly been extended over the last years in order to implement the rules of JAVA CARD DL. This chapter gives a more comprehensive description of taclets as they now exist in KeY, and also includes features that were only

[1] KeY does not use backtracking, the implemented procedure rather follows the non-destructive approach of [4].

KeY

```
\functions {
  integer exp(integer, integer);
}
\schemaVariables {
  \term integer a, b;
}
\rules {
  expZero { \find(exp(a, 0)) \replacewith(1) };
  expSucc { \find(exp(a, b)) \sameUpdateLevel
                             \replacewith(a * exp(a, b-1));
                             \add(==> b > 0) };
}
```

KeY

Fig. 1. Axiomatisation of an Exponentiation Function on Integers using Taclets

added recently. At the same time, even in the scope of this chapter many details had to be left out.

When would a KeY User introduce own Taclets? or
When to read this Chapter?

In most cases it is not necessary for a user of the theorem prover KeY to define taclets, because KeY already comes with complete implementations of the calculi for first-order and dynamic logic. There are, nevertheless, situations when the introduction of new taclets can be valuable:

- The introduction of lemmas, i.e., of non-axiom taclets that can be derived from existing rules, can help to structure complex proofs. Such taclets can be written to an external file and be loaded on demand. When lemmas are loaded, proof obligations that ensure soundness (⇒ Sect. 5) are automatically created by KeY as new proof tasks and have to be proven using already existing taclets. This means that lemmas only can add convenience, but do not increase the set of derivable formulae. Instead of applying lemmas, one could as well apply more basic rules, but this usually leads to a longer and more intricate proof. Typical examples are lemmas about complex arithmetic transformations.
- Assumptions under which a conjecture is to be proven can be formulated as taclets. For verifying an algorithm, we might, for instance, want to introduce an exponentiation function *exp* through the clauses

  $$exp(a, 0) \;=\; 1, \qquad exp(a, b) \;=\; a \cdot exp(a, b-1) \quad (b > 0)$$

  While the two equations can, in principle, simply be added as (quantified) formulae to the conjecture in question, having a large number of such definitions would clutter proofs and would also be very tedious to apply. Fig. 1

KeY

```
\schemaVariables {
  \term integer a, b, c;
}
\rules {
  expSplit { \find(exp(a, b)) \sameUpdateLevel
                              \replacewith(exp(a, b-c) * exp(a, c));
                              \add(==> c >= 0);
                              \add(==> b >= c) };
}
```

KeY

Fig. 2. Lemma for the Exponentiation Function

shows how *exp* can instead be defined with two simple taclets that can be used in a proof exactly like the ordinary rules of a calculus. The keywords and clauses of the taclets are explained in detail later in this chapter (⇒ Sect. 4).
Note, that the two taclets of Fig. 1 are not lemmas but *axioms*: the normal rules of a calculus will not tell us anything about the function *exp* and will in particular not entail that *exp* actually describes exponentiation. For this reason, such axioms cannot be loaded on demand while proving but have to be defined as part of a problem file that can be loaded by KeY. Based on the axioms, in turn, lemmas can be defined, loaded at a later point, and then also proven correct. An example for such a lemma is the following identity (the corresponding taclet is shown in Fig. 2):

$$exp(a, b) \;=\; exp(a, b - c) \cdot exp(a, c) \quad (c \geq 0,\, b \geq c)$$

– More generally, new theories can be defined and axiomatised through appropriate taclets, which is the original intention of the taclet concept. This is described in more detail by [5, 6]. Typical examples would be algebraic datatypes like lists, finite sets or trees, the laws of which can naturally be captured with taclets.

The next pages will give all information that is required to write taclets for these purposes. In the whole chapter, we assume that the reader already knows about *sequents* $\Gamma \vdash \Delta$ and about their meaning, for an introduction see [12].

Organisation of the Chapter

We continue with introducing the concepts and keywords of taclets informally in Sect. 1: we look at number of taclets that implement rules for first-order logic and dynamic logic. After that, Sect. 2 provides a complete account on schema variables. The two sections (Sect. 1 and 2) together with Sect. 3 about metavariables contain all practical information that is necessary for developing new taclets. An

in-depth introduction of the taclet language and a discussion of the soundness aspect of taclets are given in the two remaining sections (Sect. 4 and 5).

1 Taclets by Example

The next pages give a tour through the taclet language and illustrate the most important features with examples. We organise the section along logics of increasing complexity: 1. propositional logic, the fragment of first-order predicate logic that is obtained by removing quantifiers, variables and terms, 2. first-order predicate logic, and 3. dynamic logic for JAVA CARD (JAVA CARD DL). Many of the taclets discussed here correspond to rules that are given in Fig. 3. As a convention, in this chapter we use `typewriter` both for schema variables (in order to distinguish them from normal variables x, y) and for taclet names (to distinguish them from rules like allLeft).

Propositional Rules as Taclets

The first example is the taclet `close` (⇒ Fig. 4) representing an axiom that closes a branch of a proof (corresponding to rule close in Fig. 3). It can be applied whenever the sequent of a proof leaf contains the same formula both in antecedent and succedent. The taclet makes use of two different keywords of the taclet language:

- `\assumes` imposes a condition on the applicability of the taclet and has a sequent as parameter. In the case of `close`, the `\assumes` clause states that the taclet must only be applied if an arbitrary formula `phi` appears both in antecedent and succedent of a sequent (the sequent may very well contain further formulae).
- `\closegoal` specifies that an application of the taclet closes a proof branch.

The expression in an `\assumes` clause (like all expressions that turn up in a taclet) may contain *schema variables* like the variable `phi`. A schema variable has a *kind* that defines which expressions the variable can stand for (a precise definition is given in Sect. 2). In our example, `phi` represents an arbitrary formula. More generally, the taclet language provides schema variables that are necessary for all first-order logics, e.g. kinds for matching variables, terms, and formulae. Further kinds are necessary for rules of dynamic logic and enable variables representing program entities (like JAVA statements or expressions).

Note 1. The keywords of the taclet language reflect the direction in which sequent calculus proofs are constructed: we start with a formula that is supposed to be proven and create a tree upwards by *analysing* the formula and taking it apart. Taclets describe expansion steps (or, as a border case, closure steps), and by the *application* of a taclet we mean the process of adding new nodes to a leaf of a proof tree following this description.

$$\textsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta} \qquad \textsf{impLeft}\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \to \psi \vdash \Delta}$$

$$\textsf{allRight}\ \frac{\Gamma \vdash [x/c](\phi), \Delta}{\Gamma \vdash \forall x.\phi, \Delta} \qquad \textsf{allLeft}\ \frac{\Gamma, \forall x.\phi, [x/t](\phi) \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}$$

with c a new constant of type A, if $x : A$. — with t a ground term of type A', $A' \sqsubseteq A$, if $x : A$.

$$\textsf{exLeft}\ \frac{\Gamma, [x/c](\phi) \vdash \Delta}{\Gamma, \exists x.\phi \vdash \Delta} \qquad \textsf{exRight}\ \frac{\Gamma \vdash \exists x.\phi, [x/t](\phi), \Delta}{\Gamma \vdash \exists x.\phi, \Delta}$$

with c a new constant of type A, if $x : A$. — with t a ground term of type A', $A' \sqsubseteq A$, if $x : A$.

$$\textsf{close}\ \frac{}{\Gamma, \phi \vdash \phi, \Delta}$$

$$\textsf{eqLeft}\ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \vdash \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \vdash \Delta} \quad \text{if } \sigma(t_2) \sqsubseteq \sigma(t_1).$$

$$\textsf{eqRight}\ \frac{\Gamma, t_1 \doteq t_2 \vdash [z/t_2](\phi), [z/t_1](\phi), \Delta}{\Gamma, t_1 \doteq t_2 \vdash [z/t_1](\phi), \Delta} \quad \text{if } \sigma(t_2) \sqsubseteq \sigma(t_1).$$

Fig. 3. A Selection of classical first-order rules, taken from [13]

KeY

```
\schemaVariables {
  \formula phi, psi;
}
\rules {
  close    { \assumes(phi ==> phi) \closegoal };
  impRight { \find(==> phi -> psi) \replacewith(phi ==> psi) };
  cut      { \add(phi ==>); \add(==> phi) };
  mpLeft   { \assumes(phi ==>) \find(phi -> psi ==>)
                               \replacewith(psi ==>) };
}
```

KeY

Fig. 4. Examples of Taclets implementing Propositional Rules

In order to describe rules that modify formulae of a sequent, the taclet language offers keywords for specifying which expression a taclet works on (the *focus* of the taclet application) and in which way it is modified. Taclet `impRight` (⇒ Fig. 4) corresponds to rule impRight in Fig. 3 and contains clauses to this end:

- `\find` defines a pattern (in this taclet `phi` → `psi`, where `phi`, `psi` are again schema variables) that must occur in the sequent to which the taclet is supposed to be applied. Accordingly, `impRight` can be applied whenever an implication turns up in the succedent[2] of a proof leaf.
- `\replacewith` tells how the focus of the application will be altered, which for `impRight` means that an implication `phi` → `psi` in the succedent will be removed, that the formula `phi` is added to the antecedent and that `psi` is added to the succedent. In general, when a taclet with a `\replacewith` clause is applied, a new proof goal is created from the previous one by replacing the expression matched in the `\find` part with the expression in the `\replacewith` part (after substituting the correct concrete expressions for schema variables).

Besides rules that modify a term or a formula, there are also rules that add new formulae (but not terms) to a sequent. A typical example is the *cut-rule*, which is a rule with two premisses that makes a case distinction on whether a formula `phi` is true or false:

$$\mathsf{cut}\ \frac{\Gamma, \phi \vdash \Delta \qquad \Gamma \vdash \phi, \Delta}{\Gamma \vdash \Delta}$$

Taclet `cut` (⇒ Fig. 4) shows how case distinctions like this can be realised in the taclet language and contains a keyword that has not turned up so far:

- `\add` specifies formulae that are added to a sequent when the taclet is applied. Similarly to `\replacewith`, the argument of `\add` is a sequent that gives a list of formulae to be added to the antecedent and a second list to be added to the succedent.

The taclet `cut` also shows how taclets can be written for rules that have more than one premiss and split a proof branch into two branches. The clauses that belong to different branches are in the taclet separated by semicolons. In case of `cut`, an application will create two new proof goals and add `phi` to the antecedent in one of the goals and to the succedent in the other one.

The examples above exclusively contained either `\replacewith` clauses or `\add` clauses. It is, however, legal to use both in a taclet, and often they are interchangeable. We could—without changing the meaning of the taclet—write taclet `impRight` also in the following way:

[2] In a sequent $\Gamma \vdash \Delta$ we call the left part Γ the *antecedent* and the right part Δ the *succedent*.

Taclet

```
impRightAdd { \find(==> phi -> psi) \replacewith(==> psi)
                                    \add(phi ==>) };
```

Taclet

Similarly, `\assumes` and `\find` can be combined for specifying that a formula can be modified provided that certain other formulae occur in the sequent. An example is the rule known as *modus ponens*: if a formula `phi` and the implication `phi` → `psi` hold, then also `psi` will hold. Because the converse is true as well—if `phi` and `psi` hold, then also `phi` → `psi`—we can safely eliminate the implication:

$$\mathsf{mpLeft}\ \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi, \phi \to \psi \vdash \Delta}$$

The taclet `mpLeft` (⇒ Fig. 4) implements this rule. If the formulae `phi` and `phi` → `psi` both occur in the antecedent of a sequent, then the taclet is applicable and `phi` → `psi` can be replaced with `psi`. The assumption `phi` will not be altered by the taclet application.

First-Order Rules as Taclets

Dealing with a calculus for first-order logic (as opposed to propositional logic) using the taclet approach requires handling terms and variables, in particular schema variables for variables and for terms are necessary. As an example, we consider the rule allLeft (⇒ Fig. 3) for universal quantifiers, which is implemented by the taclet `allLeft` (⇒ Fig. 5). Apart from a variable `phi` for formulae, we need the following schema variables:

- `x` representing logical variables of type `G` that can be bound by a quantifier.
- `s` representing an arbitrary (ground) term with static type `G` (or a subtype of `G`).

The `\find` clause of `allLeft` specifies that the rule is applied to universally quantified formulae of the antecedent. Upon application, the taclet adds an instance of the formula to the antecedent by substituting term `s` for the quantified variable `x`. Because `s` does not turn up in the `\find` clause of the taclet, it can essentially be chosen arbitrarily when applying the taclet, reflecting the nature of the allLeft rule.

The taclet `allLeft` demonstrates a further feature: the rule allLeft is supposed to be applicable for arbitrary static types A of the quantified variable and the substituted term. This is realised in taclets by introducing a type `G` that is marked as `\generic` and that can stand for arbitrary "concrete" types A, in the same way as a schema variable can represent concrete formulae or terms.

The analogue of `allLeft` is the taclet `allRight` (⇒ Fig. 5) that realises the rule with same name in Fig. 3. In contrast to `allLeft`, here the original quantified formula can be *replaced* with an instance in which a fresh[3] constant is

[3] A symbol is called *fresh* for a given proof if it does not occur in the proof.

KeY

```
\sorts {
  \generic G;
}
\schemaVariables {
  \formula phi;      \variables G x;          \skolemTerm G cnst;
  \term G s, t;      \term[strict] G t2;      \term integer intTerm;
}
\rules {
  allLeft   { \find (\forall x; phi ==>)
              \add ({\subst x; s}phi ==>) };
  allRight  { \find (==> \forall x; phi)
              \varcond (\new(cnst, \dependingOn(phi)))
              \replacewith (==> {\subst x; cnst}phi) };
  zeroRight { \find (intTerm + 0) \replacewith (intTerm) };
  removeAll { \find (\forall x; phi) \varcond (\notFreeIn(x, phi))
              \replacewith (phi) };
  applyEq   { \assumes (t = t2 ==>) \find(t) \sameUpdateLevel
                                    \replacewith(t2) };
  applyEqAR { \find (t = t2 ==>)
              \addrules ( rewrWithEq { \find (t) \sameUpdateLevel
                                       \replacewith (t2) } ) };
}
```

KeY

Fig. 5. Examples of Taclets implementing First-Order Rules

substituted for the bound variable. There is a particular kind of schema variable for introducing new constants that can be used here: variable `cnst` is defined as a variable of kind `\skolemTerm` and will always represent a fresh constant or function symbol that does not yet turn up in the proof in question. The strange line `\varcond(\new(cnst,\dependingOn(phi)))` becomes important in the presence of metavariables (see Sect. 2.1 and 3, where detailed explanations are given) and ensures that all metavariables that occur in `phi` also are arguments of the function symbol.

Rewriting Taclets

In all of the taclets that we have looked at so far, the parameter of `\find` clauses were sequents containing exactly one formula. For implementing many first-order rules it is, however, necessary also to modify expressions (formulae or terms) *inside* of formulae, leaving the surrounding formula or term unchanged. Examples are most of the equality rules in Fig. 3, where we can use an equation $s \doteq t$ for replacing the term s with t anywhere in a sequent. The taclets that we can use for making rules like this available are called *rewriting taclets*: now, the argument of `\find` is a single formula or term and does not contain the arrow `==>`.

A first and very simple rewriting taclet is `zeroRight` (⇒ Fig. 5). It states that 0 is the right identity of addition. In `zeroRight`, the `\find` expression is a term. As we obviously cannot replace terms with formulae, in order to make the taclet well-formed then also `\replacewith` expressions have to be terms (if the `\find` expression were a formula, also `\replacewith` expressions would have to be).

Using `zeroRight`, we can for instance conduct the following proof, where the taclet is used to turn $p(a + 0)$ into $p(a)$. Subsequently, the proof can be closed using close.

$$\dfrac{\dfrac{}{p(a) \vdash p(a)}}{p(a + 0) \vdash p(a)}$$

Both of the rules eqLeft and eqRight (⇒ Fig. 3) for applying equations are implemented by taclet `applyEq` (⇒ Fig. 5), because a rewriting taclet does not distinguish between a focus in the antecedent and in the succedent. The taclet again uses an `\assumes` clause for demanding the presence of certain formulae in a sequent, here of the appropriate equation in the antecedent, and a `\find` clause for specifying the terms that can be modified. `t` and `t2` are schema variables for the left and the right side of the equation.

When examining the rules eqLeft and eqRight carefully, we see that both rules also have a side-condition $\sigma(t_2) \sqsubseteq \sigma(t_1)$ that demands that the type of t_2 is a subtype of the type of t_1. This condition is captured in the taclet `applyEq` by declaring the schema variable `t` as `strict`:

– The option `strict` demands that the term that is represented by `t` *exactly* has type `A`, otherwise also subtypes would be allowed.

Because `t2` is non-`strict`, also subtypes are allowed, which means that the condition $\sigma(t_2) \sqsubseteq \sigma(t_1)$ is met.

Implementing the rules for applying equations in a sound way—also for dynamic logic—requires a further feature of the taclet language:

– `\sameUpdateLevel` is a *state condition* and can only be added to rewriting taclets. This clause ensures that the focus of the taclet application (the term that is represented by `t` in `\find`) does not occur in the scope of modal operators apart from updates. Updates are allowed above the focus, but in this case the equation `t` $\doteq$ `t2`—or, more generally, all formulae referred to using `\assumes`, `\replacewith` and `\add`—have to be in the scope of the same[4] update.

This keyword is necessary for `applyEq`, because too liberal an application of equations is not sound in dynamic logic. In order to illustrate the effect of `\sameUpdateLevel`, we consider two potential applications of `applyEq`:

Illegal:
$$\frac{x \doteq v+1 \vdash \{v := 2\}p(v+1)}{x \doteq v+1 \vdash \{v := 2\}p(x)}$$

Legal:
$$\frac{\{v := 2\}(x \doteq v+1) \vdash \{v := 2\}p(v+1)}{\{v := 2\}(x \doteq v+1) \vdash \{v := 2\}p(x)}$$

We have to rule out the left application (by adding the flag `\sameUpdateLevel`) because the equation $x \doteq v+1$ must not be used in the state that is created by the update $v := 2$. The right application is admissible, however, because here the equation is preceded by the same update and we know that it holds if v has value 2.

Compared with the rules of Fig. 3, `applyEq` differs in a further aspect: while the rule eqRight, for instance, only *adds* new formulae to a sequent, leaving the original formulae untouched, the taclet `applyEq` will directly and destructively modify formulae. Taclets cannot immediately capture the copy-behaviour of eqRight. We will show later in this section how the behaviour of eqRight can be simulated.

Sometimes it is necessary to impose conditions on the variables that may turn up (or not turn up) in formulae or terms involved. For this purpose, the taclet language offers the keyword `\varcond` that is illustrated in the rewriting taclet `removeAll`. The taclet eliminates universal quantifiers, provided that the variable that is quantified over does not occur in the scope of the quantifier. Because the taclet is a rewriting taclet, it can also be applied in situations in which ordinary quantifier elimination (using rules like `allRight`) is not possible, namely if a quantifier is not top-level.

[4] It is enough if the updates in front of the different constituents have the same effect. This can be determined more or less liberally, like by checking for syntactic identity or by taking laws of updates into account.

Nested Taclets

Taclets have restricted higher-order features: it is possible to write taclets that upon application introduce further taclets, i.e., make further taclets available for proof construction.

– `\addrules` has as argument a list of taclets that will be made available when the parent taclet is applied. `\addrules` is used similarly to `\add`, in particular it is possible to introduce different taclets in each branch that a taclet creates.

Consider the taclet `applyEqAR` (⇒ Fig. 5), which is an alternative taclet for handling equations and is essentially equivalent to `applyEq`. If the antecedent contains an equality that can be matched by `t` $\doteq$ `t2`, then applying the taclet results in a new rewriting taclet that replaces a term matched by `t` with a term matched by `t2`. For the equation $f(a) \doteq b$, for instance, we would obtain the following additional taclet (the whole truth is, however, more complicated and explained in Sect. 4.6):

Taclet

```
rewrWithEq { \find (f(a)) \sameUpdateLevel \replacewith (b) };
```

Taclet

This means that the actual application of an equality is now performed by two taclets. Due to the `\addrules`-clause, the set of available taclets is not fixed but can grow dynamically during the course of a proof. Note that the generated taclets are not sound in general: `rewrWithEq` above is only locally rendered sound by the presence of the equation $f(a) \doteq b$ in the antecedent (but it will stay sound in children of the sequent). Soundness of taclets is discussed in Sect. 5. The flag `\sameUpdateLevel` is set in `rewrWithEq` for the same reason as for `applyEq`, but now entails that `rewrWithEq` can only be applied in the same state (below the same updates) as the taclet `applyEqAR` by which it was introduced.

Using `\addrules`, we can also store formulae that might be needed again later in a proof in the form of a taclet (before applying destructive modifications) or hide formulae:

Taclet

```
saveLeft  { \find (phi ==>)
            \addrules( insert { \add (phi ==>) } ) };
hideLeft  { \find (phi ==>) \replacewith (==>)
            \addrules( insert { \add (phi ==>) } ) };
```

Taclet

Rules of Dynamic Logic as Taclets

So far, we have shown examples for taclets representing calculus rules for propositional and first-order logic. However, taclets are not restricted to these two logics

KeY

```
\schemaVariables {
  \formula phi;
  \program SimpleExpression #se;       \program Statement #s0, #s1;
}
\rules {
  ifElseSplit { \find (==> \<{.. if(#se) #s0 else #s1 ...}\>phi)
     "if␣#se␣true":  \replacewith (==> \<{.. #s0 ...}\>phi)
                     \add (#se = TRUE ==>);
     "if␣#se␣false": \replacewith (==> \<{.. #s1 ...}\>phi)
                     \add (#se = FALSE ==>) };
}
```

KeY

Fig. 6. Example of a Taclet implementing a Rule of Dynamic Logic

but can also be used for formally capturing the rules of JAVA dynamic logic [1]. A taclet for handling the if-then-else statement in JAVA CARD is `ifElseSplit` (⇒ Fig. 6). The basic idea of the taclet is to split the if-then-else statement into two statements representing the possible branches. As it was done for formulae, terms and variables, the taclet makes use of schema variables within programs, in this case a schema variable `#se` for side-effect free expressions and `#s0`, `#s1` representing program statements.

As we have seen on page 33, it is possible to apply a rewriting taclet containing the keyword `\sameUpdateLevel` only if the application focus and the formulae referred to using `\assumes`, `\replacewith` and `\add` are in the scope of the same update. The same holds for taclets that are not rewriting taclets, i.e., where the `\find` pattern is a sequent or there is no `\find` clause, although it is not necessary to include the flag `\sameUpdateLevel` explicitly for such taclets. The following application of `ifElseSplit` is possible, in which the update $v := a$ occurs in front of all affected formulae:

$$\frac{a \geq 0 \leftrightarrow b = \text{TRUE},\ \{v := a\}\ (b = \text{TRUE}) \vdash \{v := a\} \langle\ \texttt{v++;}\ \rangle v > 0 \qquad a \geq 0 \leftrightarrow b = \text{TRUE},\ \{v := a\}\ (b = \text{FALSE}) \vdash \{v := a\} \langle\ \texttt{v=-v;}\ \rangle v > 0}{a \geq 0 \leftrightarrow b = \text{TRUE} \vdash \{v := a\} \langle\ \texttt{if (b) v++; else v=-v;}\ \rangle v > 0}$$

Another feature of taclets used in `ifElseSplit` are the dots ../... surrounding the program. These dots can be considered a further kind of schema variable and stand for the context in which a statement (here the conditional statement that is eliminated by `ifElseSplit`) occurs, which can be certain blocks around the statement (like `try`-blocks) and arbitrary trailing code.

Finally, the taclet `ifElseSplit` shows how the different branches that a taclet creates can be given names for convenience reasons. In the KeY implementation, these strings (like `"if #se true"`) are used as annotations in the proof tree and can make navigation easier. As can be seen here, branch names may also contain

schema variables (`#se`) that will be replaced with their concrete instantiation when the taclet is applied.

The remaining chapter is a more systematic and less tutorial-like account on taclets. When taking a step back, we see that the approach is based on two notions or principles that are mostly orthogonal to each other:

- *Schema variables*, the special kind of variables that is used as wildcards in expressions. Schema variables are a concept that also occurs unrelated to taclets, for instance when writing rules in textbook notation, and are a general means of describing *rule schemas*. As this chapter targets the actual implementation of rules within theorem provers, it is, however, necessary to develop the notion of schema variables in a more formal manner.
- A simple and high-level imperative language for modifying sequents. A program in this language—a *taclet*—describes conditions when a modification is possible, where it is possible, a list of modification statements for adding, removing and modifying formulae, and means of branching and closing proof goals. Speaking in terms of sequent calculi, taclets are suitable for describing and also practically executing almost arbitrary local rules.

Most parts of the chapter relate to these two concepts: Sect. 2 is an account on schema variables, whereas the actual taclet language is defined in Sect. 4.

2 Schema Variables

Despite the name *variable*, schema variables are in the context of KeY a very broad concept: they comprise a large number of different kinds of placeholders that can be used in taclets. All schema variables have in common that they are wildcards for syntactic entities, which can again be different kinds of variables (like logical variables or program variables), terms, formulae, programs or more abstract things like modal operators.

Schema variables are used to define taclets. When a taclet is applied, i.e., when a goal of a proof is modified by carrying out the steps described by the taclet, the contained schema variables will be replaced by *concrete* syntactic entities. This process is called *instantiation* and ensures that schema variables never occur in the proof itself. Instantiation is formally defined in Sect. 2.3. In order to ensure that no ill-formed expressions occur while instantiating schema variables with concrete expressions, e.g. that no formula is inserted at a place where only terms are allowed, the *kind* of a schema variable defines which entities the schema variable can represent and may be replaced with.

Example 1. In KeY syntax, we declare `phi` to be a schema variable representing formulae and `n` a variable for terms of type *integer*:

Table 1. Kinds of schema variables in the context of a type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$

`\variables` A	Logical variables of type $A \in \mathcal{T}$
`\term` A	Terms of type $B \sqsubseteq A$ (with $A \in \mathcal{T}$)
`\formula`	Formulae
`\skolemTerm` A	Skolem constants/functions of type $A \in \mathcal{T}$
`\program` t	Program entities of type t (from Table 2)
`\modalOperator` M	Modal operators that are elements of set M
`\programContext`	Program context

KeY

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

KeY

The kinds of schema variables that exist in the KeY system are given in Table 1. A more detailed explanation of each of the different categories is given in Sect. 2.1. Table 1 has been found to be rather stable during the development of KeY in the last years and is not expected to be modified a lot in the future. The part that in our experience uses to be altered most frequently (e.g. for adding support for further features of a programming language or completely new languages) are the different types of program entities that can be described with variables of kind `\program` t (see Table 2).

For the following definition of schema variables, we first introduce the notion of a *type hierarchy*:

Definition 1 (Type hierarchy [13]). *A* type hierarchy *is a tuple* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *of*

- *a finite set of* static types $\mathcal{T}$,
- *a finite set of* dynamic types $\mathcal{T}_d$,
- *a finite set of* abstract types $\mathcal{T}_a$, *and*
- *a* subtype relation $\sqsubseteq$ *on* $\mathcal{T}$,

such that

- $\mathcal{T} = \mathcal{T}_d \,\dot\cup\, \mathcal{T}_a$
- *There is an* empty type $\bot \in \mathcal{T}_a$ *and a* universal type $\top \in \mathcal{T}_d$.
- $\sqsubseteq$ *is a reflexive partial order on* $\mathcal{T}$, *i.e. for all types* $A, B, C \in \mathcal{T}$,

$$A \sqsubseteq A$$
$$\text{if } A \sqsubseteq B \text{ and } B \sqsubseteq A \text{ then } A = B$$
$$\text{if } A \sqsubseteq B \text{ and } B \sqsubseteq C \text{ then } A \sqsubseteq C$$

- $\bot \sqsubseteq A \sqsubseteq \top$ *for all* $A \in \mathcal{T}$.
- $\mathcal{T}$ *is closed under greatest lower bounds w.r.t.* $\sqsubseteq$*, i.e. for any* $A, B \in \mathcal{T}$*, there is an*[5] $I \in \mathcal{T}$ *such that* $I \sqsubseteq A$ *and* $I \sqsubseteq B$ *and for any* $C \in \mathcal{T}$ *such that* $C \sqsubseteq A$ *and* $C \sqsubseteq B$*, it holds that* $C \sqsubseteq I$*. We write* $A \sqcap B$ *for the greatest lower bound of* A *and* B *and call it the* intersection type *of* A *and* B*. The existence of* $A \sqcap B$ *also guarantees the existence of the least upper bound* $A \sqcup B$ *of* A *and* B*, called the* union type *of* A *and* B*.*
- *Every non-empty abstract type* $A \in \mathcal{T}_a \setminus \{\bot\}$ *has a non-abstract subtype:* $B \in \mathcal{T}_d$ *with* $B \sqsubseteq A$*.*

We say that A is a subtype of B *if* $A \sqsubseteq B$*. The set of non-empty static types is denoted by* $\mathcal{T}_q := \mathcal{T} \setminus \{\bot\}$*.*

Definition 2. *Let* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *by a type hierarchy. A set* SV *of* schema variables *over* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *is a set of symbols that are distinct from all other declared symbols, where each schema variable* $\texttt{sv} \in SV$ *has exactly one of the kinds from Table 1 over* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*.*

2.1 The Kinds of Schema Variables in Detail

We can roughly distinguish two different categories of schema variables, those which belong to first-order logic (the upper part of Table 1) and the more special kinds that are used to write taclets for dynamic logic.

Variables: `\variables` A

Schema variables for variables can be instantiated with logical variables that have static type A. In contrast to schema variables for terms, logical variables of subtypes of A are not allowed, as such a behaviour has been found to make development of sound taclets difficult (⇒ Sect. 5). Schema variables can also occur bound in formulae—which actually is the most common usage—and also bound occurrences will be replaced with concrete variables when instantiations are applied (this is illustrated in Example 4 below).

Terms: `\term` A

Schema variables for terms can be instantiated with arbitrary terms that have the static type A or a subtype of A. Subtypes are allowed because this behaviour is most useful in practice: there are only very few rules for which the static type of involved terms *exactly* has to match some given type (in case the reader wants to implement a rule like this, the modifier `strict` (⇒ Sect. 2.5) can be used). In general, there are no conditions on the logical variables that may occur (free) in terms substituted for such schema variables. When a term schema variable is in the scope of a quantifier, logical variables can be "captured" when applying the instantiation, which needs to be considered when writing taclets (again, this is illustrated in Example 4 below).

[5] It is well-known that the greatest lower bound is unique if it exists.

Formulae: `\formula`

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

Skolem Terms: `\skolemTerm` A

Schema variables for Skolem terms are supposed to be instantiated with terms of the form

$$f_{\mathrm{sk}}(X_1, \ldots, X_n)$$

with a fresh function symbol f_{sk} of type A and a list $X_1, \ldots, X_n$ of *metavariables* as arguments. Metavariables (or *free variables*) are a means of postponing the instantiation of schema variables for terms, which is essential for automated proving, and are described in Sect. 3. In most cases, namely if no metavariables are involved, the term will degenerate to a fresh constant c_{sk} of type A.

The taclet application mechanism in KeY will simply create new function symbols when applying a taclet that contains such schema variables. This ensures that the inserted symbols are fresh for a proof and hence can be used as Skolem symbols. In order to determine the arguments $X_1, \ldots, X_n$ of a Skolem term, the variable conditions `\new(sk,\dependingOn(te))` ($\Rightarrow$ Sect. 2.6) have to be used—adding such conditions can be necessary for ensuring that taclets are sound. For more details see Sect. 3.

In practice, there are only few rules that use schema variables for Skolem terms, and most probably the reader will never make use of them in his or her own taclets.

Program Entities: `\program` t

There is a large number of different kinds of program entities that can be represented using program schema variables. Table 2 contains the most important ones of the types that existed when this chapter was written, for a list that is more complete and up-to-date we refer to the homepage[6] accompanying [13]. In KeY, the name of a program schema variable always has to start with a hash (like `#se`), mostly for the purpose of parsing schematic programs.

Modal Operators: `\modalOperator` M

When implementing rules for dynamic logic, very often the same rule should be applicable for different modal operators. In most cases the versions of rules for box and diamond operators, for instance, do not differ apart from the fact that different modalities are used. In this situation, having to define essentially the same rule multiple times would be very inconvenient. The problem gets worse with the introduction of further modal operators.

[6] `www.key-project.org/thebook`

Table 2. A selection of the kinds of schema variables for program entities

Expressions	
`Expression`	Arbitrary JAVA expressions
`SimpleExpression`	Side-effect free expressions: 1. a local program variable, or 2. a static attribute that is a compile-time constant, or 3. a literal, or 4. an `instanceof` expression, where the first argument is a local program variable or a static compile-time-constant attribute, or 5. a `this` reference
`NonSimpleExpression`	`Expression`, but not `SimpleExpression`
`Java*Expression`	The same as `SimpleExpression`, but in addition the type of an expression has to be `*`, which can be `Boolean`, `Byte`, `Char`, `Short`, `Int` or `Long`
Left-Hand Side Expressions	
`Variable`	Certain kinds of expressions that—because of syntactic restrictions—cannot have any side-effects: 1. local program variables, or 2. static attributes that are compile-time constants
`StaticVariable`	Static attributes without prefix or with a side-effect free prefix
`LeftHandSide`	Expressions whose only side-effect—because of syntactic restrictions—can be the static initialisation of a class: 1. a `Variable`, or 2. a `StaticVariable`, or 3. a non-static attribute without prefix or with prefix `this`
Statements	
`Statement`	A single arbitrary JAVA statement
`Catch`	A `catch` clause of a `try` block
Types	
`Type`	Arbitrary JAVA type references
`NonPrimitiveType`	The same as `Type`, but not the primitive types of JAVA
Miscellaneous	
`Label`	A JAVA label

Table 3. Modal Operators that exist in KeY

`diamond`, `box`	Standard operators
`throughout`	"Throughout" modality
`diamond_trc`, `box_trc`, `throughout_trc`, `diamond_tra`, `box_tra`, `throughout_tra`, `diamond_susp`, `box_susp`, `throughout_susp`	Modalities used for handling JAVA CARD transactions

More concise definitions of rules for a variety of modal operators use schema variables that represent groups of modal operators. An overview of the modalities that exist in KeY is given in Table 3, and the syntax for such schema variables is illustrated in the following example:

Example 2. We implement the most basic assignment rule for dynamic logic (for assignments with side-effect free left- and right-hand side, see [13]):

$$\textsf{assignment}\ \frac{\vdash \mathcal{U}\{loc := val\}\langle \pi\ \ \omega\rangle\varphi}{\vdash \mathcal{U}\langle \pi\ \ loc\ \texttt{=}\ val\texttt{;}\ \ \omega\rangle\varphi}$$

This rule is shown here for the diamond modal operator, but it can be formulated in the same way for other modalities. In taclets, this can be realised by introducing a variable `#normalMod` that represents exactly the admissible operators. The syntax for using such schema variables in taclets is

```
\modality{<variable>}{ <program> }\endmodality (<postcondition>)
```

The complete declaration of the taclet is given in Fig. 7. Note that we do not have to include the update $\mathcal{U}$ explicitly, because updates in front of the formulae that a taclet operates on are allowed by default (⇒ Sect. 4.1).

KeY

```
\schemaVariables{
  \formula phi; \program Variable #loc; \program SimpleExpression #se;
  \modalOperator { diamond, box, diamond_trc,
                   box_trc, throughout_trc } #normalMod;
}
\rules{
  assign { \find (        ==> \modality{#normalMod}
                                  {.. #loc = #se; ...}
                              \endmodality(phi) )
           \replacewith ( ==> {#loc:= #se}
                              \modality{#normalMod}
                                  {.. ...}
                              \endmodality(phi) ) };
}
```

KeY

Fig. 7. Assignment taclet from Example 2

Program Context: `\programContext`

Context schema variables cannot be declared explicitly. Instead, there is always at most one variable of this kind that is hidden behind the π and ω in a formula:

$$\langle \pi\ \ p\ \ \omega\rangle\phi$$

In KeY syntax, π and ω are simply written as dots:

```
\<.. p ...\> phi
```

We will use the notation $\pi/\omega \in SV$ to talk about the context schema variable itself. An instantiation of the context schema variable π/ω is a pair (α, β) of two program fragments, where the "left half" α only consists of opening braces, opening try blocks and similar "inactive" parts of JAVA, and the "right half" β is a continuation that closes all blocks that were opened in α.

Example 3. We can use context schema variables to enclose a program statement of interest in a context of blocks and following statements. A possible choice for the program fragments α and β is shown here:

$$\underbrace{\texttt{try \{}}_{\alpha} \texttt{ x = y; } \underbrace{\texttt{f(13); \} finally \{ x = 0; \}}}_{\beta}$$

2.2 Schematic Expressions

For all families of expressions introduced in [13], like terms and formulae of first-order logic or of JAVA CARD DL, programs, sequents, etc. we can introduce corresponding *schematic* versions in which appropriate schema variables can be used as surrogates for concrete sub-expressions. For obvious reasons, however, we do not want to repeat all definitions given so far. We will only give, as an example, a simplified definition of schematic terms and formulae. The augmentation with further connectives is obvious. What is also left out in this chapter is the definition of *schematic JAVA programs*, which are defined in the same spirit as the other schematic expressions. We first need to introduce the notion of a *signature* for predicate logic:

Definition 3 (Signature [13]). *Given a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*, a* signature *is a quadruple* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *of*

- *a set of* set of variable symbols VSym*,*
- *a set of* function symbols FSym*,*
- *a set of* predicate symbols PSym*, and*
- *a typing function* α*,*

such that[7]

- $\alpha(v) \in \mathcal{T}_q$ *for all* $v \in \mathrm{VSym}$*,*
- $\alpha(f) \in \mathcal{T}_q^* \times \mathcal{T}_q$ *for all* $f \in \mathrm{FSym}$*, and*
- $\alpha(p) \in \mathcal{T}_q^*$ *for all* $p \in \mathrm{PSym}$*.*
- *There is a function symbol* $(A) \in \mathrm{FSym}$ *with* $\alpha((A)) = ((\top), A)$ *for any* $A \in \mathcal{T}_q$*, called the* cast to type A*.*
- *There is a predicate symbol* $\doteq \in \mathrm{PSym}$ *with* $\alpha(\doteq) = (\top, \top)$*.*

[7] We use the standard notation A^* to denote the set of (possibly empty) *sequences* of elements of A.

– *There is a predicate symbol* $\sqsubseteq A \in \mathrm{PSym}$ *with* $\alpha(\sqsubseteq A) = (\top)$ *for any* $A \in \mathcal{T}$*, called the* type predicate for type A*.*

We use the following notations:

– $v : A$ *for* $\alpha(v) = A$*,*
– $f : A_1, \ldots, A_n \to A$ *for* $\alpha(f) = ((A_1, \ldots, A_n), A)$*, and*
– $p : A_1, \ldots, A_n$ *for* $\alpha(p) = (A_1, \ldots, A_n)$*.*

A constant symbol *is a function symbol* c *with* $\alpha(c) = ((), A)$ *for some type* A*.*

Definition 4 (Basic schematic terms and formulae). *Suppose that the tuple* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *is a signature for the type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *and SV a set of schema variables over* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*.*

The system of sets $\{SchemaTerms_A\}_{A \in \mathcal{T}}$ *of* schematic terms of static type A *is inductively defined as the least system of sets such that*

– `sv` $\in SchemaTerms_A$ *for any schema variable* `sv` $\in SV$ *of kind* `\variables` A*,* `\term` A *or* `\skolemTerm` A*,*
– $f(t_1, \ldots, t_n) \in SchemaTerms_A$ *for any function symbol* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}$ *and terms* $t_i \in SchemaTerms_{A'_i}$ *with* $A'_i \sqsubseteq A_i$ *for* $i = 1, \ldots, n$*, and*
– $\{\backslash subst$ `va`; $s\}(t) \in SchemaTerms_A$ *for any schema variable* `va` $\in SV$ *of kind* `\variables` B *and terms* $s \in SchemaTerms_{B'}$*,* $t \in SchemaTerms_A$ *with* $B' \sqsubseteq B$*.*

The set SchemaFormulae of Java Card DL formulae *is inductively defined as the least set such that*

– `phi` $\in SchemaFormulae$ *for any schema variable* `phi` $\in SV$ *of kind* `\formula`*,*
– $p(t_1, \ldots, t_n) \in SchemaFormulae$ *for any predicate symbol* $p : A_1, \ldots, A_n \in \mathrm{PSym}$ *and terms* $t_i \in SchemaTerms_{A'_i}$ *with* $A'_i \sqsubseteq A_i$ *for* $i = 1, \ldots, n$*,*
– true, false, $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \to \psi \in SchemaFormulae$ *for any* $\phi, \psi \in SchemaFormulae$*,*
– $\forall$`va`$.\, \phi$, $\exists$`va`$.\, \phi \in SchemaFormulae$ *for any* $\phi \in SchemaFormulae$ *and any schema variable* `sv` $\in SV$ *of kind* `\variables`*.*
– $\{\backslash subst$ `va`; $s\}(\phi) \in SchemaFormulae$ *for any schema variable* `va` $\in SV$ *of kind* `\variables` B*, term* $s \in SchemaTerms_{B'}$ *with* $B' \sqsubseteq B$ *and formula* $\phi \in SchemaFormulae$*.*

Note 2. According to this definition, schematic expressions never contain logical variables, the set VSym is not used. While this is not a strict necessity, it has been found that *not* considering the case where logical and schema variables simultaneously turn up in expressions significantly simplifies the following sections. Concrete (non-schema) variables are fortunately not necessary for writing taclets, as they can always be replaced with schema variables of kind `\variables`. The actual logical variables that a taclet operates on are then only determined when the taclet is applied, i.e., when schema variables are replaced with concrete expressions (⇒ Sect. 2.3).

Note 3. Substitutions {\subst va; s} are here introduced as *syntactic* constructs and not directly as operations on terms or formulae. This becomes necessary as substitutions are used in taclets and have to be given a formal syntax. Further, we also allow the case that s contains schema variables of kind `\variables`, which corresponds to substituting terms that contain free variables (non-ground substitutions). Only considering ground substitutions would be an unreasonable restriction for taclets, but, as a downside, the application of non-ground substitutions is more involved. Sect. 2.3 and 2.4 describe how substitutions are eliminated when instantiating expressions.

2.3 Instantiation of Schema Variables and Expressions

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

Definition 5 (Instantiation of Schema Variables). *Suppose that the quadruple* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *be a signature for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *and SV a set of schema variables over* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*. An* instantiation *of SV is a partial mapping*

$$\iota : SV \nrightarrow \big(Formulae \cup \bigcup_{A \in \mathcal{T}} Terms_A \cup Programs\big)$$

that assigns concrete syntactic entities to schema variables in accordance with Tables 1 and 2. An instantiation is called complete *for SV if it is a total mapping on SV.*

For sake of brevity, we will also talk about instantiations of (schematic) terms, formulae or programs (⇒ Def. 4), which really are instantiations of the set of schema variables that turn up in the expression. Given a complete instantiation of such an expression—which in general is more complex than only a single schema variable—we can turn the expression into a concrete one by replacing all schema variables `sv` with their concrete value $\iota(\texttt{sv})$. The extension of ι to arbitrary schematic expressions makes use of a further prerequisite, *(possibly non-ground) substitutions* $[x/s](t)$, which is provided in Sect. 2.4 but follows the same idea as the ground substitutions in [13]. Again, the corresponding definition for instantiation of schematic programs is left out.

Definition 6 (Instantiation of Terms and Formulae). *Let ι be a complete instantiation of SV. We extend ι to arbitrary schematic terms over SV:*

- $\iota(f(t_1, \ldots, t_n)) := f(\iota(t_1), \ldots, \iota(t_n))$
- $\iota(\{\backslash subst\ \texttt{va};\ s\}(t)) := [\iota(\texttt{va})/\iota(s)](\iota(t))$

Likewise, ι is extended to schematic formulae over SV:

Table 4. Examples of schematic expressions and their instantiation

t	ι	$\iota(t)$
$f(\texttt{te})$	$\{\texttt{te} \mapsto g(a)\}$	$f(g(a))$
$f(\texttt{va})$	$\{\texttt{va} \mapsto x\}$	$f(x)$
$\forall \texttt{va}.\, p(\texttt{va})$	$\{\texttt{va} \mapsto x\}$	$\forall x.\, p(x)$
$\forall \texttt{va}.\, p(\texttt{te})$	$\{\texttt{va} \mapsto x, \texttt{te} \mapsto x\}$	$\forall x.\, p(x)$
$\forall \texttt{va}.\, \texttt{phi}$	$\{\texttt{va} \mapsto x,\ \texttt{phi} \mapsto p(x)\}$	$\forall x.\, p(x)$
$\texttt{phi} \wedge p(\texttt{te})$	$\{\texttt{phi} \mapsto q \vee r,\ \texttt{te} \mapsto f(a)\}$	$(q \vee r) \wedge p(f(a))$
$p(\texttt{sk}) \rightarrow \exists \texttt{va}.\, p(\texttt{va})$	$\{\texttt{sk} \mapsto c,\ \texttt{va} \mapsto x\}$	$p(c) \rightarrow \exists x.\, p(x)$
$\{\backslash\text{subst } \texttt{va};\ \texttt{sk}\}(\texttt{phi}) \rightarrow \exists \texttt{va}.\, \texttt{phi}$	$\{\texttt{sk} \mapsto c,\ \texttt{va} \mapsto x,\ \texttt{phi} \mapsto p(x)\}$	$p(c) \rightarrow \exists x.\, p(x)$

- $\iota(p(t_1, \ldots, t_n)) := p(\iota(t_1), \ldots, \iota(t_n))$
- $\iota(\text{true}) := \text{true}$ *and* $\iota(\text{false}) := \text{false}$,
- $\iota(\neg\phi) := \neg\iota(\phi)$,
- $\iota(\phi \wedge \psi) := \iota(\phi) \wedge \iota(\psi)$ *(and correspondingly for* $\phi \vee \psi$ *and* $\phi \rightarrow \psi$*)*,
- $\iota(\forall \texttt{va}.\, \phi) := \forall \iota(\texttt{va}).\, \iota(\phi)$ *and* $\iota(\exists \texttt{va}.\, \phi) := \exists \iota(\texttt{va}).\, \iota(\phi)$,
- $\iota(\{\backslash subst\ \texttt{va};\ s\}(\phi)) := [\iota(\texttt{va})/\iota(s)](\iota(\phi))$.

Example 4. Table 4 illustrates the instantiation of the different kinds of schema variables for first-order logic. We use the following schema variables:

KeY

```
\schemaVariables {
  \variables A va;          \term A te;
  \formula phi;             \skolemTerm A sk;
}
```

KeY

Apart from that, we assume that $f, g : A \rightarrow A$ are function symbols, $a, c : A$ are constants, $p : A$ and q, r are predicates and $x : A$ is a logical variable. The most interesting instantiation takes place in the last line of Table 4, where first the schema variables are replaced with terms and variables and then the substitution is applied:

$$\begin{aligned} &\iota(\{\backslash\text{subst } \texttt{va};\ \texttt{sk}\}(\texttt{phi}) \rightarrow \exists \texttt{va}.\, \texttt{phi}) \\ &\quad = \quad [x/c](p(x)) \rightarrow \exists x.\, p(x)) \quad = \quad p(c) \rightarrow \exists x.\, p(x) \end{aligned}$$

Note 4. Example 4 demonstrates the interrelation between schema variables `\variables`, `\term` and `\formula`. Instantiations of variables of the latter two kinds, like $\iota(\texttt{phi}) = p(x)$, may contain free logical variables that also schema variables `va` of kind `\variables` are instantiated with. Such occurrences can become bound when evaluating an expression like $\iota(\forall \texttt{va}.\, \texttt{phi})$, effectively turning `\term` and `\formula` variables into higher-order variables: the variable `phi`

represents a predicate with the formal argument `va`. This feature is essential for taclets like `allLeft` (⇒ Fig. 5) or induction rules. A more thorough discussion is given in Sect. 4.2.

2.4 Substitutions Revisited

Substitutions are syntactic operations on terms or formulae that replace variables with terms and that, similarly to schema variables, are always eliminated when a rule is applied. Substitutions never turn up in the actual proofs.

In the context of a general rule language like taclets, the restriction to ground substitutions (as it is done in the first chapters of [13]) would often be a real limitation. We might, for instance, formulate a taclet that eliminates existential quantifiers if the only possible solution can directly be read off:

Taclet

```
uniqueEx { \find (\exists va; (va=t & phi))
           \varcond (\notFreeIn (va, t))
           \replacewith ({\subst va; t}phi) };
```

Taclet

The taclet is a rewriting taclet and can also be applied in the scope of quantifiers, for instance in

$$\frac{\vdash \forall x.\, \forall z.\, x \doteq z}{\vdash \forall x.\, \exists y.\, (y \doteq x \wedge \forall z.\, y \doteq z)}$$

where the expression $[y/x](\forall z.\, y \doteq z)$ has to be evaluated. Note that the substitution applied here is not ground.

Unfortunately, application of non-ground substitutions can raise problems that do not occur in the ground case. While both formulae given in the example above are obviously not valid, the following slight modification of the conclusion (to which `uniqueEx` is applied) shows that the taclet is not sound if the substitution is carried out naively. We rename the innermost bound variable to x, which does not alter the meaning of the lower formula:

$$\frac{\vdash \forall x.\, \forall x.\, x \doteq x}{\vdash \forall x.\, \exists y.\, (y \doteq x \wedge \forall x.\, y \doteq x)}$$

Surprisingly, the result of applying `uniqueEx` is a valid formula (the premiss above the bar). A rule that draws invalid conclusions from valid premisses, however, is not sound. The cause of unsoundness is the application of the substitution in $[y/x](\forall x.\, y \doteq x)$, which turns a formula that is not valid into a valid one. This phenomenon is known as *variable capture* or *collision* and occurs whenever a term containing a (free) variable x is moved into the scope of a quantifier like $\forall x.$ (or any operator binding x). Because the transformation changes the place where x is bound, also the meaning of an expression will be altered drastically.

It is possible to circumvent variable capture by suitable *bound renaming*, i.e., by renaming quantified variables when the danger of captured variables

arises. The concept of bound renaming often occurs when working with bound variables: giving variables names (or an identity) is only a means to determine the place where a variable is bound. One variable can always be exchanged (or renamed) with another (unused) variable. This is also known as *α-conversion*. Such a renaming can as well be performed deeply within formulae. The KeY implementation performs bound renaming automatically whenever it is necessary. In order to ensure that we can always pick a variable that is fresh for some expression or proof, in this section we assume that a signature $(\mathrm{VSym}, \mathrm{FSym}_r \cup \mathrm{FSym}_{nr}, \mathrm{PSym}_r \cup \mathrm{PSym}_{nr}, \alpha)$ always contains infinitely many variables for each type.

Definition 7. *A* substitution *is a function τ that assigns (possibly non-ground) terms to some finite set of variable symbols* $\mathrm{dom}(\tau) \subseteq \mathrm{VSym}$*, the* domain *of the substitution, with the restriction that*

If $v \in \mathrm{dom}(\tau)$ and $v : B$, then $\tau(v) \in Terms_A$, for some A with $A \sqsubseteq B$.

We write $\tau = [u_1/t_1, \ldots, u_n/t_n]$ for the substitution that has the domain $\mathrm{dom}(\tau) = \{u_1, \ldots, u_n\}$ and $\tau(u_i) := t_i$.

We denote by τ_x the result of removing a variable from the domain of τ, i.e. $\mathrm{dom}(\tau_x) := \mathrm{dom}(\tau) \setminus \{x\}$ and $\tau_x(v) := \tau(v)$ for all $v \in \mathrm{dom}(\tau_x)$.

When extending substitutions to arbitrary terms or formulae, the only interesting and new case are quantifiers, where it can be necessary to perform renaming.

Definition 8. *The application of a substitution τ is extended to non-variable terms by the following definitions:*

- $\tau(x) := x$ *for a variable* $x \notin \mathrm{dom}(\tau)$.
- $\tau(f(t_1, \ldots, t_n)) := f(\tau(t_1), \ldots, \tau(t_n))$.

The application of a substitution τ to a formula is defined by

- $\tau(p(t_1, \ldots, t_n)) := p(\tau(t_1), \ldots, \tau(t_n))$.
- $\tau(\mathrm{true}) := \mathrm{true}$ *and* $\tau(\mathrm{false}) := \mathrm{false}$.
- $\tau(\neg\phi) := \neg(\tau(\phi))$,
- $\tau(\phi \wedge \psi) := \tau(\phi) \wedge \tau(\psi)$, *and correspondingly for* $\phi \vee \psi$ *and* $\phi \rightarrow \psi$.
- *If there exists* $y \in \mathrm{dom}(\tau) \setminus \{x\}$ *such that* $x : A$ *occurs in* $\tau(y)$*, then* $\tau(\forall x.\, \phi) := \forall z.\, \tau([x/z]\phi)$ *for a fresh variable* $z : A \in \mathrm{VSym}$*. Otherwise,* $\tau(\forall x.\, \phi) := \forall x.\, \tau_x(\phi)$*. (Correspondingly for* $\exists x.\, \phi$*)*

Expressions modulo Bound Renaming

An elegant approach to bound renaming—that often occurs in literature—is to always work with the equivalence classes of formulae modulo bound renaming. When following this notion, two formulae like $\forall x.\, p(x)$ and $\forall y.\, p(y)$ are different representatives of the same equivalence class and are considered as *the same* formula. While definitions usually get shorter and less technical this

Table 5. Modifiers for Schema Variables

Modifier	*Applicable to*	
`rigid`	`\term` A `\formula`	Terms or formulae that can syntactically be identified as rigid
`strict`	`\term` A	Terms of type A (and not of proper subtypes of A)
`list`	`\program` t	Sequences of program entities. The type t can be any of the types of Table 2 apart from `Label`, `Type` and `NonPrimitiveType`. Sequences of expressions can be used to represent arguments of method invocations.

way because bound renaming does not have to be handled explicitly anymore, the differences between the approaches are negligible for an implementation.

With these definitions, we can repeat the previously unsound application of `uniqueEx` and now get a correct result (that we also would obtain when using KeY):

$$\frac{\vdash \forall x.\, \forall u.\, x \doteq u}{\vdash \forall x.\, \exists y.\, (y \doteq x \land \forall x.\, y \doteq x)}$$

where u is an arbitrary unused variable.

2.5 Schema Variable Modifiers

Some of the schema variable kinds come in more than one flavour: it is possible to change the set of concrete expressions that are represented by the schema variable using certain *modifiers*. Such modifiers can restrict the instantiations allowed for a variable further, or can also modify the meaning of a kind more drastically. The KeY prover currently implements the three modifiers that are given in Table 5.

Example 5. We define `phi` to represent exclusively *rigid* formulae (instead of arbitrary formulae, see [13]), `te` to represent rigid terms that have *exactly* static type A (subtypes of A are not allowed), and `#slist` to represent a whole sequence of JAVA statements (separated by `;`).

KeY

```
\schemaVariables {
  \formula[rigid] phi;
  \term[rigid,strict] A te;
  \program[list] statement #slist;
}
```

KeY

Table 6. Schema Variable Conditions

First-Order Conditions	
`\notFreeIn(va, te)` `\notFreeIn(va, fo)`	The logical variable that is instantiation of `va` must not occur (free) in the instantiation of `te`/`fo`.
`\new(sk, \dependingOn(te))` `\new(sk, \dependingOn(fo))`	If `sk` is instantiated with $f_{\mathrm{sk}}(X_1, \ldots, X_n)$, then ensure that $X_1, \ldots, X_n$ contains all metavariables that occur in the instantiation of `te`/`fo`. There can be more than one such condition for `sk`. Also see Sect. 3.
Introducing Fresh Local Program Variables KeY will create a new program variable as instantiation of `#x` when a taclet with such a condition is applied.	
`\new(#x, *)`	`#x` will have the JAVA type `*` (for instance `int[]`).
`\new(#x, \typeof(#y))`	`#x` will have the same JAVA type as the instantiation of `#y`.
`\new(#x, \elemTypeof(#y))`	The instantiation of `#y` has to be of a JAVA array type. Its component type will be the type of `#x`.

- `va` has kind `\variables` t
- `te` has kind `\term` s
- `fo` has kind `\formula`
- `sk` has kind `\skolemTerm` t
- `#x`, `#y` have kind `\program LeftHandSide` or `\program Variable`

2.6 Schema Variable Conditions

The simple notion of *kinds* of schema variables described in the previous sections is often not expressive enough for writing useful taclets. In many cases, one has to impose further restrictions on the instantiations of schema variables, e.g. state that certain logical variables must not occur free in certain terms. The taclet formalism is hence equipped with a simple language for expressing such conditions, *variable conditions*. To each taclet, a list of variable conditions can be attached (⇒ Sect. 4.1) that will be checked when the taclet is about to be applied.

Table 6 contains the most important variable conditions in KeY. Particularly useful is the `\notFreeIn` condition that is frequently needed for defining theories using taclets.

2.7 Generic Types

Schema variables for terms, logical variables and Skolem terms are typed and may only be instantiated with terms or variables of certain static types. While such schema variables are in principle sufficient for implementing all rules of a calculus for dynamic logic, this would not be particularly convenient: for certain rules, a number of taclets would be required, one for each existing type. Example for such rules are allLeft, allRight (⇒ Fig. 3).

To handle this situation in a better way, the taclet formalism provides the possibility of writing *generic* taclets, i.e., taclets in which the types of schema variables involved are flexible and are assigned only when the taclet is applied. The concept resembles schema variables, which represent concrete syntactic entities and are instantiated when applying the taclet. When writing taclets, we thus distinguish between *concrete* types, which are exactly the types in Def. 1, and *generic* types that are mapped to concrete types when applying the taclet. Like schema variables, generic types can only be used for defining taclets and are not part of the actual signature of a logic.

Note 5. Generic types should not be confused with the *abstract* types of Def. 1, the two notions are not related. For technical reasons, we will in fact consider generic types as dynamic (i.e., non-abstract), but when applying a taclet a generic type can represent both abstract and non-abstract concrete types.

For introducing generic types we extend the notion of a type hierarchy. Because *generic type hierarchies* can still be seen as normal type hierarchies by simply ignoring the distinction between generic and concrete types, Def. 4 about schematic expressions does not have to be changed but also covers terms or formulae containing "generic parts".

Definition 9. *A* generic type hierarchy *is a tuple* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$ *of*

- *a set of* static types $\mathcal{T}$*,*
- *a set of* dynamic types $\mathcal{T}_d$*,*
- *a set of* abstract types $\mathcal{T}_a$*,*
- *a set of* generic types $\mathcal{T}_g$*,*
- *a* subtype relation $\sqsubseteq$*, and*
- *a* range relation $\mathcal{R}_g$ *of the generic types*

such that

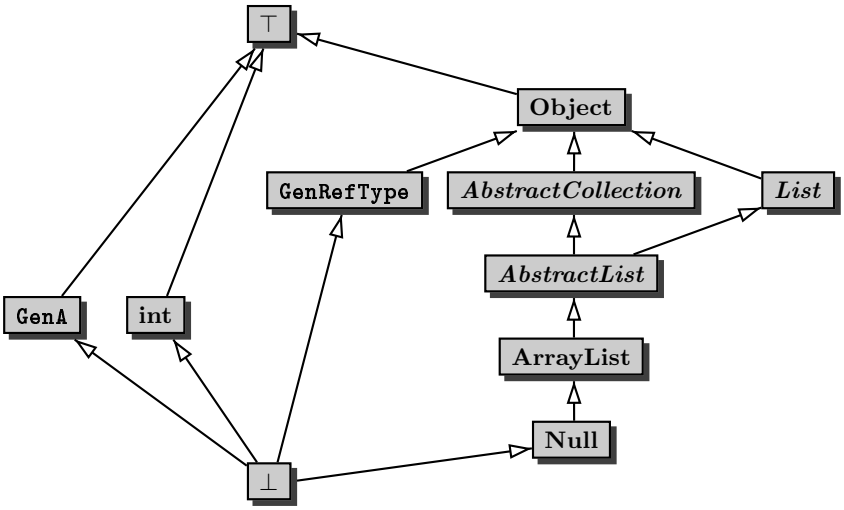
- *Each generic type is a dynamic type, but is not universal:* $\mathcal{T}_g \subseteq \mathcal{T}_d \backslash \{\top\}$
- $\mathcal{R}_g$ *is a relation between generic and concrete types:* $\mathcal{R}_g \subseteq \mathcal{T}_g \times (\mathcal{T} \backslash \mathcal{T}_g)$
- $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *is a type hierarchy (⇒ Def. 1)*
- *The concrete (non-generic) types also form a type hierarchy on their own:*

$$\big(\mathcal{T} \backslash \mathcal{T}_g,\ \mathcal{T}_d \backslash \mathcal{T}_g,\ \mathcal{T}_a,\ \sqsubseteq \cap \big((\mathcal{T} \backslash \mathcal{T}_g) \times (\mathcal{T} \backslash \mathcal{T}_g)\big)\big)$$

  is a type hierarchy
- *The subtypes* $A \sqsubseteq G$ *of generic types* $G \in \mathcal{T}_g$ *are either generic or empty:* $A \in \mathcal{T}_g \cup \{\bot\}$

Example 6. Fig. 8 shows a type hierarchy with two generic types:

- `GenA`, which is subtype of $\top$ and can thus be used to denote *arbitrary* concrete types in a taclet, and
- `GenRefType`, which can only represent reference types (subtypes of class Object).



KeY

```
\sorts {
  \generic GenA;
  \generic GenRefType \extends Object;
}
```

KeY

Fig. 8. An example type hierarchy

As range relation, we choose the full relation $\mathcal{R}_g = \mathcal{T}_g \times (\mathcal{T} \backslash \mathcal{T}_g)$. The figure also shows how to declare the two generic types in the concrete syntax of KeY (provided that type Object exists). After the declaration, we could use the types in taclets as illustrated in Sect. 1, for instance in order to implement the rule allLeft.

When a taclet containing generic types (i.e., containing schema variables with generic type) is applied, first an instantiation of these types with concrete types is chosen: all generic types that occur in the taclet are replaced with concrete types. It can then be checked whether instantiations of schema variables are allowed according to Table 1. Instantiations of generic types cannot be arbitrary, however, as the creation of ill-formed terms or formulae has to be prevented. Referring to the types of the previous example, a taclet could, for instance, contain the term $f(\texttt{te})$, where $f : \text{Object} \to \text{Object}$ is a function symbol and `te` a schema variable of kind `\term GenRefType` (note that `GenRefType` is a subtype of Object). It is obvious that we would run the risk of ill-formed terms if `GenRefType` was allowed to be instantiated with types that themselves are *not* subtypes of Object, because then also `te` could be replaced with terms whose type is not a subtype of Object. This insight is generalised by demanding that type instantiations always are *monotonic* wrt. the subtype relation.

Definition 10. *Given a generic type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$*, a* type instantiation *is a partial mapping* $\iota_t : \mathcal{T} \nrightarrow \mathcal{T}$ *such that*

- ι_t *is defined on concrete types* $A \in \mathcal{T} \backslash \mathcal{T}_g$*, which are fixed-points:* $\iota_t(A) = A$
- *Generic types* $G \in \mathcal{T}_g$ *are mapped to concrete types:* $\iota_t(G) \notin \mathcal{T}_g$ *(provided that* $\iota_t(G)$ *is defined)*
- *The mapping is monotonic: if* $A \sqsubseteq B$ *then also* $\iota_t(A) \sqsubseteq \iota_t(B)$ *(provided that* $\iota_t(A)$ *and* $\iota_t(B)$ *are defined)*
- *Generic types* $G \in \mathcal{T}_g$ *are mapped to types within its range:* $(G, \iota_t(G)) \in \mathcal{R}_g$ *(provided that* $\iota_t(G)$ *is defined)*

Example 7. For the type hierarchy that is declared in Example 6, one possible type instantiation is given by

$$\iota_t(\texttt{GenA}) = \top, \qquad \iota_t(\texttt{GenRefType}) = \mathit{AbstractList}, \qquad \iota_t(A) = A \quad (A \notin \mathcal{T}_g)$$

An instantiation $\iota_t(\texttt{GenRefType}) = \mathit{int}$ would not be allowed, because there is no monotonic extension of this mapping to the set $\mathcal{T}$ of all types that maps Object to itself.

Given the notion of a type instantiation, we can augment Def. 5 to also take schema variables of generic types into account. First, we can extend type instantiations ι_t to the kinds of schema variables:

$$\begin{aligned}
\texttt{\textbackslash term}\ A &\mapsto \texttt{\textbackslash term}\ \iota_t(A) \\
\texttt{\textbackslash variables}\ A &\mapsto \texttt{\textbackslash variables}\ \iota_t(A) \\
\texttt{\textbackslash skolemTerm}\ A &\mapsto \texttt{\textbackslash skolemTerm}\ \iota_t(A) \\
k &\mapsto k \qquad \text{(all other kinds)}
\end{aligned}$$

In the presence of generic types, instantiations are then described by a pair consisting of a schema variable instantiation and a type instantiation:

Definition 11. *Let* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *be a signature for a generic type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$ *and SV a set of schema variables over the same type system. An* instantiation under generic types *of a set SV of schema variables is a pair* (ι_t, ι)*, where*

- ι_t *is a type instantiation that is defined for all types of variables in SV, and*
- ι *is a partial mapping (as in Def. 5)*

$$\iota : SV \nrightarrow \big(Formulae \cup \bigcup_{A \in \mathcal{T}} Terms_A \cup Programs\big)$$

such that, for each schema variable $\mathtt{sv} \in SV$ *of kind* k *with* $\iota(\mathtt{sv}) \neq \bot$*,* $\iota(\mathtt{sv})$ *is an admissible instantiation for a schema variable of kind* $\iota_t(k)$*.*

Example 8. The purpose of *range relations* $\mathcal{R}_g$ is to provide a more direct control over the concrete types that can be chosen for generic ones. If we, for instance, would like to write a taclet that only can be applied for abstract types, we could strengthen the declaration of Fig. 8:

KeY

```
\sorts {
  \generic GenA;
  \generic GenRefType \extends Object
           \oneof { AbstractCollection, AbstractList, List };
}
\schemaVariables {
  \term[strict] GenRefType te;
}
```

KeY

The schema variable `te` would then exclusively represent terms of the types `AbstractCollection`, `AbstractList`, `List`. Leaving out the keyword `strict`, `te` could also stand for terms of the subtypes `ArrayList` and `Null`.

Generic Types vs. Schema Variables

The role of generic types is comparable to that of schema variables, and an alternative way to define taclets that are parametric over types would indeed be to have a concept of type schema variables. The two approaches only represent different views on the same idea. While the differences are negligible for an implementation, we believe that including generic types in the normal type hierarchy enables an easier theory and presentation: in this approach, also schematic terms that occur in a taclet are well-typed.

3 Instantiations and Metavariables—A Taster

Before a taclet can be applied, generic types and schema variables need to be instantiated. Selecting the expression that a schema variable `sv` represents is comparatively easy if the variable turns up in the `\find` or `\assumes` clause of a taclet: in this case, the expression already has to be part of the sequent to which the taclet is applied and can be found by *matching* the formulae of the sequent with the schematic terms or formulae of `\find` and `\assumes`. This situation also allows for a simple automated application of taclets.

There can also be schema variables that only turn up in the goal templates of a taclet, i.e., only in `\replacewith` or `\add` clauses. The most well-known example for such a rule is the taclet `allLeft` (⇒ Sect. 1), in which the schema variable for terms only occurs in `\add` (a further taclet containing such schema variables is `expSplit` (⇒ Fig. 2)). This means that the term that is represented by `s` can be chosen arbitrarily when applying the taclet. While making this choice can already be a difficult task for the human user of a proof assistant, the automated application of the taclet is even more hampered, and it is necessary to put a large amount of "intelligence", heuristics and knowledge about the particular problem domain into an automatic search strategy for guessing the right terms. The problem is also made difficult by the fact that the terms that need to be inserted do in general not appear in the proof up to this point (although they often do in practice). It can be necessary to "invent" or synthesise completely new terms.

A general approach to overcome the problem, which has been developed in the area of automated theorem proving, are "free variables" or "metavariables". Metavariables are place-holders that can be introduced instead of actual expressions when applying rules. For an extensive account on metavariables in first-order logic see, for instance, [12]. While metavariables can in principle be introduced for all kinds of expressions, most commonly (and also in KeY) they are only used as place-holders for terms: whenever a taclet is applied that contains schema variables for terms that only occur in goal templates, instead of immediately choosing the instantiation of the schema variables, metavariables can be introduced.

At some point after introducing a metavariable, it often becomes obvious which term the metavariable should stand for, or it becomes necessary to choose a certain term in order to apply a rule. This is achieved by applying a *substitution* (⇒ Sect. 2.4) to the whole proof tree that replaces the metavariable with the chosen term. Because such a replacement is a destructive operation (substituting the wrong term can make it necessary to start over with parts of the proof), KeY follows the non-destructive approach that is described in [12, 4] and actually never applies substitutions. Instead of substitutions, *constraints* are stored that describe substitution candidates. Constraints are generated whenever the application of substitutions becomes necessary in order to apply rules and are attached to formulae and to proof goals. The actual application of the substitution can then be postponed until it is certain that the substitution is not harmful.

Not all terms can be substituted for metavariables. Because metavariables are considered as *rigid* symbols, in particular, it is not allowed to substitute non-rigid terms (like program variables) for metavariables. In practice, this seriously limits the usefulness of metavariables when doing proofs in dynamic logic and is an issue that belongs to the "Future Work" section.

Further examples for the usage of metavariables and constraints are given in [13].

Skolem Symbols and the "Occurs Check"

The concept of metavariables collides, to a certain degree, with rules that are supposed to introduce fresh symbols (like `allRight` in Sect. 1). The problematic situation is as follows: by applying substitutions to a proof tree, the sequents that rules are operating on can be modified *after* actually applying the rules. This means that we can no longer be sure that a symbol that does not turn up in sequents actually is fresh, because it could also be inserted at a later point through a substitution. In order to illustrate this phenomenon, we try to prove the (non-valid) formula $\exists x.\, \forall y.\, x \doteq y$ using a metavariable X:

$$\dfrac{\dfrac{\vdash \exists x.\, \forall y.\, x \doteq y, X \doteq c_{\mathrm{sk}}}{\vdash \exists x.\, \forall y.\, .x \doteq y, \forall y.\, X \doteq y}\ \mathsf{allRight}}{\vdash \exists x.\, \forall y.\, x \doteq y}\ \mathsf{exRight}$$

At this point, it becomes obvious that we would like to substitute c_{sk} for X:

$$\dfrac{\dfrac{\vdash \exists x.\, \forall y.\, x \doteq y, c_{\mathrm{sk}} \doteq c_{\mathrm{sk}}}{\vdash \exists x.\, \forall y.\, x \doteq y, \forall y.\, c_{\mathrm{sk}} \doteq y}\ \mathsf{allRight}}{\vdash \exists x.\, \forall y.\, x \doteq y}\ \mathsf{exRight}$$

The proof can now be closed by applying $\mathsf{eqClose}$ to the formula $c_{\mathrm{sk}} \doteq c_{\mathrm{sk}}$. Searching for the mistake, we see that the application of $\mathsf{allRight}$ becomes illegal after applying the substitution, because the constant c_{sk} that is introduced is no longer fresh.

There is a simple and standard solution to this inconsistency: when introducing symbols that are supposed to be fresh, critical metavariables have to be listed *as arguments* of the fresh symbols. A correct version of our proof attempt is:

$$\dfrac{\dfrac{\vdash \exists x.\, \forall y.\, x \doteq y, X \doteq c_{\mathrm{sk}}(X)}{\vdash \exists x.\, \forall y.\, x \doteq y, \forall y.\, X \doteq y}\ \mathsf{allRight}}{\vdash \exists x.\, \forall y.\, x \doteq y}\ \mathsf{exRight}$$

It is easy to see that no substitution can make the terms X and $c_{\mathrm{sk}}(X)$ syntactically equal, so that it is impossible to close the proof: the terms are *not unifiable*, because X occurs in the term that it is supposed to represent. This situation is known as a failing *occurs check*.

When writing taclets that introduce fresh symbols (using schema variables of kind `\skolemTerm`), it is currently necessary to specify the metavariables

that have to turn up as arguments of the symbol by hand and using the variable condition `\new(...,\dependingOn(...))` (⇒ Sect. 2.6). An example is the taclet `allRight` (⇒ Sect. 1). It is likely, however, that these dependencies will be computed automatically by KeY in the future.

4 Systematic Introduction of Taclets

This section introduces the syntax and semantics of the taclet language. The first pages are written in the style of a reference manual for the different taclet constructs and provide most of the information that is necessary for writing one's own taclets. Later, the meaning of taclets is defined in a more rigorous setting.

4.1 The Taclet Language

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

Taclets describe elementary goal expansion steps. In short, a taclet contains information about 1. when and to which parts of a sequent the taclet can be applied, and 2. in which way the proof is expanded or a proof goal is closed. This information is given by the different parts that make up the body of a taclet. Fig. 9 shows the syntax of the taclet parts, which are explained in more detail on the following pages.

Context Assumptions: What has to be present in a sequent

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

Context assumptions are—together with the `\find` part of a taclet—the means of expressing that a goal modification can only be performed if certain formulae are present in the goal. If a taclet contains an `\assumes` clause, then the taclet may only be applied if the given sequent is part of the goal that is supposed to be modified. Formulae specified as assumptions will not be consumed[8] by the application of the taclet, they will instead be kept and also be present in the modified goals.

Examples in Sect. 1: `close`, `mpLeft` (⇒ Fig. 4), `applyEq` (⇒ Fig. 5)

[8] It is possible, however, that a taclet is applied on one of its assumptions, i.e., that an assumption is also matched by the `\find` pattern of a taclet. In this situation a taclet application can modify or remove an assumption.

KeY syntax

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

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

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

⟨*stateCondition*⟩ ::= **\inSequentState** | **\sameUpdateLevel**

⟨*variableConditions*⟩ ::= **\varcond** **(** ⟨*variableConditionList*⟩ **)**
⟨*variableConditionList*⟩ ::= ⟨*variableCondition*⟩ (**,** ⟨*variableCondition*⟩)∗

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

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

KeY syntax

Fig. 9. The Taclet Syntax

Find Pattern: To which expressions a taclet can be applied

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

More specifically than just to a goal of a proof, taclets are usually applied to an occurrence of either a formula or a term within this goal. This occurrence is called the *focus* of the taclet application and is the only place in the goal where the taclet can modify an already existing formula (apart from which, only new formulae can be added to the goal). There are three different kinds of patterns a taclet can match on:

- A schematic sequent that contains exactly one formula: this either specifies that the taclet can be applied if the given formula is an element of the antecedent, or if it is an element of the succedent, with the formula being the focus of the application. It is allowed, however, that the occurrence of the formula is preceded by updates (see the section on "State Conditions").
- A formula: the focus of the application can be an arbitrary occurrence of the given formula (also as subformula) within a goal.
- A term: the focus of the application can be any occurrence of the given term within a goal.

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

Examples in Sect. 1: Virtually all taclets given there

State Conditions: Where a taclet can be applied

⟨*stateCondition*⟩ ::= **\inSequentState** | **\sameUpdateLevel**

In a modal logic like dynamic logic, a finer control over where the focus of a taclet application may be located is needed than is provided by the different kinds of `\find` patterns. For rewriting rules it is, for instance, often necessary to forbid taclet applications within the scope of modal operators (like program blocks) in order to ensure soundness. There are three different "modes" that a taclet can have:

- `\inSequentState`: the most restrictive mode, in which the focus of a taclet application must not be located within the scope of *any* modal operator (like programs or updates).
- `\sameUpdateLevel`: this mode is only allowed for rewriting taclets (i.e., there is a `\find` clause, and the pattern is not a sequent) and allows the application focus of a taclet to lie within the scope of updates, but not in the scope of other modal operators. The same updates have to occur in front of the application focus and the formulae referred to using `\assumes`, `\replacewith` and `\add`.

  Examples in Sect. 1: `applyEq`, `applyEqAR` (⇒ Fig. 5)

Table 7. Matrix of the different taclet modes and the different `\find` patterns. For each combination, it is shown 1. where the focus of the taclet application can be located, and 2. which updates consequently have to occur above the formulae that are matched or added by `\assumes`, `\add` or `\replacewith`.

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

– Default: the most liberal mode. For rewriting taclets, this means that the focus can occur arbitrarily deeply nested and in the scope of any modal operator. If the `\find` pattern of the taclet is a sequent, then the application focus may occur below updates, but not in the scope of any other operator.

State conditions also affect the formulae that are required or added by `\assumes`, `\add` or `\replacewith` clauses. Such formulae have to be preceded by the same updates as the focus of the taclet application (which also explains the keyword `\sameUpdateLevel`). The only exception are rewriting taclets in "default" mode, where formulae that are described by `\assumes` or `\add` must *not* be in the scope of updates, whereas there are no restrictions on the location of the focus. The relation between the positions of the different formulae is also shown in Table 7.

Variable Conditions: How schema variables may be instantiated

⟨*variableConditions*⟩ ::= **`\varcond`** `(` ⟨*variableConditionList*⟩ `)`
⟨*variableConditionList*⟩ ::= ⟨*variableCondition*⟩ (`,` ⟨*variableCondition*⟩)∗

Schema variable conditions have already been introduced in Sect. 2.6, together with the concrete syntax for such conditions that is used in KeY. A list of such conditions can be attached to each taclet to control how schema variables are allowed to be instantiated.

Examples in Sect. 1: `removeAll` (⇒ Fig. 5)

Goal Templates: The effect of the taclet application

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

If the application of a taclet on a certain goal and a certain focus is permitted and is carried out, the *goal templates* of the taclet describe in which way the goal is altered. Generally, the taclet application will first create a number of new proof goals (split the existing proof goal into a number of new goals) and then modify each of the goals according to one of the goal templates. A taclet without goal templates will close a proof goal. As shown above as well as in Sect. 1, in this case the keyword `\closegoal` is written instead of a list of goal templates in order to clarify this behaviour syntactically.

Goal templates are made up of three kinds of operations:

– `\replacewith`: if a taclet contains a `\find` clause, then the focus of the taclet application can be replaced with new formulae or terms. `\replacewith` has

to be used in accordance with the kind of the `\find` pattern: if the pattern is a sequent, then also the argument of the keyword `\replacewith` has to be a sequent, etc. In contrast to `\find` patterns, there is no restriction concerning the number of formulae that may turn up in a sequent being argument of `\replacewith`. It is possible to remove a formula from a sequent by replacing it with an empty sequent, or to replace it with multiple new formulae.
- `\add`: independently of the kind of the `\find` pattern, the taclet application can add new formulae to a goal.
- `\addrules`: a taclet can also create new taclets when being applied. We will ignore this feature for the time being and come back to it in Sect. 4.6.

Examples in Sect. 1: `applyEqAR` (⇒ Fig. 5), `saveLeft`, `hideLeft`

Apart from that, each of the new goals (or branches) can be given a name in order to improve readability of proof trees.

Rule Sets: How taclets are applied automatically

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

Each taclet can be declared to be element of a number of rule sets, which in turn are used by the *strategies* in KeY that are responsible for applying rules automatically. Rule sets are intended as an abstraction from the actual taclets and identify taclets that should be treated in the same way. Existing rule sets in KeY are, amongst many others, `alpha`[9] (for non-splitting elimination of propositional connectives), `beta` (splitting elimination of connectives) and `simplify` (simplification of expressions).

4.2 Well-Formedness Conditions on Taclets

Not all taclets that can be written using the syntax of Sect. 4.1 are meaningful or desirable descriptions of rules. We want to avoid, in particular, rules whose application could destroy well-formedness of formulae or sequents. In this section, we thus give a number of additional constraints on taclets that go beyond the pure taclet syntax. As a note up-front, all of the following issues could also be solved in different ways, possibly leading to a more flexible taclet language, but experience shows that the requirement to write taclets in a more explicit way reduces the risk of introducing bugs and unsound taclets.

In the KeY implementation, non-well-formed taclets are immediately rejected and cannot even be loaded.

[9] The names `alpha` and `beta` are common terminology in tableaux-style theorem provers.

Sequents do not Contain Free Variables

Following [13], we do not allow sequents of our proofs to contain free logical variables (in contrast to metavariables (⇒ Sect. 3), which are never bound). Unfortunately, this is a property that can easily be destroyed by incorrect taclets:

Taclet

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

Taclet

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

In order to handle taclets like `illegalTac1`, we simply forbid taclets containing `\replacewith` or `\add` clauses with unbound schema variables `va`:

Definition 12. *An occurrence of a schema variable* `va` *of kind* `\variables` *in a schematic expression is called* bound *if it is in the scope of a quantifier* ∀`va`., ∃`va`. *or a substitution* {*subst* `sv`; t}*, and if it is not itself part of a binder (like* ∀`va`.*).*

Requirement 1 (Variables are bound). *All occurrences of a schema variable of kind* `\variables` *in* `\find`, `\assumes`, `\replacewith` *or* `\add` *clauses are either part of a binder or are bound.*

Note 6. It would be safe to allow schema variables of kind `\variables` to occur free in `\find` or `\assumes`. This would only lead to useless taclets that are never applicable (provided that sequents are not already ill-formed and contain free logical variables).

It is less obvious how taclet `illegalTac2` should be taken care of. According to Sect. 2, the schema variable `phi` can stand for arbitrary formulae containing arbitrary free variables, but as the example shows this is too liberal. Whenever a variable x occurs free in a formula ι(`phi`), the formula also has to be in the scope of a binder of x. The binder could either occur explicitly in the taclet—like a quantifier ∀`va`.—or for a rewriting taclet it could be part of the context in which the taclet is applied. An example for the latter possibility is shown in Sect. 2.4.

We will here go for a rigorous solution of the problem and require the author of `illegalTac2` to state his or her intention more clearly. The first half of the solution is given in the following definition, where we require that the variables that are explicitly bound in the taclet above occurrences of `phi` are consistent.

The second part is Def. 14, describing which logical variables we allow to occur free in a formula $\iota(\texttt{phi})$.

Requirement 2 (Unique Context Variables). *Suppose that t is a taclet, that* `sv` *is a schema variable of kind* `\term` *or* `\formula`*, and that* `va` *is a schema variable of kind* `\variables`*. If an occurrence of* `sv` *in* `\find`*,* `\assumes`*,* `\replacewith` *or* `\add` *clauses of t is in the scope of a binder of* `va` *(which could be* $\forall$`va.`*,* $\exists$`va.` *or* $\{$`\subst sv; u`$\}$*), then*

- all *occurrences of* `sv` *in t are in the scope of a binder of* `va`*, or*
- *t has a variable condition* `\notFreeIn(va, sv)`*.*

The variable `va` *is called a* context variable *of* `sv` *in t. More formally, the set of context variables of* `sv` *in t is defined as*

$$\begin{aligned} \Pi_t(\texttt{sv}) = \quad & \{\texttt{va} \mid \texttt{va} \textit{ is of kind } \texttt{\textbackslash variables}, \texttt{sv} \textit{ is in the scope of } \texttt{va}\} \\ & \setminus \{\texttt{va} \mid t \textit{ has variable condition } \texttt{\textbackslash notFreeIn(va, sv)}\} \end{aligned}$$

Correct versions of the taclet shown above are thus:

Taclet

```
legalTac2a { \find(==> \forall va; phi)
             \replacewith(==> {\subst va; t} phi) };
legalTac2b { \find(==> \forall va; phi)
             \varcond(\notFreeIn(va, phi))
             \replacewith(==> phi) };
```

Taclet

The context of `phi` in these two taclets, i.e., the sets of variables that are bound for all occurrences of `phi` in the taclets, is $\Pi_{\texttt{legalTac2a}}(\texttt{phi}) = \{\texttt{va}\}$ and $\Pi_{\texttt{legalTac2b}}(\texttt{phi}) = \emptyset$.

Note 7. The KeY implementation contains a further well-formedness condition: schema variables of kind `\variables` are allowed to be bound at most once in the `\find` and `\assumes` clauses of a taclet (together). This is not a severe restriction, because it is always possible to apply bound renaming for ensuring that variables are only bound in one place, without changing the intended meaning of a taclet. For the representation of taclets in this chapter, however, it is not necessary to enforce unique binding of variables.

4.3 Implicit Bound Renaming and Avoidance of Collisions

As already seen in Sect. 2.4 about substitutions, the actual identity of bound variables is not important for the meaning of a formula. If a variable $z \notin fv(\phi)$ does not occur free in ϕ, then formulae $\forall x.\, \phi$ and $\forall z.\, [x/z](\phi)$ will be equivalent.

Although it is not strictly necessary for achieving completeness,[10] from a practical point of view it is desirable that the applicability of taclets does not depend on which variables are bound in formulae. We would like to treat sequents like

$$\forall x.\, p(x) \vdash \forall x.\, p(x)$$
$$\forall x.\, p(x) \vdash \forall y.\, p(y)$$

in the same way, in particular a taclet like `close` (⇒ Sect. 1) should be applicable to one of the sequents if and only if it is applicable to the other sequent. For the second sequent, this would mean that the schema variable `phi` represents two different formulae, which will obviously not work without further measures.

The most common (theoretic) standpoint is to allow implicit renaming steps whenever they are necessary for applying rules. We follow this approach in the scope of this chapter, and in the definitions on the following pages we will only formulate conditions on (bound) variables that possibly have to be established by implicit renaming. In this setting, an application of `close` to the second sequent would be

$$\dfrac{\dfrac{}{\forall y.\, p(y) \vdash \forall y.\, p(y)}\ \texttt{close}}{\forall x.\, p(x) \vdash \forall y.\, p(y)}\ (\mathsf{rename})$$

where the first step (rename) is performed implicitly for preparing the sequent for the actual taclet application. Likewise, in both formulae the variables x (or any other variable) could have been introduced. Renaming steps are often not shown at all in proofs.

Note 8. KeY applies bound renaming, whenever it becomes necessary, as part of its taclet application mechanism. As renaming steps are not shown in the proof tree, the user will in most cases not even notice that a renaming has occurred.

Collisions

We have already met the problem of *collisions* in Sect. 2.4. A similar phenomenon occurs when applying and evaluating taclets, also here it is possible that the place where a variable is bound changes unintentionally. The reason is that distinct schema variables of kind `\variables` can happen to be instantiated with *the same* logical variable. If an "artificial", but in principle reasonable, taclet for eliminating universal quantifiers in the antecedent

Taclet

```
allExLeft {
     \find (\forall x; \exists y; phi ==>)
     \varcond (\notFreeIn(y, t))
     \add (\exists y; {\subst x; t} phi ==>)  };
```

Taclet

[10] The calculus for first-order logic in [13] is complete also without rules for bound renaming or comparing formulae modulo bound renaming. This shows that it is, in principle, sufficient to provide rules for *eliminating* quantifiers.

was applied naively, we would have to consider the taclet as unsound. The taclet commutes binders, so that the location where variables are bound can change:

$$\dfrac{\dfrac{\forall x.\, \exists x.\, x \doteq 0,\ \exists x.\, 1 \doteq 0 \vdash}{\forall x.\, \exists x.\, x \doteq 0,\ \exists x.\, [x/1](x \doteq 0) \vdash}}{\forall x.\, \exists x.\, x \doteq 0 \vdash}$$

Here, the lower sequent is not valid, but the upper one is, which means that the application of the taclet has to be considered unsound. In order to make the development of sound taclets easier (feasible), we will forbid such taclet applications:

Definition 13. *Suppose that t is a taclet and that ι is an instantiation of the variables of t. Then ι* has distinct bound variables *concerning t if all logical variables $\iota($*`va`*$)$ represented by schema variables* `va` *of kind* `\variables` *in t are distinct.*

As illustrated in the first part of this section, we assume that the conditions of the previous definition are implicitly established by suitable variable renaming when applying a taclet. In rather special cases, it can also be necessary to duplicate formulae in order to establish distinctness. As an example, we can imagine a taclet that works with terms representing surjective functions:

Taclet

```
surjectivity {
     \assumes (\forall x; \exists y; x = t ==>)
     \find (\exists z; phi) \sameUpdateLevel
     \varcond (\notFreeIn(x, t), \notFreeIn(y, phi))
     \replacewith (\exists y; {\subst z; t} phi)
};
```

Taclet

Provided that a term `t`—in which a variable `y` can occur free—describes a surjective mapping, we can use it to rewrite quantified formulae:

$$\dfrac{\forall a.\, \exists b.\, a \doteq b + 1 \vdash \exists b.\, b + 1 - 1 \doteq 3}{\forall a.\, \exists b.\, a \doteq b + 1 \vdash \exists c.\, c - 1 \doteq 3}\ \texttt{surjectivity}$$

In a more pathological application, however, the taclet can be used to modify the quantified formula of the antecedent itself:

$$\dfrac{\forall a.\, \exists b.\, a \doteq b + 1 + 1 \vdash}{\forall a.\, \exists b.\, a \doteq b + 1 \vdash}\ \texttt{surjectivity}$$

Strictly speaking, this transformation is only possible if the formula is first duplicated and the variable b is renamed to a new variable b'. The variable condition `\notFreeIn(y, phi)` would otherwise be violated: `phi` becomes instantiated

with $a \doteq b + 1$ and `y` with the variable b, which contradicts the variable condition. Hence, a detailed proof tree showing the taclet application looks as follows:

$$\dfrac{\dfrac{\dfrac{\forall a.\, \exists b.\, a \doteq b + 1 + 1 \vdash}{\forall a.\, \exists b.\, a \doteq b + 1,\ \forall a.\, \exists b.\, a \doteq b + 1 + 1 \vdash}\ \textsf{(weak.)}}{\forall a.\, \exists b.\, a \doteq b + 1,\ \forall a.\, \exists b'.\, a \doteq b' + 1 \vdash}\ \texttt{surj.}}{\forall a.\, \exists b.\, a \doteq b + 1 \vdash}\ \textsf{(rename)}$$

In the KeY implementation, the steps (rename) and (weaken) would be carried out automatically and not be shown in the proof tree. The example illustrates that it can—in seldom cases—be necessary to duplicate formulae and apply renaming before the application of a taclet is possible.

Note 9. The condition of Def. 13 is sufficient for preventing collisions, but is usually more defensive than necessary. In the KeY implementation, more precise (and complicated) conditions are used that often avoid variable renaming or duplicating formulae.

4.4 Applicability of Taclets

This section describes when the application of a taclet is possible. An informal account of this was already given in Sect. 4.1, where the different clauses of a taclet are introduced, and is now accompanied with a more rigorous treatment. In order to apply a taclet on a proof goal, several parameters have to be provided:

- If the taclet contains schema variables of generic types (⇒ Sect. 2.7), then these types first have to be mapped to concrete types.
- If the taclet contains schema variables, then one has to give an instantiation (⇒ Sect. 2.3) that describes how to replace the variables with concrete expressions.
- If the taclet contains a `\find` clause (⇒ Sect. 4.1), then it is necessary to select a focus of the taclet application within the goal in question.

Not all values that can be chosen for these unknowns are meaningful and should be admitted. It is obvious that taclet application should not be allowed if, e.g., variable conditions are violated, but there are a number of further and more subtle requirements.

Free Variables in Instantiations of Schema Variables

The examples in the previous sections show that instantiations of schema variables for terms or formulae should be allowed to contain certain logical variables free. At the same time, it has to be ensured that no free variables are introduced in sequents. In taclet `legalTac2a` (⇒ Sect. 4.2), for instance, it should be possible to instantiate variable `va` with x and `phi` with $p(x)$ (where x occurs free), so that the taclet matches on a formula $\forall x.\, p(x)$. The tool for deciding about the free variables that are permitted will be the *context variables* of a schema

variable (Prop 2), which are exactly those variables that are guaranteed to be bound (in the taclet) for each occurrence of a schema variable.

For certain rewriting taclets, it is desirable to allow further free variables that are *not* context variables of a schema variable. An example is the taclet `zeroRight` (⇒ Sect. 1), which we also would like to apply to a formula like $\forall x.\, p(x+0)$. The variable `intTerm` does not have any context variables, however, as there are no variables bound at all in the taclet. Situations like this are taken into account as well by the next definition.

Definition 14. *Suppose that t is a taclet, that ι is an instantiation of the variables of t and that focus*[11] *is a potential application focus of t (we choose focus $= \bot$ if t does not have a* `\find` *clause). We say that ι* respects variable contexts *concerning focus if, for every schema variable* `sv` *of t of kind* `\term` *or* `\formula` *and all free variables $x \in fv(\iota(\texttt{sv}))$,*

- *there is a schema variable* $\texttt{va} \in \Pi_t(\texttt{sv})$ *with* $\iota(\texttt{va}) = x$*, or*
- *t contains at most one* `\replacewith` *clause,* `sv` *turns up only in* `\find`*,* `\replacewith` *or* `\varcond` *clauses of t, and x is bound above focus.*

The second item makes it possible to apply taclet `zeroRight` to a formula $\forall x.\, p(x+0)$. According to Table 7, it only applies to rewriting taclets, because for other taclets there will never be any variables bound above *focus*.

Note 10. The requirement of the second item—that there is not more than one `\replacewith` clause—is added because quantifiers or other binders do in general not distribute through conjunctions. It would hardly be possible to formulate sound rewriting taclets with more than one `\replacewith` clause if free variables were allowed to turn up. An example is the (artificial, but not unreasonable) taclet performing a case distinction on a term of type *boolean*

Taclet

```
splitBool { \find (b) \replacewith (TRUE);
                      \replacewith (FALSE) };
```

Taclet

which could be used in the following unsound way:

$$\frac{\vdash \exists x : boolean.\, TRUE \not\doteq x \qquad \vdash \exists x : boolean.\, FALSE \not\doteq x}{\vdash \exists x : boolean.\, x \not\doteq x}$$

Permitted Taclet Applications

We will now give a complete definition of when we consider a taclet, given a sequent and all necessary parameters, as applicable:

[11] We write *focus* for an occurrence of the formula or term *focus*, i.e., *focus* describes not only an expression but also a location within a sequent.

Definition 15 (Matching Instantiation). *Suppose that t is a taclet over a generic type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \mathcal{T}_g, \sqsubseteq, \mathcal{R}_g)$ and a set SV of schema variables. A* matching instantiation of t *is a tuple $(\iota_t, \iota, \mathcal{U}, \Gamma \vdash \Delta, \underline{focus})$ consisting of*

- *a type instantiation ι_t,*
- *a complete instantiation ι of the schema variables of t (apart from those variables that only occur within* `\addrules` *clauses),*
- *an update $\mathcal{U}$ describing the context of the taclet application ($\mathcal{U}$ can be empty),*
- *a sequent $\Gamma \vdash \Delta$ to which the taclet is supposed to be applied, and*
- *an application focus $\underline{focus}$ within $\Gamma \vdash \Delta$ that is supposed to be modified (we write $\underline{focus} = \bot$ if t does not have a* `\find` *clause)*

that satisfies the following conditions:

1. *the pair (ι_t, ι) is an admissible instantiation of SV under generic types ($\Rightarrow$ Def. 11),*
2. *ι satisfies all variable conditions of taclet t (referring to Table 6),*
3. *ι respects the variable context of t concerning $\underline{focus}$ ($\Rightarrow$ Def. 14),*
4. *ι has distinct bound variables concerning t ($\Rightarrow$ Def. 13),*
5. *if t has a* `\find` *clause, then the position of $\underline{focus}$ is consistent with the state conditions of t (Table 7),*
6. *$\mathcal{U}$ is derived from $\underline{focus}$ according to the middle part "Which updates have to occur above* `\assumes` *and* `\add` *formulae" of Table 7 (for $\underline{focus} = \bot$ and the fields "forbidden combination" we choose the empty update* `skip`*),*
7. *for each formula ϕ of an* `\assumes` *clause of t, $\Gamma \vdash \Delta$ contains a corresponding formula $\mathcal{U}\iota(\phi)$ (on the correct side),*
8. *if t has a clause* `\find(`f`)`*, where f is a formula or a term, then $\iota(f) = \underline{focus}$ (the* `\find` *pattern has to match the focus of the application),*
9. *if t has a clause* `\find(`f`)`*, where f is a sequent containing a single formula ϕ, then $\iota(\phi) = \underline{focus}$ and the formulae ϕ and $\underline{focus}$ occur on the same sequent side (both antecedent or both succedent).*

Example 9. We show how this definition applies to taclet `instAll` ($\Rightarrow$ Fig. 10), which is a variant of `allLeft` and allows to select the term that is supposed to be substituted as focus. We can apply the taclet to the sequent

$$\Gamma \vdash \Delta \quad = \quad \forall o.\, f(o) \doteq o.a,\ f(self) \doteq 1 \vdash \{i := 2\}(\underline{self}.a \doteq 1)$$

where the application focus is underlined, the constant *self* and the logical variable o have type A and $f : A \to$ `integer` is a function. The remaining components of the matching instantiation are:

- the type instantiation $\iota_t = \{G \mapsto A\}$,
- the instantiation $\iota = \{\texttt{phi} \mapsto f(o) \doteq o.a,\ \texttt{x} \mapsto o,\ \texttt{s} \mapsto self\}$,
- the (effect-less) update $\mathcal{U} =$ `skip`.

That the instantiation is indeed matching can be observed as follows:

KeY

```
\sorts {
  \generic G;
}
\schemaVariables {
  \formula phi;      \variables G x;      \term G s;
}
\rules {
  instAll { \assumes (\forall x; phi ==>) \find (s)
            \add ({\subst x; s}phi ==>) };
}
```

KeY

Fig. 10. The Taclet described in Example 9.

1. We have $\iota_t(\texttt{\textbackslash variables}\ G) = \texttt{\textbackslash variables}\ A$, $\iota_t(\texttt{\textbackslash term}\ G) = \texttt{\textbackslash term}\ A$. Because of the types of *self* and o, the pair (ι_t, ι) is an admissible instantiation.
2. There are no variable conditions.
3. The variable contexts of `instAll` are the sets $\Pi_{\texttt{instAll}}(\texttt{phi}) = \{\texttt{x}\}$ and $\Pi_{\texttt{instAll}}(\texttt{s}) = \emptyset$, they are respected by ι as $fv(\iota(\texttt{phi})) = \{o\} = \{\iota(\texttt{x})\}$ and $\iota(\texttt{s})$ does not contain variables.
4. There is only one schema variable of kind `\variables`. Hence, ι has distinct bound variables.
5. `instAll` has no explicit state conditions, and thus all operators are allowed above the focus $\underline{self}$.
6. According to Table 7, $\mathcal{U}$ has to be the empty update `skip`. Note that this is the case even though the application focus is in the scope of an update.
7. The `\assumes` clause contains only one formula, which is correctly mapped to one of the formulae of Γ: $\iota(\forall \texttt{x}.\, \texttt{phi}) = \forall o.\, f(o) \doteq o.a$
8. The `\find` expression is correctly mapped to the term of the application focus: $\iota(\texttt{s}) = \mathit{self}$
9. (Does not apply)

4.5 The Effect of a Taclet

Applying a taclet to a goal and a focus will carry out the modification steps that are described by the goal templates of the taclet. Each goal template can alter the focus the taclet is applied to (`\replacewith`), add further formulae to a goal (`\add`) and make further taclets available (`\addrules`). In this section, we concentrate on the first two kinds of effects and postpone a discussion of the latter kind until Sect. 4.6.

Definition 16 (Applying a Goal Template). *Suppose that a matching instantiation $(\iota_t, \iota, \mathcal{U}, \Gamma \vdash \Delta, \underline{focus})$ of a taclet t is given. One goal template is applied on $\Gamma \vdash \Delta$ by performing the following steps (in the given order):*

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

The complete application of a taclet involves duplicating a proof goal and applying each of its goal templates. In case of taclets that do not have any goal templates, this actually closes the proof goal.

Definition 17 (Applying a Taclet). *Suppose that a matching instantiation* $(\iota_t, \iota, \mathcal{U}, \Gamma \vdash \Delta, \underline{focus})$ *of a taclet* t *is given, where* $\Gamma \vdash \Delta$ *is the sequent of one proof goal* g*. Carrying out the application of* t *means performing the following steps (in the given order):*

1. n *new proof goals with sequent* $\Gamma \vdash \Delta$ *are created as children of* g*, where* n *is the number of goal templates of* t*. For* $n = 0$ *the goal* g *is closed.*
2. *Each of the goal templates of* t *is applied to one of the new goals, given the matching instantiation* $(\iota_t, \iota, \mathcal{U}, \Gamma \vdash \Delta, \underline{focus})$*.*

Example 10. We continue Example 9 and apply `instAll` with the matching instantiation shown there. The taclet `instAll` has only a single goal template, so the first step is to duplicate the initial sequent:

$$\frac{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1 \;\vdash\; \{i := 2\}(self.a \doteq 1)}{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1 \;\vdash\; \{i := 2\}(self.a \doteq 1)}$$

Applying the goal template here only means to carry out the `\add` clause. The formula that is to be added is (the update can be left out immediately)

$$\mathcal{U}\iota(\{\backslash\text{subst } \texttt{x};\ \texttt{s}\}\texttt{phi}) \quad = \quad \{\texttt{skip}\}([o/self](f(o) \doteq o.a)) \quad = \quad f(self) \doteq self.a$$

Finally, the rule application yields

$$\frac{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1, \quad f(self) \doteq self.a \;\vdash\; \{i := 2\}(self.a \doteq 1)}{\forall o.\, f(o) \doteq o.a, \quad f(self) \doteq 1 \;\vdash\; \{i := 2\}(self.a \doteq 1)}$$

4.6 Taclets in Context: Taclet-Based Proofs

So far, we have introduced and defined the meaning of taclets as modification steps that can be applied to a proof tree. Taclets can, however, also modify the rule base that is used to construct a proof. Probably the best example for this feature is taclet `applyEqAR` (⇒ Sect. 1) for rewriting terms in the presence

of an equation in an antecedent. Applying the taclet to an equation that can be matched by `t` $\doteq$ `t2` results in a new taclet `rewrWithEq` that replaces the term matched by `t` with the term matched by `t2`. It is clear, however, that the taclet `rewrWithEq` must not be added to the rule base "globally", as it is only correct for those sequents that actually contain an equation $f(a) \doteq b$. `\addrules` is only meaningful if we have a notion of "local" rules that only exist in certain parts of a proof tree, and that are not available elsewhere. To realise such a notion, taclets will get a character that is similar to the formulae of a sequent: to each sequent, a set of taclets is attached that are available for application. If a proof is expanded by adding children to a parent goal, then these goals will inherit all rules from the parent goal, but will possibly also get further rules that were not present in the parent goal (like `rewrWithEq`).

Partially Instantiated Taclets

What is attached to sequents are not only the actual taclets, but also further information that is necessary to restrict the applicability of taclets in the right way. What is actually added when applying `applyEqAR` to the equation $f(a) \doteq b$ is the taclet `rewrWithEq`

Taclet

```
rewrWithEq { \find (t) \sameUpdateLevel \replacewith (t2) }
```

Taclet

together with the following components:

- the type instantiation $\iota_t = \{G \mapsto A\}$ (where A is the type of $f(a)$),
- the instantiation $\iota = \{\iota(\texttt{t}) \mapsto f(a),\quad \iota(\texttt{t2}) \mapsto b\}$, and
- the update $\mathcal{U} =$ `skip`.

The two instantiation functions have to be considered as partial in this setting, because an inner taclet like `rewrWithEq` can contain schema variables or generic types that are not part of the parent taclet and, thus, are not yet determined.

Definition 18. *A* partially instantiated taclet *is a tuple* $(t, \iota_t, \iota, \mathcal{U})$ *consisting of*

- *a taclet* t,
- *a (partial) type instantiation* ι_t,
- *a (partial) schema variable instantiation* ι, *and*
- *an update* $\mathcal{U}$ *describing the context of the taclet application (*$\mathcal{U}$ *can be empty or* $\bot$*).*

When applying a partially instantiated taclet, the information already given has to be extended so that the application is possible.

Definition 19. *Suppose that* $(t, \iota_t', \iota', \mathcal{U}')$ *is a partially instantiated taclet. A* matching instantiation *of* $(t, \iota_t', \iota', \mathcal{U}')$ *is a tuple* $(\iota_t, \iota, \mathcal{U}, \Gamma \vdash \Delta, \underline{focus})$ *such that*

- $(\iota_t, \iota, \mathcal{U}, \Gamma \vdash \Delta, \underline{focus})$ *is a matching instantiation of* t *(⇒ Def. 15),*
- ι_t *is an extension of* ι_t' *(as function),*
- ι *is an extension of* ι' *(as function),*
- *if* $\mathcal{U}' \neq \bot$ *then* $\mathcal{U} = \mathcal{U}'$.

Taclet-Based Proofs

In contrast to a proof tree in an ordinary sequent calculus (see, for instance, [12]), to each node of a taclet-based proof tree a set of partially instantiated taclets is attached. The root of the tree is given the base set of rules, which will be partially instantiated taclets $(t, \bot, \bot, \bot)$, i.e., the instantiation mappings are completely undefined, and the update context of taclet applications is not yet determined. During the construction of the proof tree, further taclets can be added to proof nodes below the root using the `\addrules` clause.

Taking this into account, we can extend the definitions of the effect of taclets in the previous section.

Definition 20 (Continuation of Def. 16).

4. If the goal template has a clause `\addrules(`*rules*`)`*, then for each taclet r in rules the partially instantiated taclet $(r, \iota'_t, \iota', \mathcal{U})$ is added, where*
   - *ι' is the restriction of ι to the schema variables of r,*
   - *ι'_t is the restriction of the mapping ι_t to the types that occur within the kinds k of schema variables* `sv` *with $\iota($`sv`$) \neq \bot$.*

Definition 21 (Continuation of Def. 17).

1b. Each of the new proof goals is given the same set of partially instantiated taclets as the parent goal.

5 Reasoning about the Soundness of Taclets

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

For treating the correctness of taclets in a more systematic way, we would rather like to have some *calculus* for reasoning about soundness of taclets. This will be provided in this section for some of the features of taclets.[12] Note, that the following two translation steps correspond to the two main ingredients of taclets in the end of Sect. 1 (page 36).

- We describe a translation of taclets into formulae (the *meaning formulae* of taclets), such that a taclet is sound if the formula is valid. This translation captures the semantics of the different clauses that a taclet can consist of. Meaning formulae do, however, still contain schema variables, which means that for proving their validity methods like induction over terms or programs are necessary.

[12] The issue of metavariables (⇒ Sect. 3), for instance, will not be taken into account on the next pages.

– A second transformation handles the elimination of schema variables in meaning formulae, which is achieved by replacing schema variables with Skolem terms or formulae. The result is a formula of first order logic or dynamic logic (depending on the expressions that turned up in the taclet), such that the original formula is valid if the derived formula is valid. This step is only possible for certain kinds of schema variables; handling schema variables for program entities, in particular, can be difficult or impossible [14]. Depending on the kind of the schema variable, it can happen that only an incomplete transformation is possible, in the sense that the resulting formula can be invalid although the meaning formula actually is valid and the taclet is sound.

The two steps can be employed in different settings:

– The first step can be carried out, and one can reason about the resulting formula using an appropriate proof assistant, e.g. based on higher-order logic. For taclets that contain JAVA CARD programs, this will usually require to have a formalisation of the JAVA CARD semantics for the chosen proof assistant. In this context, some of the assignment rules for JAVA CARD [13] have been proven correct by [15] using the Isabelle/HOL proof assistant [8] and the Bali formalisation of JAVA [16]. [17] follow a similar strategy and prove the correctness of certain rules for the symbolic execution of JAVA referring to an existing JAVA semantics in rewriting logic [18].
– Both steps can be carried out, which opens up for a wider spectrum of provers or proof assistants that the resulting formulae can be tackled with. The formulae can in particular be treated by a prover for dynamic logic itself, such as KeY. This is applicable for *lemma* rules, i.e., for taclets which can be proven sound referring to other—more basic—taclets. The complete translation from taclets to formulae of dynamic logic can automatically be performed by KeY and makes it possible to write and use lemmas whenever this is useful, see [14].

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

5.1 Soundness in Sequent Calculi

In the whole section we write $(\Gamma \vdash \Delta)^* := \bigwedge \Gamma \rightarrow \bigvee \Delta$ for the formula that expresses the meaning of the sequent $\Gamma \vdash \Delta$. This formula is, in particular:

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

By the validity of a sequent we consequently mean the validity of the disjunction $(\Gamma \vdash \Delta)^*$.

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

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

where we assume that duplicate formulae are implicitly removed.

Definition 22 (Soundness). *A sequent calculus C is* sound *if only valid sequents are derivable in C, i.e., if the root $\Gamma \vdash \Delta$ of a closed proof tree is valid.*

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

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

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

Lemma 1. *Suppose that for each rule $R \in C$ of a calculus C and all tuples $(\langle P_1, \ldots, P_k\rangle, Q) \in R$ the following implication holds:*

$$\text{if } P_1, \ldots, P_k \text{ are valid, then } Q \text{ is valid.} \tag{1}$$

Then the calculus C is sound.

If condition (1) holds for all tuples $(\langle P_1, \ldots, P_k\rangle, Q) \in R$ of a rule R, then this rule is also called *sound*.

5.2 A Basic Version of Meaning Formulae

In our case, the rules of a calculus C are defined through taclets t over a set SV of schema variables, and within the next paragraphs we discuss how Lem. 1 can be applied considering such a rule. For a start, we consider a taclet whose `\find` pattern is a sequent and that has the following basic shape:

Taclet

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

Taclet

Using text-book notation for rules in sequent calculi (as in Fig. 3), the taclet describes the rule

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

In order to apply Lem. 1, it is then necessary to show implication (1) for all possible applications of the rule, i.e., essentially for all possible ways the schema variables that now turn up in the sequents can be instantiated. If ι is such a possible instantiation ($\Rightarrow$ Def. 15), and if $\Gamma \vdash \Delta$ is an arbitrary sequent, then

$$\begin{array}{rcll} P_i & = & \iota(\mathtt{rw}i \cup \mathtt{add}i \cup \mathtt{assum}) \cup (\Gamma \vdash \Delta) & (i = 1, \ldots, k), \\ Q & = & \iota(\mathtt{findSeq} \cup \mathtt{assum}) \cup (\Gamma \vdash \Delta) & \end{array} \tag{2}$$

Implication (1)—which is a *global* soundness criterion—can be replaced with a *local* implication:

$$\big(P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^*\big) \text{ is valid.} \tag{3}$$

Inserting the sequents (2) extracted from taclet `t1` into (3) leads to a formula whose validity is sufficient for implication (1):

$$\begin{array}{rcl} P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^* & = & \bigwedge_{i=1}^{k} \big(\iota(\mathtt{rw}i \cup \mathtt{add}i \cup \mathtt{assum}) \cup (\Gamma \vdash \Delta)\big)^* \\ & & \rightarrow \big(\iota(\mathtt{findSeq} \cup \mathtt{assum}) \cup (\Gamma \vdash \Delta)\big)^* \end{array} \tag{4}$$

In order to simplify the right side of Eq. (4), we can now make use of the fact that ι distributes through all propositional connectives ($\rightarrow$, $\wedge$, $\vee$, etc.) and also through the union of sequents. Furthermore, there is a simple law describing the relation between * and the union of sequents:

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

Thus, the formulae of Eq. (4) are equivalent to

$$\iota\Big(\bigwedge_{i=1}^{k} \big(\mathtt{rw}i \cup \mathtt{add}i \cup \mathtt{assum} \cup (\Gamma \vdash \Delta)\big)^* \rightarrow \big(\mathtt{findSeq} \cup \mathtt{assum} \cup (\Gamma \vdash \Delta)\big)^*\Big)$$

and can then be simplified to

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

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

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

for all possible instantiations ι. We will call the formula

$$M(\texttt{t1}) \;=\; \bigwedge_{i=1}^{k} \left(\texttt{rw}i^* \vee \texttt{add}i^*\right) \rightarrow \left(\texttt{findSeq}^* \vee \texttt{assum}^*\right) \tag{5}$$

the *meaning formula* of `t1`. From the construction of $M(\texttt{t1})$, it is clear that if $M(\texttt{t1})$ is valid whatever expressions we replace its schema variables with, then the taclet `t1` will be sound.

Meaning Formulae for Taclets that do not contain all clauses

We can easily adapt Eq. (5) if some of the clauses of `t1` are missing in a taclet:

- If the `\find` clause is missing: in this case, `findSeq` can simply be considered as the empty sequent, which means that we can set $\texttt{findSeq}^* = \text{false}$ in Eq. (5).
- If `\assumes` or `\add` clauses are missing: again we can assume that the respective sequents are empty and set

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

- If a clause `\replacewith(rwi)` is not present: then we can normalise by setting $\texttt{rw}i = \texttt{findSeq}$, which means that the taclet will replace the focus of the application with itself. If both `\replacewith` and `\find` are missing, we can simply set $\texttt{rw}i^* = \text{false}$.

Example 11. We consider the taclet `impRight` (⇒ Fig. 4) from Sect. 1 that eliminates implications within the succedent. The taclet represents the rule schema

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

and the meaning formula is the logically valid formula

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

5.3 Meaning Formulae for Rewriting Taclets

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

Taclet

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

Taclet

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

Taclet

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

Taclet

We create a taclet that adds equations `findTerm=rw1`, ..., `findTerm=rwk` to the antecedent. Using taclet `t2b` and a general rule for applying equations in the antecedent, the effect of `t2` can be simulated.[13] On the other hand, also taclet `t2b` can be simulated using `t2` and standard rules (cut, reflexivity of equality), which means that it suffices to consider the soundness of `t2b`. Eq. (5) and some propositional simplifications then directly give us the meaning formula

$$M(\texttt{t2b}) \;\equiv\; M(\texttt{t2}) \;=\; \bigwedge_{i=1}^{k} \left(\texttt{findTerm} \doteq \texttt{rw}i \rightarrow \texttt{add}i^*\right) \rightarrow \texttt{assum}^* \qquad (6)$$

In the same way, rewriting taclets for formulae can be treated, if equations are replaced with equivalences:

$$\bigwedge_{i=1}^{k} \left((\texttt{findFor} \leftrightarrow \texttt{rw}i) \rightarrow \texttt{add}i^*\right) \rightarrow \texttt{assum}^* \qquad (7)$$

5.4 Meaning Formulae in the Presence of State Conditions

Taclets that do *not* contain the statement `\inSequentState` (i.e., unlike all taclets whose soundness we have tackled so far) require a bit more care when deriving meaning formulae. As introduced in Sect. 4.1, there are two further modes that taclets can have, `\sameUpdateLevel` and the "default" mode without any flags. From the soundness point of view, it is meaningful to consider the following two categories of taclets:

- Taclets with mode `\sameUpdateLevel` and non-rewriting taclets with default mode: in contrast to taclets with mode `\inSequentState`, such taclets can also be applied in the scope of updates (see Table 7 on page 59). It is ensured that all parts of the taclets work in the same update context, i.e., the same updates will occur above the taclet application focus, above assumptions of the taclet (`\assumes`) and above expressions that are modified or added by the clauses `\replacewith` and `\add`.

[13] Strictly speaking, this transformation only works if `findTerm` and `rw`i are not instantiated to terms that contain free variables from the application context, as it is allowed in the second item of Def. 14. We can imagine to implicitly add universal quantifiers for such variables.

– Rewriting taclets with default mode: this is the most liberal case, in which the application focus can be in the scope of arbitrary modal operators. This means that the application focus can in particular be located in a different state from assumptions of the taclet (`\assumes`) or formulae that are added by an `\add` clause.

The following paragraphs sketch how meaning formulae for taclets of these two kinds can be created.

Taclets with `\sameUpdateLevel`, Non-Rewriting Taclets with Default Mode

We only consider a taclet in default mode in which `\find` pattern is a sequent, but the same reasoning applies to rewriting taclets with the statement `\sameUpdateLevel`.

Taclet

```
t3 { \assumes(assum) \find(findSeq)
     \replacewith(rw1) \add(add1);
     ...
     \replacewith(rwk) \add(addk) };
```

Taclet

In text-book notation, the rule implemented by the taclet will consequently look as follows:

$$\frac{\begin{array}{c}\mathcal{U}\,\mathtt{rw1} \cup \mathcal{U}\,\mathtt{add1} \cup \mathcal{U}\,\mathtt{assum} \cup (\Gamma \vdash \Delta) \\ \cdots \\ \mathcal{U}\,\mathtt{rwk} \cup \mathcal{U}\,\mathtt{addk} \cup \mathcal{U}\,\mathtt{assum} \cup (\Gamma \vdash \Delta)\end{array}}{\mathcal{U}\,\mathtt{findSeq} \cup \mathcal{U}\,\mathtt{assum} \cup (\Gamma \vdash \Delta)}$$

We write $\mathcal{U}\,(\Gamma \vdash \Delta)$ for denoting that an arbitrary update $\mathcal{U}$ is added in front of each formula of $\Gamma \vdash \Delta$. For such a rule, we can derive a meaning formula exactly as in Sect. 5.2, with the only difference that the whole formula is preceded with the update $\mathcal{U}$:

$$\mathcal{U}\left(\bigwedge_{i=1}^{k}\left(\mathtt{rw}i^* \vee \mathtt{add}i^*\right) \rightarrow \left(\mathtt{findSeq}^* \vee \mathtt{assum}^*\right)\right) \tag{8}$$

Fortunately, now the update $\mathcal{U}$ can be left out: because $\mathcal{U}$ can be the empty update `skip`, the validity of (8) entails that also the formula after $\mathcal{U}$ has to be valid. But if the formula after $\mathcal{U}$ is logically valid, i.e., if it is true for all structures and states, then (8) also has to hold for arbitrary updates $\mathcal{U}$. We can thus define the meaning formula of `t3` as in Sect. 5.2:

$$M(\mathtt{t3}) \;=\; \bigwedge_{i=1}^{k}\left(\mathtt{rw}i^* \vee \mathtt{add}i^*\right) \rightarrow \left(\mathtt{findSeq}^* \vee \mathtt{assum}^*\right) \tag{9}$$

Rewriting Taclets with Default Mode

The second and more difficult case concerns rewriting taclets where the application focus can be in the scope of arbitrary modal operators. We consider a taclet similar to the one treated in Sect. 5.3.

Taclet

```
t4 { \assumes(assum) \find(findTerm)
     \replacewith(rw1) \add(add1);
     ...
     \replacewith(rwk) \add(addk) };
```

Taclet

The strategy followed in Sect. 5.3 for deriving a meaning formula for `t2` was to find an equivalent non-rewriting taclet. For `t4`, such a taclet would need to have the following shape:

Taclet

```
t4b { \assumes(assum) \inSequentState
      \add( (∀𝒰.𝒰 (findTerm=rw1) ==>) ∪ add1 );
      ...
      \add( (∀𝒰.𝒰 (findTerm=rwk) ==>) ∪ addk ) };
```

Taclet

The quantifiers $\forall\mathcal{U}.$ have to be added in order to ensure that the inserted equations are also applicable in the scope of modal operators. Such quantifiers over states do not exist in our dynamic logic, but can be added in a straight-forward way (they are, in fact, present in the KeY implementation in a similar form). The meaning formula of `t4b` would be

$$M(\texttt{t4b}) \;=\; \bigwedge_{i=1}^{k} \left(\forall\mathcal{U}.\mathcal{U}\ (\texttt{findTerm} \doteq \texttt{rw}i) \rightarrow \texttt{add}i^*\right) \rightarrow \texttt{assum}^*$$

5.5 Meaning Formulae for Nested Taclets

So far, only taclets were considered that do not contain the `\addrules` clause (⇒ Sect. 4.6). The keyword `\addrules` makes it possible to nest taclets and to use one taclet as part of another. For a start, we will consider taclets of the following shape:

Taclet

```
t3 { \assumes(assum) \find(findSeq) \sameUpdateLevel
     \replacewith(rw1) \add(add1) \addrules(s1_1; ...; s1_m1);
     ...
     \replacewith(rwk) \add(addk) \addrules(sk_1; ...; sk_mk) };
```

Taclet

where s1_1, ..., sk_mk are again taclets (we will call them *sub-taclets* in the next paragraphs). We can construct meaning formulae of such taclets recursively and using a similar argument as in Sect. 5.3 about rewriting taclets. Essentially, one can imagine replacing taclet t3 with a taclet that introduces the meaning formulae of the sub-taclets s1_1, ... in the antecedent using the \add clause:

Taclet

```
t3b {
  \assumes(assum) \find(findSeq) \sameUpdateLevel
  \replacewith(rw1) \add((M(s1_1), ..., M(s1_m1) ==>) ∪ add1);
  ...
  \replacewith(rwk) \add((M(sk_1), ..., M(sk_mk) ==>) ∪ addk) };
```

Taclet

This is not directly possible, because the meaning formulae of the sub-taclets will contain schema variables whose instantiation is not yet determined when applying t3, but it leads us to the following variant of Eq. (5):

$$M(\mathtt{t3}) \;=\; \begin{array}{l} \bigwedge_{i=1}^{k} \left(M(\mathtt{s}i\mathtt{\_1}) \wedge \cdots \wedge M(\mathtt{s}i\mathtt{\_m}i) \rightarrow (\mathtt{rw}i^{*} \vee \mathtt{add}i^{*})\right) \\ \rightarrow \left(\mathtt{findSeq}^{*} \vee \mathtt{assum}^{*}\right) \end{array}$$

In the same way, Eq. (6) and Eq. (7) can be extended to take sub-taclets into account.

Example 12. In order to illustrate meaning formulae for nested taclets, we consider the taclet applyEqAR (⇒ Sect. 1). The meaning formulae for the sub-taclet rewrWithEq and the complete taclet are

$$\begin{array}{rcl} M(\mathtt{rewrWithEq}) & = & \mathtt{t} \doteq \mathtt{t2} \\ M(\mathtt{applyEqAR}) & = & M(\mathtt{rewrWithEq}) \vee \mathtt{t} \not\doteq \mathtt{t2} \\ & = & \mathtt{t} \doteq \mathtt{t2} \vee \mathtt{t} \not\doteq \mathtt{t2} \end{array}$$

Obviously, $M(\mathtt{rewrWithEq})$ is not a valid formula for most instantiations of the variables t and t2, which reflects the observation from Sect. 4.6 that the taclet is not correct in general. As $M(\mathtt{applyEqAR})$ is a tautology, however, rewrWithEq is correct in situations in which applyEqAR can be applied, which distinguishes admissible instantiations of t and t2.

Unfortunately, there is one difficulty when dealing with nested taclets. For some taclets, which we consider in the following as ill-formed, the meaning formulae defined so far do not ensure soundness:

Example 13. We derive the meaning formula of the following taclet, which—at first glance—seems to implement the cut rule, but which in fact can be used to add arbitrary formulae to a sequent:

Taclet

```
illegalTac3 { \addrules( introduceRight { \add(==> phi) } );
              \addrules( introduceLeft  { \add(phi ==>) } ) };
```

Taclet

The "meaning formula" is the tautology $M(\texttt{illegalTac3}) \equiv \texttt{phi} \vee \neg\texttt{phi}$, however. This does not reflect that the two occurrences of `phi` can be instantiated independently when applying `introduceRight` and `introduceLeft`.

The problem with taclets like this is that different instantiations of schema variables can be chosen when applying the sub-taclets, whereas one schema variable will only represent one and the same expression in the meaning formula. `illegalTac3` can be corrected to a taclet that better reflects the nature of schema variables in sub-taclets:

Taclet

```
legalTac3 { \addrules( introduceRight { \add(==> phi1) } );
            \addrules( introduceLeft  { \add(phi2 ==>) } ) };
```

Taclet

Now, the meaning formula is $M(\texttt{legalTac3}) \equiv \neg\texttt{phi1} \vee \texttt{phi2}$ and is no longer valid.

The following requirement will prohibit taclets like `illegalTac3` and could be seen as an item that belongs to Sect. 4.2 about well-formedness of taclets. It is, however, only important when deriving meaning formulae of taclets (it is irrelevant for the effect of taclets according to Sect. 4.5), and we assume only in this section that it is satisfied by considered taclets. We demand that common schema variables of sub-taclets of a taclet t also appear in t outside of sub-taclets, which entails that they are already instantiated when applying t. Arbitrary taclets can easily be transformed into equivalent taclets that respect this property.

Requirement 3 (Uniqueness of Variables in Sub-Taclets). *If a taclet t has two sub-taclets containing a common schema variable* `sv`*, then* `sv` *also appears in t outside of* `\addrules` *clauses.*

5.6 Elimination of Schema Variables

Meaning formulae of taclets in general contain schema variables, i.e., placeholders for syntactic constructs like terms, formulae or programs. In order to prove a taclet sound, it is necessary to show that its meaning formula is valid for all possible instantiations of the schema variables. Looking at Example 12, for instance, we would have to prove the formula

$$M(\texttt{applyEqAR}) \quad = \quad \texttt{t} \doteq \texttt{t2} \vee \texttt{t} \not\doteq \texttt{t2}$$

for all terms $\iota(\texttt{t})$, $\iota(\texttt{t2})$ that we can substitute for `t`, `t2`. Note, that this *syntactic* quantification ranges over terms and is completely different from a first order

formula $\forall x : \mathit{integer}.\, p(x)$, which is *semantic* and expresses that x ranges over all integers.

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

$$\phi \quad = \quad c \doteq d \vee c \not\doteq d$$

in which c, d are fresh constant symbols. The validity of $M(\texttt{applyEqAR})$ for all other instantiations follows, because the symbols c, d can take the values of arbitrary terms $\iota(\texttt{t})$, $\iota(\texttt{t2})$. Fortunately, ϕ is only a first order formula that can be tackled with a calculus as defined in [13].

We will only sketch how Skolem expressions can be introduced for some of the schema variable kinds that are described in Sect. 2. Schema variables for program entities will be left out at this point, a detailed description that also covers such variables can be found in [14]. Also, more involved features like generic types will not be considered here. For the rest of the section, we assume that a taclet t and its meaning formula $M(t)$ are fixed. We will then construct an instantiation ι_{sk} of the schema variables that turn up in t with Skolem expressions. In the example above, this instantiation would be

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

Variables: `\variables` A

Because of Def. 13, instantiations of schema variables `va` for logical variables are always distinct. Such variables only occur bound in taclets and the identity of bound variables does not matter. Therefore, $\iota_{\text{sk}}(\texttt{va})$ can simply be chosen to be a fresh logical variable $\iota_{\text{sk}}(\texttt{va}) = x$ of type A.

Terms: `\term` A

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

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

As a further complication, the symbol f_{sk} has to be *non-rigid* (unless the schema variable modifier `rigid` is used), because the term that is represented by `te` can also be non-rigid. This entails that updates in front of f_{sk} matter, in contrast to rigid function symbols where such updates can immediately be removed, e.g.

$$\{o.a := 3\} f_{\text{sk}}(x) \quad \neq \quad f_{\text{sk}}(x)$$

Formulae: `\formula`

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

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

Skolem Terms: `\skolemTerm` A

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

We can handle schema variables `sk` for Skolem terms more faithfully: if in implication (1) the sequents $P_1, \ldots, P_k$ contain symbols that do not occur in Q, then these symbols can be regarded as universally quantified. Because a negation occurs in front of the quantifiers in (3) (the quantifiers are on the left side of an implication), the symbols have to be considered as existentially quantified when looking at the whole meaning formula. This entails that schema variables for Skolem terms can be eliminated and replaced with existentially quantified variables: $\iota_{\text{sk}}(\texttt{sk}) = x$, where x is a fresh variable of type A.[14] At the same time, an existential quantifier $\exists x.$ has to be added in front of the whole meaning formula.

Example 14. The meaning formula of the taclet `allRight` ($\Rightarrow$ Sect. 1) is

$$M(\texttt{allRight}) \quad = \quad \{\backslash\text{subst } \texttt{x};\ \texttt{cnst}\}(\texttt{phi}) \rightarrow \forall \texttt{x}.\, \texttt{phi}$$

In order to eliminate the schema variables of this taclet, we first assume that the generic type `G` of the taclet is instantiated with a concrete type A. Then,

[14] Strictly speaking, this violates Def. 5, because schema variables for Skolem terms must not be instantiated with variables according to this definition. The required generalisation of the definition is, however, straightforward.

the schema variable `x` can be replaced with a fresh logical variable $\iota_{\mathrm{sk}}(\mathtt{x}) = y$ of type A. The schema variable `phi` is eliminated through the instantiation $\iota_{\mathrm{sk}}(\mathtt{phi}) = p_{\mathrm{sk}}(y)$, where p_{sk} is a fresh non-rigid predicate symbol. Finally, we can replace the schema variable `cnst` for Skolem terms with a fresh logical variable $\iota_{\mathrm{sk}}(\mathtt{cnst}) = z$ of type A and add an existential quantifier $\exists z.$. The resulting formula without schema variables is

$$\exists z.\, \iota_{\mathrm{sk}}(M(\mathtt{allRight})) \quad = \quad \exists z.\, (p_{\mathrm{sk}}(z) \rightarrow \forall y.\, p_{\mathrm{sk}}(y))$$

Acknowledgements

I would like to thank the people that gave feedback and comments on this chapter: Wolfgang Ahrendt, Yves Bertot, Reiner Hähnle, Larry Paulson and Andreas Roth.

References

1. Beckert, B.: A dynamic logic for the formal verification of JavaCard programs. In Attali, I., Jensen, T., eds.: Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France. Volume 2041 of LNCS., Springer (2001) 6–24
2. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
3. Giese, M.: Taclets and the KeY prover. In Aspinall, D., Lüth, C., eds.: Proceedings, User Interfaces for Theorem Provers Workshop, Rome, Italy. Electronic Notes in Theoretical Computer Science, Elsevier (2004)
4. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, IJCAR, Siena, Italy. Volume 2083 of LNAI., Springer (2001) 545–560
5. Habermalz, E.: Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln. PhD thesis, Universität Karlsruhe (2000)
6. Habermalz, E.: Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe (2000) `http://i12www.ira.uka.de/~key/doc/2000/stsr.ps.gz`.
7. Geisler, R., Klar, M., Cornelius, F.: InterACT: An interactive theorem prover for algebraic specifications. In Wirsing, M., Nivat, M., eds.: Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany, July 1–5, 1996, Proceedings. Volume 1101 of LNCS., Springer (1996) 563–566
8. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
9. Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., Werner, B.: The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France (1993) Version 5.8.
10. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In Alur, R., Henzinger, T.A., eds.: Proceedings, CAV. Volume 1102 of LNCS., Springer (1996) 411–414

11. Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Roth, A., Rümmer, P., Schlager, S.: Taclets: a new paradigm for constructing interactive theorem provers. Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas **98** (2004) Special Issue on Symbolic Computation in Logic and Artificial Intelligence.
12. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
13. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software: The KeY Approach. LNAI. Springer (2006) To appear.
14. Bubel, R., Roth, A., Rümmer, P.: Ensuring correctness of lightweight tactics for JavaCard dynamic logic. In: Informal Proceedings of Workshop on Logical Frameworks and Meta-Languages (LFM) at IJCAR 2004. (2004) 84–105
15. Trentelman, K.: Proving correctness of JavaCard DL Taclets using Bali. In Beckert, B., Aichernig, B., eds.: Proceedings, International Conference on Software Engineering and Formal Methods, IEEE Computer Science (2005)
16. Oheimb, D.v., Nipkow, T.: Machine-checking the Java specification: Proving type-safety. In Alves-Foss, J., ed.: Formal Syntax and Semantics of Java. Volume 1523 of LNCS. Springer (1999)
17. Ahrendt, W., Roth, A., Sasse, R.: Automatic validation of transformation rules for java verification against a rewriting semantics. In Sutcliff, G., Voronkov, A., eds.: Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica. Volume 3835 of LNCS., Springer (2005) 412–426
18. Meseguer, J., Rosu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: Proceedings, IJCAR, Cork, Ireland. Volume 3097 of LNCS., Springer (2004)

Paper 2

Sequential, Parallel, and Quantified Updates of First-Order Structures

Philipp Rümmer

Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden
philipp@cs.chalmers.se

Abstract. We present a datastructure for storing memory contents of imperative programs during symbolic execution—a technique frequently used for program verification and testing. The concept, called updates, can be integrated in dynamic logic as runtime infrastructure and models both stack and heap. Here, updates are systematically developed as an imperative programming language that provides the following constructs: assignments, guards, sequential composition and bounded as well as unbounded parallel composition. The language is equipped both with a denotational semantics and a correct rewriting system for execution, whereby the latter is a generalisation of the syntactic application of substitutions. The normalisation of updates is discussed. All results and the complete theory of updates have been formalised and proven using the Isabelle/HOL proof assistant.

1 Introduction

First-Order Dynamic Logic [1] is a program logic that enables to reason about the relation between pre- and post-states of imperative programs. One way to build calculi for dynamic logic is to follow the symbolic execution paradigm and to execute programs (symbolically) in forward direction. This requires infrastructure for storing the memory contents of the program, for updating the contents when assignments occur and for accessing information whenever the program tries to read from memory. Sequent calculi for dynamic logic often represent memory using formulas and handle state changes by renaming variables and by relating pre- and post-states with equations. All information about the considered program states is stored in the side-formulas of a sequent $\Gamma \vdash \langle\alpha\rangle\, \phi, \Delta$.

As an alternative, this paper presents a datastructure called *Updates*, which are a generalisation of substitutions designed for storing symbolic memory contents. When using updates, typical sequents during symbolic execution have the shape $\Gamma \vdash \{u\}\, \langle\alpha\rangle\, \phi, \Delta$. The program α is preceded by an update u that can determine parts of the program state. Compared with side-formulas, updates (i) attach information about the program state directly to the program, (ii) avoid the introduction of new symbols, (iii) can be simplified and avoid the storage of obsolete information, like of assignments that have been overridden by other assignments, (iv) represent accesses to variables, array cells or instance attributes

(in object-oriented languages) in a uniform way, (v) delay case-distinctions that can become necessary due to aliasing, (vi) can be eliminated mechanically once a program has been worked off completely.

Historically, updates have evolved over years as a central component of the KeY system [2]—a system for deductive verification of imperative programs. They are used both for interactive and automated verification. In the present paper, we define updates as a formal language (independently of particular program logics) and give them a denotational semantics based on model-theoretic semantics of first-order predicate logic. The language is proposed as an intermediate language to which sequential parts of more complicated languages (like Java) can stepwise be translated. In order to mechanically compute the effect of updates, we give a rewriting system that allows to simplify, execute or eliminate updates. Further rewriting rules and identities enable simplification and normalisation. The main contributions of the paper are new update constructs (in particular quantification), the development of a complete metatheory of updates and its formalisation[1] using the Isabelle/HOL proof assistant [3], including proofs of all lemmas that are given in the present paper.

The paper is organised as follows: Sect. 2 gives an example for the application of updates as a runtime infrastructure. Sect. 3 and 4 introduce syntax and semantics of a basic version of updates in the context of a minimalist first-order logic. Sect. 5 and 6 contain the rewriting system for executing updates. Sect. 7 adds an operator for sequential composition to the update language. Sect. 8 states soundness and completeness of the rewriting system for update application. Sect. 9 shows how stack and heap structures can be modelled and modified using updates, which is applied in Sect. 10 about symbolic execution. Sect. 11 discusses laws for simplification of updates, and Sect. 12 sketches a method for normalisation of updates.

2 Updates for Symbolic Execution in Dynamic Logic

We give an example for symbolic execution using updates in dynamic logic. Notation and constructs used here are later introduced in detail. The program fragment in question is written in a Java-like language and is executed in the context of a class/record *List* representing doubly-linked lists with attributes *next*, *prev* and *val* for the successor, predecessor and value of list nodes:

$$max \quad = \quad \texttt{if}\ (a.val \doteq a.next.val)\ g = a.next.val; \texttt{else}\ g = a.val;$$

where a and g are program variables pointing to list nodes. The initial state of program execution is specified in an imperative way using an update:

$$\begin{aligned} init \quad = \quad & a.prev := nil \mid b.next := nil \mid a.next := b \mid b.prev := a \mid \\ & a.val := c \mid b.val := d \end{aligned}$$

[1] `www.cs.chalmers.se/~philipp/updates.thy`, $\approx$ 3500 lines Isabelle/Isar code

init can be read as an program that is executing a number of assignments in parallel and that is setting up a list with nodes a and b. In case $a \doteq b$—which is possible because we do not specify the opposite—the two nodes will collapse to the single node of a cyclic list and will carry value d: assignments that literally occur later ($b.val := d$) can override earlier assignments ($a.val := c$). This means that parallel composition in updates also has a sequential component: while the left- and right-hand sides of the assignments are all evaluated in parallel, the actual writing to locations is carried out sequentially from left to right.

When adding updates to a dynamic logic, they can be placed in front of modal operators for programs, like in $\{init\}\ \langle max\rangle\ \phi$. The diamond formula $\langle max\rangle\ \phi$ alone expresses that a given formula ϕ holds in at least one final state of max. Putting the update *init* in front means that first *init* and then the program max is supposed to be executed—*init* sets up the pre-state of max.

We execute max symbolically by working off the statements in forward direction. Effects of the program are either appended to the update *init* or are translated to first-order connectives. We denote execution steps of max by $\rightsquigarrow$ and write $\equiv$ for an update simplification step. *init* is used as an abbreviation.

$$\{init\}\ \langle \texttt{if}\ (a.val \dot{<} a.next.val)\ g = a.next.val; \texttt{else}\ g = a.val\rangle\ \phi$$

A conditional statement can be translated to propositional connectives. The branch condition is $co = (a.val \dot{<} a.next.val)$.

$$\rightsquigarrow\quad \{init\}\ \big((co \wedge \langle g = a.next.val\rangle\ \phi) \vee (\neg co \wedge \langle g = a.val\rangle\ \phi)\big)$$

The application of *init* distributes through propositional connectives. Applying *init* to *co* yields the condition $co' = (\{init\}\ co) \equiv ((\text{if}\ \ a \doteq b\ \text{ then }\ d\ \text{ else }\ c) \dot{<} d)$.

$$\equiv\quad (co' \wedge \{init\}\ \langle g = a.next.val\rangle\ \phi \vee (\neg co' \wedge \{init\}\ \langle g = a.val\rangle\ \phi)$$

The program assignments are turned into update assignments that are sequentially (;) connected with *init*.

$$\rightsquigarrow\quad (co' \wedge \{init\,;\ g := a.next.val\}\ \phi) \vee (\neg co' \wedge \{init\,;\ g := a.val\}\ \phi)$$

The updates are simplified by turning sequential composition ; into parallel composition | . The update *init* has to be applied to the right-hand sides, which become $(\{init\}\ a.next.val) \equiv d$ and $(\{init\}\ a.val) \equiv (\text{if}\ \ a \doteq b\ \text{ then }\ d\ \text{ else }\ c)$.

$$\equiv\quad (co' \wedge \{init \mid g := d\}\ \phi) \vee (\neg co' \wedge \{init \mid g := (\text{if}\ \ a \doteq b\ \text{ then }\ d\ \text{ else }\ c)\}\ \phi)$$

The last formula is logically equivalent to the original formula $\{init\}\ \langle max\rangle\ \phi$ and can further be simplified by applying the updates to ϕ. In all points of the proof, updates in front of programs specify the memory contents. An implementation like in KeY can, of course, easily carry out all shown steps automatically.

3 Syntax of Terms, Formulas and Updates

The present paper is a self-contained account on updates. To this end, we abstract from concrete program logics and define syntax and semantics of a (min-

imalist)[2] first-order logic that is equipped with updates. Updates can, however, be integrated in virtually any predicate logic, e.g., in dynamic logic.

We first define a basic version of our logic that contains the most common constructors for terms and formulas (see e.g. [4]), the equality predicate $\doteq$ and a strict order relation $\dot{<}$, as well as operators for minimum and conditional terms. The two latter are not strictly necessary, but enable a simpler definition of laws and rewriting rules. In this section, updates are only equipped with the connectives for parallelism, guards and quantification, sequential composition is added later in Sect. 7.

In order to define the syntax of the logic, we need (i) a vocabulary (Σ, α) of function symbols, where $\alpha : \Sigma \to \mathbb{N}$ defines the arity of each symbol, and (ii) an infinite set *Var* of variables.

Definition 1. *The sets Ter, For and Upd of terms, formulas and updates are defined by the following grammar, in which $x \in Var$ ranges over variables and $f \in \Sigma$ over functions:*

$$
\begin{array}{rcl}
\mathit{Ter} & ::= & x \mid f(\mathit{Ter}, \ldots, \mathit{Ter}) \mid \text{if } \mathit{For} \text{ then } \mathit{Ter} \text{ else } \mathit{Ter} \mid \min x.\, \mathit{For} \mid \{\mathit{Upd}\}\ \mathit{Ter} \\
\mathit{For} & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{For} \wedge \mathit{For} \mid \mathit{For} \vee \mathit{For} \mid \neg \mathit{For} \mid \forall x.\, \mathit{For} \mid \exists x.\, \mathit{For} \mid \\
 & & \mathit{Ter} \doteq \mathit{Ter} \mid \mathit{Ter} \dot{<} \mathit{Ter} \mid \{\mathit{Upd}\}\ \mathit{For} \\
\mathit{Upd} & ::= & \texttt{skip} \mid f(\mathit{Ter}, \ldots, \mathit{Ter}) := \mathit{Ter} \mid \mathit{Upd} \mid \mathit{Upd} \mid \texttt{if } \mathit{For} \ \{\mathit{Upd}\} \mid \texttt{for } x\ \{\mathit{Upd}\}
\end{array}
$$

The update constructors represent the empty update `skip`, assignments to function terms $f(s_1, \ldots, s_n) := t$, parallel updates $u_1 \mid u_2$, guarded updates $\texttt{if } \phi\ \{u\}$, and quantified updates $\texttt{for } x\ \{u\}$. The possibility of having function terms as left-hand sides of assignments is crucial for modelling heaps. In Sect. 2, expressions like *a.prev* are really function terms $\mathit{prev}(a)$, but we use the more common notation from programming languages. More details are given in Sect. 9. There are also constructors for applying updates to terms and to formulas (like $\{u\}\ \phi$).

We mostly use vector notation for the arguments $\bar{t}$ of functions. Operations on terms are extended canonically or in an obvious way to vectors, for instance $f(\{u\}\ \bar{t}) = f(\{u\}\ t_1, \ldots, \{u\}\ t_n)$, $\mathrm{val}_{S,\beta}(\bar{t}) = (\mathrm{val}_{S,\beta}(t_1), \ldots, \mathrm{val}_{S,\beta}(t_n))$, $\mathrm{fv}(\bar{t}) = \bigcup_i \mathrm{fv}(t_i)$, $(\bar{t} \doteq \bar{s}) = (t_1 \doteq s_1 \wedge \cdots \wedge t_n \doteq s_n)$.

4 Semantics of Terms, Formulas and Updates

The meaning of terms and formulas is defined using classical model-theoretic semantics. We consider interpretations as mappings from *locations* to *individuals* of a universe U (the predicates $\doteq$ and $\dot{<}$ are handled separately):

Definition 2. *Given a vocabulary (Σ, α) of function symbols and an arbitrary set U, we define the set $Loc_{(\Sigma,\alpha),U}$ of* locations *over (Σ, α) and U by*

$$Loc_{(\Sigma,\alpha),U} := \{\langle f, (a_1, \ldots, a_n)\rangle \mid f \in \Sigma,\ \alpha(f) = n,\ a_1, \ldots, a_n \in U\}$$

[2] We do not include many common features like arbitrary predicate symbols, in order to keep the presentation concise. Adding such concepts is straightforward.

If the indexes are clear from the context, we just write Loc instead of $Loc_{(\Sigma,\alpha),U}$.

The following definition of structures/algebras deviates from common definitions in the addition of a strict well-ordering on the universe.[3] The well-ordering is used for resolving clashes that can occur in quantified updates (see Example 1 and Sect. 10).

Definition 3. *Suppose that a vocabulary* (Σ, α) *of function symbols is given. A* well-ordered algebra *over* (Σ, α) *is a tuple* $S = (U, <, I)$*, where*

- U *is an arbitrary non-empty set (the* universe*),*
- $<$ *is a strict well-ordering on* U*, i.e., a binary relation with the properties*[4]
  - *Irreflexivity:* $a \not< a$ *for all* $a \in U$*,*
  - *Transitivity:* $a_1 < a_2$*,* $a_2 < a_3$ *entails* $a_1 < a_3$ $(a_1, a_2, a_3 \in U)$*,*
  - *Well-orderedness: Each set* $\emptyset \neq A \subseteq U$ *contains an element* $\min_< A \in A$ *such that* $\min_< A < a$ *for all* $a \in A \backslash \{\min_< A\}$*,*
- I *is a (total) mapping* $Loc_{(\Sigma,\alpha),U} \to U$ *(the* interpretation*).*

A partial interpretation *is a partial function* $Loc_{(\Sigma,\alpha),U} \nrightarrow U$.

A (partial) function $f : M \nrightarrow N$ is here considered as a subset of the cartesian product $M \times N$. For combining and modifying interpretations, we frequently make use of the *overriding* operator $\oplus$, which can be found in Z [6] and many other specification languages. For two (partial or total) functions $f, g : M \nrightarrow N$ we define

$$f \oplus g \;:=\; \{(a \mapsto b) \in f \mid \text{for all } c\text{: } (a \mapsto c) \notin g\} \cup g$$

i.e., g overrides f but leaves f unchanged at points where g is not defined. For $S = (U, <, I)$, we also write $S \oplus A := (U, <, I \oplus A)$ as a shorthand notation.

Definition 4. *A variable assignment over a set Var of variables and a well-ordered algebra* $(U, <, I)$ *is a mapping* $\beta : Var \to U$.

Given a variable assignment β, we denote the assignment that is altered in exactly one point as is common:

$$\beta^a_x(y) \;:=\; \begin{cases} a & \text{for } x = y \\ \beta(y) & \text{otherwise} \end{cases}$$

From now on, we consider the vocabulary (Σ, α) and *Var* as fixed.

[3] As every set can be well-ordered (based on Zermelo-Fraenkel set theory [5]) this does not restrict the range of considered universes. Because the well-ordering is also accessible through the predicate $\dot{<}$, however, the expressiveness of the logic goes beyond pure first-order predicate logic. One can, for instance, axiomatise natural numbers up to isomorphism with a finite set of formulas. In our experience, this is not a problem for the application of updates, because quantification in updates will in practice only be used for variables representing integers, objects or similar types. On such domains, appropriate well-orderings are readily available and have to be handled anyway.

[4] Note, that well-orderings are linear, i.e., $a < b$, $a = b$, or $b < a$ for arbitrary $a, b \in U$. Further, well-orderings are well-founded—there are no infinite descending chains—which enables us to use well-founded recursion when defining update evaluation.

Definition 5. *Given a well-ordered algebra $S = (U, <, I)$ and a variable assignment β, we define the* evaluation *of terms, formulas and updates through the equations of Table 1 as the (overloaded) mapping*

$$\mathrm{val}_{S,\beta} : \mathit{Ter} \to U, \quad \mathrm{val}_{S,\beta} : \mathit{For} \to \{tt, ff\}, \quad \mathrm{val}_{S,\beta} : \mathit{Upd} \to (\mathit{Loc} \nrightarrow U),$$

i.e., in particular updates are evaluated to partial interpretations.

The most involved part of the update evaluation concerns quantified expressions $\texttt{for}\ x\ \{u\}$, whose value is defined by well-founded recursion on $(U, <)$. The definition shows that quantification is a generalisation of parallel composition: Informally, for a well-ordered universe $U = \{a < b < c < \cdots\}$ we have

$$\mathrm{val}_{S,\beta}(\texttt{for}\ x\ \{u\}) \;=\; \cdots \;\oplus \mathrm{val}_{S,\beta^c_x}(u) \oplus \mathrm{val}_{S,\beta^b_x}(u) \oplus \mathrm{val}_{S,\beta^a_x}(u)$$

For a general definition (see Table 1) of the partial interpretation on the right-hand side, we need a union operator on partial functions:[5]

$$\big(\bigcup M\big)(x) = \begin{cases} f(x) & \text{if there is } f \in M \text{ with } f(x) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

where we write $f(x) = \bot$ if a partial function f is not defined at point x.

Example 1. The following examples refer to the well-ordered algebra $(\mathbb{N}, <, I)$, where $<$ is the standard order on $\mathbb{N}$. We assume that the vocabulary contains literals and operations $+, \cdot$, and that these symbols are interpreted as usual for $\mathbb{N}$.

$$\mathrm{val}_{S,\beta}(a := 2) \;=\; \{\langle a\rangle \mapsto 2\}$$

In parallel composition, the effect of the left update is invisible to the right one:

$$\mathrm{val}_{S,\beta}(a := 2 \mid f(a) := 3) \;=\; \{\langle a\rangle \mapsto 2,\ \langle f, (\mathrm{val}_{S,\beta}(a))\rangle \mapsto 3\}$$

The right update in parallel composition overrides the left update when clashes occur. Here, this happens for $\mathrm{val}_{S,\beta}(a) = 1$:

$$\mathrm{val}_{S,\beta}(f(a) := 1 \mid f(1) := 2) \;=\; \{\langle f, (1)\rangle \mapsto 2\}$$

In contrast, for $\mathrm{val}_{S,\beta}(a) \neq 1$ both assignments have an effect:

$$\mathrm{val}_{S,\beta}(f(a) := 1 \mid f(1) := 2) \;=\; \{\langle f, (\mathrm{val}_{S,\beta}(a))\rangle \mapsto 1,\ \langle f, (1)\rangle \mapsto 2\}$$

Quantified updates make it possible to define whole functions:

$$\mathrm{val}_{S,\beta}(\{\texttt{for}\ x\ \{f(x) := 2 \cdot x + 1\}\}\ f(5)) \;=\; 11$$

When clashes occur in quantified updates, smaller valuations of the quantified variable will dominate. The smallest individual of $(\mathbb{N}, <)$ is 0:

$$\mathrm{val}_{S,\beta}(\texttt{for}\ x\ \{a := x\}) \;=\; \{\langle a\rangle \mapsto 0\}$$

[5] The operator $\bigcup$ is obviously not uniquely defined by the given equation, but because of $A(a) \subseteq A(b)$ for $a < b$ its result is unique when defining the evaluation function.

Table 1. Evaluation of Terms, Formulas and Updates

For terms:

$$\begin{aligned}
\mathrm{val}_{S,\beta}(x) &= \beta(x) && (x \in \mathit{Var})\\
\mathrm{val}_{S,\beta}(f(\bar{t})) &= I\langle f, \mathrm{val}_{S,\beta}(\bar{t})\rangle && (S = (U, <, I))\\
\mathrm{val}_{S,\beta}(\text{if } \phi \text{ then } t_1 \text{ else } t_2) &= \begin{cases} \mathrm{val}_{S,\beta}(t_1) & \text{for } \mathrm{val}_{S,\beta}(\phi) = tt \\ \mathrm{val}_{S,\beta}(t_2) & \text{otherwise} \end{cases}\\
\mathrm{val}_{S,\beta}(\min x.\, \phi) &= \begin{cases} \min_< A & \text{for } A \neq \emptyset \\ \min_< U & \text{otherwise} \end{cases}
\end{aligned}$$

where $S = (U, <, I)$ and $A = \{a \in U \mid \mathrm{val}_{S,\beta^a_x}(\phi) = tt\}$

For formulas:

$$\begin{aligned}
\mathrm{val}_{S,\beta}(\mathit{true}) = tt, &\qquad \mathrm{val}_{S,\beta}(\mathit{false}) = \mathit{ff}\\
\mathrm{val}_{S,\beta}(\phi_1 \wedge \phi_2) = tt &\quad \text{iff} \quad \mathit{ff} \notin \{\mathrm{val}_{S,\beta}(\phi_1), \mathrm{val}_{S,\beta}(\phi_2)\}\\
\mathrm{val}_{S,\beta}(\phi_1 \vee \phi_2) = tt &\quad \text{iff} \quad tt \in \{\mathrm{val}_{S,\beta}(\phi_1), \mathrm{val}_{S,\beta}(\phi_2)\}\\
\mathrm{val}_{S,\beta}(\neg\phi) = tt &\quad \text{iff} \quad \mathrm{val}_{S,\beta}(\phi) = \mathit{ff}\\
\mathrm{val}_{S,\beta}(\forall x.\, \phi) = tt &\quad \text{iff} \quad \mathit{ff} \notin \{\mathrm{val}_{S,\beta^a_x}(\phi) \mid a \in U\}\\
\mathrm{val}_{S,\beta}(\exists x.\, \phi) = tt &\quad \text{iff} \quad tt \in \{\mathrm{val}_{S,\beta^a_x}(\phi) \mid a \in U\}\\
\mathrm{val}_{S,\beta}(t_1 \doteq t_2) = tt &\quad \text{iff} \quad \mathrm{val}_{S,\beta}(t_1) = \mathrm{val}_{S,\beta}(t_2)\\
\mathrm{val}_{S,\beta}(t_1 \dot{<} t_2) = tt &\quad \text{iff} \quad \mathrm{val}_{S,\beta}(t_1) < \mathrm{val}_{S,\beta}(t_2) \qquad (S = (U, <, I))
\end{aligned}$$

For updates:

$$\begin{aligned}
\mathrm{val}_{S,\beta}(\texttt{skip}) &= \emptyset\\
\mathrm{val}_{S,\beta}(f(\bar{s}) := t) &= \{\langle f, \mathrm{val}_{S,\beta}(\bar{s})\rangle \mapsto \mathrm{val}_{S,\beta}(t)\}\\
\mathrm{val}_{S,\beta}(u_1 \mid u_2) &= \mathrm{val}_{S,\beta}(u_1) \oplus \mathrm{val}_{S,\beta}(u_2)\\
\mathrm{val}_{S,\beta}(\texttt{if } \phi \texttt{ } \{u\}) &= \begin{cases} \mathrm{val}_{S,\beta}(u) & \text{for } \mathrm{val}_{S,\beta}(\phi) = tt \\ \emptyset & \text{otherwise} \end{cases}\\
\mathrm{val}_{S,\beta}(\texttt{for } x \texttt{ } \{u\}) &= \bigcup \{A(a) \mid a \in U\}
\end{aligned}$$

where $A : U \to (\mathit{Loc} \nrightarrow U)$ is defined by well-founded recursion on $(U, <)$ and the equation $A(a) = \mathrm{val}_{S,\beta^a_x}(u) \oplus \bigcup \{A(b) \mid b \in U,\, b < a\}$

Application of updates: $(S' = S \oplus \mathrm{val}_{S,\beta}(u)$ and $\alpha \in \mathit{Ter} \cup \mathit{For})$

$$\mathrm{val}_{S,\beta}(\{u\}\ \alpha) = \mathrm{val}_{S',\beta}(\alpha)$$

Update constructors can be nested arbitrarily, like in quantified parallel updates:

$$\mathrm{val}_{S,\beta}(\texttt{for } x \ \{(f(x+3) := x \mid f(2\cdot x) := x+1)\}) \ =$$
$$\{\langle f,(3)\rangle \mapsto 0, \langle f,(4)\rangle \mapsto 1, \langle f,(5)\rangle \mapsto 2, \cancel{\langle f,(6)\rangle \mapsto 3}, \langle f,(7)\rangle \mapsto 4, \ldots,$$
$$\langle f,(0)\rangle \mapsto 1, \langle f,(2)\rangle \mapsto 2, \cancel{\langle f,(4)\rangle \mapsto 3}, \langle f,(6)\rangle \mapsto 4, \langle f,(8)\rangle \mapsto 5, \ldots \ \}$$

In the last example, both kinds of clashes occur: (i) the pair $\langle f,(6)\rangle \mapsto 3$ stems from $f(x+3) := x$ and is overridden by $\langle f,(6)\rangle \mapsto 4$ (from $f(2\cdot x) := x+1$), because updates on the right side of parallel composition dominate updates on the left side ("last-win semantics"). (ii) the pair $\langle f,(4)\rangle \mapsto 3$ stems from the valuation $x \mapsto 2$ and is overridden by $\langle f,(4)\rangle \mapsto 1$ (from $x \mapsto 1$), because small valuations of variables dominate larger valuations ("well-ordered semantics").

We formalise the behaviour of updates for the latter kind of clashes:

Lemma 1. *Small valuations of variables in updates override larger ones:*

$$\mathrm{val}_{S,\beta}(\texttt{for } x \ \{u\})(loc) \ = \ \mathrm{val}_{S,\beta^m_x}(u)(loc)$$
$$\textit{where} \quad m = \begin{cases} \min_< A & \textit{for } A \neq \emptyset \\ \textit{arbitrary} & \textit{otherwise} \end{cases} \quad \textit{and} \quad A = \{a \mid \mathrm{val}_{S,\beta^a_x}(u)(loc) \neq \bot\}$$

We can now also introduce the equivalence symbol $\equiv$ used in Sect. 2:

Definition 6. *We call two terms, formulas or updates $\alpha_1, \alpha_2 \in Ter \cup For \cup Upd$* equivalent *and write $\alpha_1 \equiv \alpha_2$ if they are necessarily evaluated to the same value: For all well-ordered algebras S and all variable assignments β over S,*

$$\mathrm{val}_{S,\beta}(\alpha_1) = \mathrm{val}_{S,\beta}(\alpha_2)$$

$\equiv$ is a congruence relation for all constructors given in Def. 1 (see Lem. 2).

5 Application of Updates by Rewriting

Updates do in principle not increase the expressiveness of terms or formulas: Given an arbitrary term, formula or update α, there will always be an equivalent expression $\alpha' \equiv \alpha$ that does not contain the update application operator.[6] We obtain this result by giving a rewriting system that eliminates updates using altogether 44 rules like $\{u\}\ (t_1 * t_2) \to \{u\}\ t_1 * \{u\}\ t_2$ (with $* \in \{\doteq, \dot{<}\}$).

For space reasons, we refrain from giving an introduction to the rewriting concept and instead refer to literature, see for instance [7]. Some of our rules have side-conditions concerning free variables, like $x \notin \mathrm{fv}(u)$, in order to avoid variable capture. We will not dwell on details about bound renaming or give a precise definition of the set $\mathrm{fv}(u)$ of free variables of an expression (see, e.g., [4]), but assume that bound renaming is implicitly applied whenever necessary.

[6] As we have not formally proved that our rewriting system that turns α into α' is terminating (but consider it as obvious), we do not state this as a theorem.

Table 2. Rewriting Rules for the Application of Updates

$\{u\}\ x$	$\to$	x	$(x \in \mathit{Var})$	(R1)
$\{u\}\ f(\bar{t})$	$\to$	$\textsc{non-rec}(u,\ f, \{u\}\ \bar{t})$		(R2)
$\{u\}$ if ϕ then t_1 else t_2	$\to$	if $\{u\}\ \phi$ then $\{u\}\ t_1$ else $\{u\}\ t_2$		(R3)
$\{u\}\ \min x.\, \phi$	$\to$	$\min x.\, \{u\}\ \phi$	$(x \notin \mathrm{fv}(u))$	(R4)
$\{u\}\ \mathit{lit}$	$\to$	lit	$(\mathit{lit} \in \{\mathit{true}, \mathit{false}\})$	(R5)
$\{u\}\ (\phi_1 * \phi_2)$	$\to$	$\{u\}\ \phi_1 * \{u\}\ \phi_2$	$(* \in \{\wedge, \vee\})$	(R6)
$\{u\}\ \neg\phi$	$\to$	$\neg\{u\}\ \phi$		(R7)
$\{u\}\ Q\, x.\, \phi$	$\to$	$Q\, x.\, \{u\}\ \phi$	$(Q \in \{\forall, \exists\},\ x \notin \mathrm{fv}(u))$	(R8)
$\{u\}\ (t_1 * t_2)$	$\to$	$\{u\}\ t_1 * \{u\}\ t_2$	$(* \in \{\doteq, \dot{<}\})$	(R9)
$\textsc{non-rec}(\texttt{skip},\ f, \bar{t})$	$\to$	$f(\bar{t})$		(R10)
$\textsc{non-rec}(f(\bar{s}) := r,\ f, \bar{t})$	$\to$	if $\bar{t} \doteq \bar{s}$ then r else $f(\bar{t})$		(R11)
$\textsc{non-rec}(g(\bar{s}) := r,\ f, \bar{t})$	$\to$	$f(\bar{t})$	$(f \neq g)$	(R12)
$\textsc{non-rec}(u_1 \mid u_2,\ f, \bar{t})$	$\to$	if $\textsc{in-dom}(f, \bar{t},\ u_2)$ then $\textsc{non-rec}(u_2,\ f, \bar{t})$ else $\textsc{non-rec}(u_1,\ f, \bar{t})$		(R13)
$\textsc{non-rec}(\texttt{if}\ \phi\ \{u\},\ f, \bar{t})$	$\to$	if ϕ then $\textsc{non-rec}(u,\ f, \bar{t})$ else $f(\bar{t})$		(R14)
For $x \notin \mathrm{fv}(\bar{t})$ and $r = \min x.\, \textsc{in-dom}(f, \bar{t},\ u)$:				
$\textsc{non-rec}(\texttt{for}\ x\ \{u\},\ f, \bar{t})$	$\to$	$\textsc{non-rec}(\{x/r\}\ u,\ f, \bar{t})$		(R15)
$\textsc{in-dom}(f, \bar{t},\ \texttt{skip})$	$\to$	false		(R16)
$\textsc{in-dom}(f, \bar{t},\ f(\bar{s}) := r)$	$\to$	$\bar{t} \doteq \bar{s}$		(R17)
$\textsc{in-dom}(f, \bar{t},\ g(\bar{s}) := r)$	$\to$	false	$(f \neq g)$	(R18)
$\textsc{in-dom}(f, \bar{t},\ u_1 \mid u_2)$	$\to$	$\textsc{in-dom}(f, \bar{t},\ u_1) \vee \textsc{in-dom}(f, \bar{t},\ u_2)$		(R19)
$\textsc{in-dom}(f, \bar{t},\ \texttt{if}\ \phi\ \{u\})$	$\to$	$\phi \wedge \textsc{in-dom}(f, \bar{t},\ u)$		(R20)
$\textsc{in-dom}(f, \bar{t},\ \texttt{for}\ x\ \{u\})$	$\to$	$\exists x.\, \textsc{in-dom}(f, \bar{t},\ u)$	$(x \notin \mathrm{fv}(\bar{t}))$	(R21)
$\textsc{reject}(\texttt{skip}, \overline{u})$	$\to$	$\texttt{skip}$		(R22)
$\textsc{reject}(f(\bar{s}) := t, \overline{u})$	$\to$	$\texttt{if}\ \neg\textsc{in-dom}(f, \bar{s},\ u)\ \{f(\bar{s}) := t\}$		(R23)
$\textsc{reject}(u_1 \mid u_2, \overline{u})$	$\to$	$\textsc{reject}(u_1, \overline{u}) \mid \textsc{reject}(u_2, \overline{u})$		(R24)
$\textsc{reject}(\texttt{if}\ \phi\ \{u_1\}, \overline{u})$	$\to$	$\texttt{if}\ \phi\ \{\textsc{reject}(u_1, \overline{u})\}$		(R25)
$\textsc{reject}(\texttt{for}\ x\ \{u_1\}, \overline{u})$	$\to$	$\texttt{for}\ x\ \{\textsc{reject}(u_1, \overline{u})\}$	$(x \notin \mathrm{fv}(u))$	(R26)

Syntactic application of updates to terms or formulas, i.e., simplification of expressions $\{u\}\ \alpha$, is carried out in two phases: first, the update is propagated to subterms or subformulas. In the second phase, when the update has reached a function application, it is analysed whether the update assigns the represented location. This separation of propagation and evaluation is achieved by introducing (amongst others) a non-recursive update application operator NON-REC. We also add ordinary substitution of variables as an independent operator, which is necessary for handling quantified updates. Substitutions are discussed in Sect. 6.

In order to introduce the further operators, we extend the syntax given Def. 1 as well as the semantics of Def. 5. Practical application is realised by rewriting rules that stepwise eliminate the operators.[7]

Definition 7. *We define the sets Ter_A, For_A and Upd_A of terms, formulas and updates as in Def. 1, but with further constructors ($x \in Var$ ranges over variables and $f \in \Sigma$ over functions):*

$$\begin{aligned} Ter_A &::= \cdots \mid \{x/Ter_A\}\ Ter_A \mid \text{NON-REC}(Upd_A,\ f, (Ter_A, \ldots, Ter_A)) \\ For_A &::= \cdots \mid \{x/Ter_A\}\ For_A \mid \text{IN-DOM}(f, (Ter_A, \ldots, Ter_A),\ Upd_A) \\ Upd_A &::= \cdots \mid \{x/Ter_A\}\ Upd_A \mid \text{REJECT}(Upd_A, \overline{Upd_A}) \end{aligned}$$

The constructors represent the explicit application of substitutions to terms, formulas, and updates (like $\{x/s\}\ t$), the non-recursive application of an update u to function terms $f(\bar{t})$ (like $\text{NON-REC}(u,\ f, \bar{t})$), the test whether an update u assigns to the location denoted by $f(\bar{t})$ (like $\text{IN-DOM}(f, \bar{t},\ u)$), and filtered updates $\text{REJECT}(u_1, \overline{u_2})$ (which are described in Sect. 11). We also extend the evaluation function $\text{val}_{S,\beta}$ on Ter_A, For_A and Upd_A by adding the following clauses:

$$\text{val}_{S,\beta}(\{x/s\}\ \alpha) = \text{val}_{S,\beta'}(\alpha)$$

where $\beta' = \beta_x^{\text{val}_{S,\beta}(s)}$ and $\alpha \in Ter_A \cup For_A \cup Upd_A$

$$\text{val}_{S,\beta}(\text{NON-REC}(u,\ f, \bar{t})) = I'\langle f, \text{val}_{S,\beta}(\bar{t})\rangle$$

where $S = (U, <, I)$ and $I' = I \oplus \text{val}_{S,\beta}(u)$

$$\text{val}_{S,\beta}(\text{IN-DOM}(f, \bar{t},\ u)) = tt \quad \text{iff} \quad \text{val}_{S,\beta}(u)\langle f, \text{val}_{S,\beta}(\bar{t})\rangle \neq \bot$$

$$\text{val}_{S,\beta}(\text{REJECT}(u_1, \overline{u_2})) = \{(loc \mapsto a) \in \text{val}_{S,\beta}(u_1) \mid \text{val}_{S,\beta}(u_2)(loc) = \bot\}$$

The difference between non-recursive application $\text{NON-REC}(u,\ f, \bar{t})$ and ordinary application $\{u\}\ f(\bar{t})$ is that the subterms $\bar{t}$ are in the first case evaluated in the unmodified algebra, whereas in the latter case the algebra is first updated by u. Formally, we have $\{u\}\ f(\bar{t}) \equiv \text{NON-REC}(u,\ f, \{u\}\ \bar{t})$. The non-recursive operator enables us to separate the syntactic propagation of updates to subterms and subformulas from the syntactic evaluation of updates.

The actual syntactic application of updates is described by the rewriting rules in Table 2. Soundness and completeness of the rules is stated in Sect. 8.

[7] Alternatively, one could also give a purely syntactic characterisation in terms of recursively defined functions. For reasoning about correctness and for gaining an intuition of what is happening, however, we believe that a separation of syntax and semantics is beneficial.

6 Application of Substitutions by Rewriting

Table 3 contains rewriting rules that apply substitutions syntactically, which follows the idea of explicit substitutions [8]. The system essentially performs pattern matching and distinguishes the different constructors that can occur after a substitution. No rules exist for the cases substitution and update application. Permuting substitutions with update application is not directly possible, which can be seen for

$$\{x/f(a)\}\ \{f(a) := b\}\ x$$

in which an update modifies the meaning of terms that turn up in the substitution. This problem is related to variable capture (when passing binders), but because updates can also assign non-nullary function symbols, avoidance of capture by means of renaming is more intricate. Instead, in the case above we delay the application of the substitution until the update has been eliminated by rewriting.

When using updates in a logic like dynamic logic, it is common that updates cannot be eliminated completely, e.g. updates in front of programs (see Sect. 2). This implies that also substitutions cannot be eliminated in certain cases. Then, the substitution either has to be kept, or has to be realised by other means like equations.

7 Sequentiality and Application of Updates to Updates

We extend the basic version of updates from Sect. 3 a second time and introduce sequential composition. Sequentiality already occurs when applications of updates are nested, for instance in an expression $\{u_1\}\ \{u_2\}\ \alpha$ for a term or update α. It seems natural to make an operator for sequential composition compatible with nesting of updates: $\{u_1\}\ \{u_2\}\ \alpha \equiv \{u_1\,;\ u_2\}\ \alpha$. Sequential composition of this kind can be reduced to parallel composition by extending the update application operator to updates themselves, i.e., by considering updates $\{u_1\}\ u_2$.

Definition 8. *We define the sets Ter_{AS}, For_{AS} and Upd_{AS} of terms, formulas and updates as in Def. 7, but with two further constructors:*

$$Upd_{AS} \ ::= \ \cdots \mid Upd_{AS}\,;\ Upd_{AS} \mid \{Upd_{AS}\}\ Upd_{AS}$$

Again, the evaluation function is extended to Ter_{AS}, For_{AS} and Upd_{AS} by adding two clauses (in both cases $S' = S \oplus \mathrm{val}_{S,\beta}(u_1)$):

$$\mathrm{val}_{S,\beta}(u_1\,;\ u_2) = \mathrm{val}_{S,\beta}(u_1) \oplus \mathrm{val}_{S',\beta}(u_2), \qquad \mathrm{val}_{S,\beta}(\{u_1\}\ u_2) = \mathrm{val}_{S',\beta}(u_2)$$

The second clause resembles the semantics of update application to terms and formulas. The first clause is very similar to the evaluation of parallel updates, with the only difference that the right update u_2 is evaluated in the structure S' updated by u_1. Intuitively, with parallel composition the effect of u_1 is invisible to u_2 (and vice versa), whereas sequential composition carries out u_1 before

Table 3. Rewriting Rules for the Application of Substitutions

$\{x/s\}\ x$	$\to$	s		(R27)
$\{x/s\}\ y$	$\to$	y	$(x \neq y, y \in \mathit{Var})$	(R28)
$\{x/s\}\ f(\bar{t})$	$\to$	$f(\{x/s\}\ \bar{t})$		(R29)
$\{x/s\}$ if ϕ then t_1 else t_2	$\to$	if $\{x/s\}\ \phi$ then $\{x/s\}\ t_1$ else $\{x/s\}\ t_2$		(R30)
$\{x/s\}\ \min x.\, \phi$	$\to$	$\min x.\, \phi$		(R31)
$\{x/s\}\ \min y.\, \phi$	$\to$	$\min y.\, \{x/s\}\ \phi$	$(x \neq y,\ y \notin \mathrm{fv}(s))$	(R32)
$\{x/s\}\ \mathit{lit}$	$\to$	lit	$(\mathit{lit} \in \{\mathit{true}, \mathit{false}\})$	(R33)
$\{x/s\}\ (\phi_1 * \phi_2)$	$\to$	$\{x/s\}\ \phi_1 * \{x/s\}\ \phi_2$	$(* \in \{\wedge, \vee\})$	(R34)
$\{x/s\}\ \neg\phi$	$\to$	$\neg\{x/s\}\ \phi$		(R35)
$\{x/s\}\ Q\, x.\, \phi$	$\to$	$Q\, x.\, \phi$	$(Q \in \{\forall, \exists\})$	(R36)
$\{x/s\}\ Q\, y.\, \phi$	$\to$	$Q\, y.\, \{x/s\}\ \phi$	$(Q \in \{\forall, \exists\},\ x \neq y,\ y \notin \mathrm{fv}(s))$	(R37)
$\{x/s\}\ (t_1 * t_2)$	$\to$	$\{x/s\}\ t_1 * \{x/s\}\ t_2$	$(* \in \{\doteq, \dot{<}\})$	(R38)
$\{x/s\}$ `skip`	$\to$	`skip`		(R39)
$\{x/s\}\ (f(\bar{r}) := t)$	$\to$	$f(\{x/s\}\ \bar{r}) := \{x/s\}\ t$		(R40)
$\{x/s\}\ (u_1 \mid u_2)$	$\to$	$\{x/s\}\ u_1 \mid \{x/s\}\ u_2$		(R41)
$\{x/s\}$ `if` ϕ $\{u\}$	$\to$	`if` $\{x/s\}\ \phi\ \{\{x/s\}\ u\}$		(R42)
$\{x/s\}$ `for` x $\{u\}$	$\to$	`for` x $\{u\}$		(R43)
$\{x/s\}$ `for` y $\{u\}$	$\to$	`for` y $\{\{x/s\}\ u\}$	$(x \neq y,\ y \notin \mathrm{fv}(s))$	(R44)

Table 4. Rewriting Rules for Sequential Composition

$u_1\,;\ u_2$	$\to$	$u_1 \mid \{u_1\}\ u_2$		(R45)
$\{u\}$ `skip`	$\to$	`skip`		(R46)
$\{u\}\ (f(\bar{s}) := t)$	$\to$	$f(\{u\}\ \bar{s}) := \{u\}\ t$		(R47)
$\{u\}\ (u_1 \mid u_2)$	$\to$	$\{u\}\ u_1 \mid \{u\}\ u_2$		(R48)
$\{u\}\ (\texttt{if}\ \phi\ \{u_1\})$	$\to$	`if` $\{u\}\ \phi\ \{\{u\}\ u_1\}$		(R49)
$\{u\}\ (\texttt{for}\ x\ \{u_1\})$	$\to$	`for` $x\ \{\{u\}\ u_1\}$	$(x \notin \mathrm{fv}(u))$	(R50)

u_2. This directly leads to the equivalence $u_1\,;\ u_2 \equiv u_1 \mid \{u_1\}\ u_2$ that makes it possible to eliminate sequentiality. The complete system of rewriting rules is given in Table 4.

The relation $\equiv$ from Def. 6 can be extended to Ter_{AS}, For_{AS} and Upd_{AS}:

Lemma 2. *Equivalence $\equiv$ of terms, formulas and updates is a congruence relation for all constructors given in Def. 1, 7 and 8.*

Example 2. We continue Example 1 and assume the same vocabulary/algebra.

$$\begin{aligned} a := 1\,;\ f(a) := 2 \quad &\equiv \quad a := 1 \mid f(1) := 2 \\ \mathrm{val}_{S,\beta}(a := 1\,;\ f(a) := 2) \quad &= \quad \{\langle a\rangle \mapsto 1,\ \langle f,(1)\rangle \mapsto 2\} \\ \mathrm{val}_{S,\beta}(a := 1\,;\ (a := 3 \mid f(a) := 2)) \quad &= \quad \{\langle a\rangle \mapsto 3,\ \langle f,(1)\rangle \mapsto 2\} \end{aligned}$$

We normalise the update in the second line using the given rewriting rules:

$$\begin{aligned} & a := 1\,;\ (a := 3 \mid f(a) := 2) \\ (\mathrm{R45}) \quad \rightarrow \quad & a := 1 \mid \{a := 1\}\ (a := 3 \mid f(a) := 2) \\ (\mathrm{R48}) \quad \rightarrow \quad & a := 1 \mid (\{a := 1\}\ a := 3 \mid \{a := 1\}\ f(a) := 2) \\ *(\mathrm{R47}) \quad \rightarrow \quad & a := 1 \mid (a := \{a := 1\}\ 3 \mid f(\{a := 1\}\ a) := \{a := 1\}\ 2) \\ *(\mathrm{R2}), *(\mathrm{R12}) \quad \rightarrow \quad & a := 1 \mid (a := 3 \mid f(\textsc{non-rec}(a := 1,\ a, ())) := 2) \\ (\mathrm{R11}) \quad \rightarrow \quad & a := 1 \mid (a := 3 \mid f(\text{if } \mathit{true} \text{ then } 1 \text{ else } a) := 2) \end{aligned}$$

The last expression can be simplified further using rules for conditional terms, which are, however, beyond the scope of this paper. Further, using (R54) in Table 5, it is possible to eliminate the assignment $a := 1$, which is overridden by $a := 3$.

8 Soundness and Completeness of Update Application

The following two lemmas state that the rewriting rules from Sect. 5, 6 and 7 are sound and complete. Both lemmas have been proven using the Isabelle/HOL tool. The first and more important result is that rewriting does not change the value of terms, formulas or updates, i.e., that rewriting is an equivalence transformation:

Lemma 3. *The rules of Tables 2, 3 and 4 are sound: if $\alpha \rightarrow \alpha'$ then $\alpha \equiv \alpha'$.*

The second lemma characterises the form of terms, formulas or updates to which no further rewriting rules are applicable. Knowing that some rule is applicable as long as the update application operator, substitutions, any of the "helper" constructors NON-REC, IN-DOM, REJECT, or the sequential composition operator occur in an expression ensures that no cases have been left out:

Lemma 4. *If an expression $\alpha \in Ter_{AS} \cup For_{AS} \cup Upd_{AS}$ is irreducible (up to bound renaming) concerning the rules of Tables 2, 3 and 4, then α will not contain the operators* NON-REC, IN-DOM *or* REJECT *or sequentially composed updates, i.e., $\alpha \in Ter \cup For \cup Upd$. Further, α does not contain any update applications or substitution applications.*

9 Modelling Stack and Heap Structures

The memory of imperative and object-oriented programs can be modelled as a well-ordered algebra by choosing appropriate vocabularies Σ. By updating the values of function symbols, the memory contents can then be defined and modified symbolically. Compared to a more explicit encoding of program states as individuals (for instance, elements of a datatype), directly representing memory using a first-order vocabulary leads to very readable formulas that are in particular suited for interactive proof systems. The downside of this representation is that the possibilities of meta-reasoning about the semantics of a language are limited.

In the whole section, we assume that the universe for evaluating updates are the natural numbers $\mathbb{N}$, and that the standard well-ordering $<$ is used (as in Example 1). A more realistic application would, of course, require a typed logic and to model the datatypes of programming languages properly. For this section, it shall suffice to treat both data and addresses/pointers as natural numbers.

Variables: The simplest way to store data in programs is the usage of global variables, which can be seen as constants $g, h, i, \ldots \in \Sigma$ when representing program memory using well-ordered algebras ($\alpha(g) = \alpha(h) = \cdots = 0$). Assignments are naturally performed through updates $g := t$. Expanding a sequential update into a parallel update yields a representation of the post-state by describing the post-values of all modified variables in terms of the pre-values:[8]

$$gswap \quad = \quad i := g\,;\ g := h\,;\ h := i \quad \equiv \quad g := h \mid h := g \mid i := g$$

Local Variables: Although it is never necessary to use temporary or local variables in updates, the visibility of assignments in updates can be restricted. When an update is expanded into its parallel representation, such "local variables" will no longer turn up as left-hand sides of assignments. The helper variable i that is used in the definition of *gswap*, for instance, becomes unnecessary in the parallel representation: here, the assignment $i := g$ could be removed without changing the effect of the update on the remaining variables g, h. More generally, we can use the operation REJECT (from Def. 7 in Sect. 11), which can be carried out by purely syntactic means, for hiding variables.

$$\begin{aligned} i := 3\,;\ \textsc{reject}(gswap, \overline{i := 0}) \quad &\equiv \quad i := 3\,;\ (g := h \mid h := g) \\ &\equiv \quad g := h \mid h := g \mid i := 3 \\ i := 3\,;\ \textsc{reject}(gswap, \overline{i := 0 \mid g := 0}) \quad &\equiv \quad h := g \mid i := 3 \end{aligned}$$

Effectively, the application of REJECT turns i or i, g into local variables of *gswap*. The right-hand side of the assignments $i := 0$ and $g := 0$ used in the expressions does not matter.

[8] We leave out parentheses because both parallel and sequential composition are associative, see (R52) and (R53) in Table 5.

Explicit Stack: We can also model local variables $l, m, n \ldots \in \Sigma$ more explicitly by introducing a stack. Therefore, we represent the variables as unary functions ($\alpha(l) = \alpha(m) = \cdots = 1$) and give them a stack address (a natural number) as argument. We also need a stack pointer $sp \in \Sigma$ that, in turn, is a constant ($\alpha(sp) = 0$) that is increased when entering a "procedure" and decreased when exiting:[9]

$$\begin{array}{rcl} \mathit{swap}(g,h) & = & \mathit{sp} := \mathit{sp}+1\,;\ \ l(\mathit{sp}) := g\,;\ g := h\,;\ h := l(\mathit{sp})\,;\ \ \mathit{sp} := \mathit{sp}-1 \\ & \equiv_{\mathbb{N}} & g := h \mid h := g \mid l(\mathit{sp}+1) := g \end{array}$$

Again, we might want to restrict the visibility of assignments to local variables:

$$\textsc{reject}(\mathit{swap}(g,h), \overline{\texttt{for}\ x\ \{\texttt{if}\ \mathit{sp} \dot{<} x\ \{l(x) := 0\}\}}) \quad \equiv_{\mathbb{N}} \quad g := h \mid h := g$$

The following formula characterises *swap*. Simply applying the updates will render the formula trivially valid:

$$\begin{array}{l} \forall x. \forall y.\ \{g := x \mid h := y\}\ \{\mathit{swap}(g,h)\}\ (g \doteq y \wedge h \doteq x) \\ \qquad\qquad \equiv\ \ \forall x. \forall y.\ (y \doteq y \wedge x \doteq x) \quad \equiv \quad \mathit{true} \end{array}$$

Classes and Attributes: Also the individual objects of a class can be distinguished using addresses (natural numbers). Instance attributes of a class C are then unary functions $a_C, b_C \ldots \in \Sigma$ (with $\alpha(a_C) = \alpha(b_C) = \cdots = 1$) that take an address as argument. As an example, we consider again the class *List* representing doubly-linked lists from Sect. 2 (with attributes $\mathit{next}, \mathit{prev}, \mathit{val} \in \Sigma$). The following two updates describe the setup of singleton lists (that hold a value v) and the concatenation of two lists (where one list ends with the object e and the second one begins with the object b):

$$\begin{array}{rcl} \mathit{setup}(o,v) & = & o.\mathit{prev} := \mathit{nil} \mid o.\mathit{val} := v \mid o.\mathit{next} := \mathit{nil} \\ \mathit{cat}(e,b) & = & e.\mathit{next} := b \mid b.\mathit{prev} := e \end{array}$$

(we assume that $\mathit{nil} \in \Sigma$ denotes invalid addresses and the beginning and end of lists). Lists can then be created and modified as follows: (*init* as in Sect. 2)

$$\begin{array}{rcl} \mathit{init} & \equiv & \mathit{setup}(a,c)\,;\ \mathit{setup}(b,2)\,;\ \mathit{cat}(a,b)\,;\ a.\mathit{next}.\mathit{val} := d \\ & \equiv & a.\mathit{prev} := \mathit{nil} \mid b.\mathit{next} := \mathit{nil} \mid a.\mathit{next} := b \mid b.\mathit{prev} := a \mid \\ & & a.\mathit{val} := c \mid b.\mathit{val} := d \\ \mathit{seq} & = & \texttt{for}\ x\ \{\texttt{if}\ x \dot{<} n+1\ \{\mathit{setup}(x,x)\}\}\,;\texttt{for}\ x\ \{\texttt{if}\ x \dot{<} n\ \{\mathit{cat}(x,x+1)\}\} \\ & \equiv_{\mathbb{N}} & 0.\mathit{prev} := \mathit{nil} \mid n.\mathit{next} := \mathit{nil} \mid \texttt{for}\ x\ \{\texttt{if}\ x \dot{<} n+1\ \{x.\mathit{val} := x\}\} \mid \\ & & \texttt{for}\ x\ \{\texttt{if}\ x \dot{<} n\ \{x.\mathit{next} := x+1\}\} \mid \\ & & \texttt{for}\ x\ \{\texttt{if}\ x \dot{<} n\ \{(x+1).\mathit{prev} := x\}\} \end{array}$$

[9] In this section, we write $u_1 \equiv_{\mathbb{N}} u_2$ for updates that have the same value over algebras $(\mathbb{N}, <, I)$, provided that I interprets the functions $+$, $-$ and literals as is common.

Properties about the lists can again be proven by applying the updates and performing first-order reasoning:

$$\forall x.\,(\neg x \dot{<} n \lor \{seq\}\ x.next.prev \doteq x) \quad \equiv_{\mathbb{N}} \quad \forall x.\,(\neg x \dot{<} n \lor x \doteq x) \quad \equiv \quad true$$

Object Allocation: Updates cannot add or remove individuals from a universe (*constant-domain semantics*). In modal logic, the usual way to simulate changing universes is to introduce a predicate that distinguishes between existing and non-existing individuals. Likewise, for our heap model "implicit" attributes $created_C$ can be defined that, for instance, have value 1 for existing and 0 for non-existing objects of a class C. An initial state in which no objects are allocated can be reached through the update $\texttt{for}\ x\ \{x.created_C := 0\}$. We write an allocator for list nodes as follows:[10]

$$alloc(o,v) \quad = \quad o := \min i.\,(i.created_{List} \doteq 0)\,;\ \big(o.created_{List} := 1 \mid setup(o,v)\big)$$

Note, that allocating objects in parallel using this method will produce clashes, because parallel updates cannot observe each other's effects. When running in parallel, $alloc(a,1)$ and $alloc(b,2)$ will deterministically allocate the same object:

$$alloc(a,1) \mid alloc(b,2) \quad \equiv \quad alloc(b,2)\,;\ a := b \quad \not\equiv \quad alloc(a,1)\,;\ alloc(b,2)$$

Arrays: Arrays in a Java-like language behave much like objects of classes, with the difference that arrays provide numbered cells instead of attributes. We can model arrays be introducing a binary access function $ar \in \Sigma$ and a unary function $len \in \Sigma$ telling the length of arrays ($\alpha(ar) = 2$ and $\alpha(len) = 1$). Array allocation can be treated just like allocation of objects through an implicit attribute $created_{ar}$. Given this vocabulary, we can allocate an array of length n and fill it with numbers $0, \ldots, n-1$: (we write $o[x]$ instead of $ar(o,x)$)

$$\begin{aligned} alloc_{ar}(o,n) \quad &= \quad o := \min i.\,(i.created_{ar} \doteq 0)\,;\ \big(o.created_{ar} := 1 \mid o.len := n\big) \\ seq_{ar} \quad &= \quad alloc_{ar}(o,n)\,;\ \texttt{for}\ x\ \{\texttt{if}\ x \dot{<} o.len\ \{o[x] := x\}\} \end{aligned}$$

10 Symbolic Execution in Dynamic Logic Revisited

As shown in Sect. 2, during symbolic execution, updates can represent a certain prefix (or path) of a program, whereas the suffix that remains to be executed is given in the original language. In order to use updates for symbolic execution, first of all a suitable representation of the program states using a first-order vocabulary and algebras (along the lines of Sect. 9) has to be chosen. Rewriting rules then define the semantics of program features in terms of updates and

[10] For practical purposes, it is reasonable to have more book-keeping about allocated objects than shown here. One approach is to introduce variables $nextToCreate_C$ and to allocate objects sequentially.

of connectives of first-order logic. This approach has been used to implement symbolic execution for the "real-world" language JavaCard [9]. Examples for the rewriting rules are:[11]

$$\langle\rangle\ \phi \quad \rightsquigarrow \quad \phi, \qquad\qquad \langle s = t;\ \alpha\rangle\ \phi \quad \rightsquigarrow \quad \{s := t\}\ \langle\alpha\rangle\ \phi$$
$$\langle \texttt{if}\ (b)\ \beta_1;\ \texttt{else}\ \beta_2;\ \alpha\rangle\ \phi \quad \rightsquigarrow \quad (b \wedge \langle\beta_1;\ \alpha\rangle\ \phi) \vee (\neg b \wedge \langle\beta_2;\ \alpha\rangle\ \phi)$$

It is important to note that updates are *not* intended as an intermediate representation for complete programs: the focus is on handling the sequential parts. For reasoning about general loops or recursion, techniques like induction or invariants are still necessary. It is, nevertheless, possible to translate certain loops directly to an update [10]. An example are many array operations in Java with unbounded runtime:[12]

$$\langle \texttt{System.arrayCopy}(ar_1, o_1, ar_2, o_2, n)\rangle\ \phi$$
$$\rightsquigarrow \quad \{\texttt{for}\ x\ \{\texttt{if}\ \neg x \doteq o_2 \wedge x \doteq o_2 + n\ \{ar_2[x] := ar_1[x - o_2 + o_1]\}\}\}\ \phi$$

Compared to a declarative specification of `arrayCopy` using a post-condition that contains a universally quantified formula, the imperative update can be applied to formulas or terms like a substitution. We consider updates as advantageous both for interactive and automated reasoning: the program structure is preserved, and unnecessary non-determinism in a derivation is avoided.

A characteristic of imperative programs is that memory locations can be assigned to/overwritten multiple times. After elimination of sequential composition, overwritten locations occur as clashes in updates. An example is update *init* from Sect. 2 and 9, which contains potential clashes because of aliasing: for $a \doteq b$, the expressions $a.val$ and $b.val$ denote the same location. Due to last-win semantics, it is not necessary to distinguish the possible cases when turning sequential composition into parallel composition. Only when applying the update, as in the expression co' in Sect. 2, the case $a \doteq b$ has to be handled explicitly.

Well-ordered semantics enables an implicit handling of output dependencies in loops (different iterations assign to the same locations) in a similar way [10]. A simple example is: ($e(i)$ is a side-effect free, possibly non-injective expression)

$$\langle \texttt{while}\ \ (\neg i \doteq 0)\ \ \{i = i - 1;\ a[e(i)] = i;\}\rangle\ \phi$$
$$\rightsquigarrow \quad \{i := 0 \mid \texttt{for}\ x\ \{\texttt{if}\ x \doteq i\ \{a[e(x)] := x\}\}\}\ \phi$$

11 Laws for Update Simplification

Sect. 9 demonstrates how updates can be simplified and written as parallel composition of assignments. More formally, we can extend Sect. 5 and state that,

[11] s, t, b have to be free of side-effects. It general, it will also be necessary to define a translation of side-effect free program expressions into terms.

[12] For sake of clarity, the example ignores the diverse errors that can occur when calling `arrayCopy`, for instance for $ar_1 \doteq ar_2$.

Table 5. Laws for Commuting and Distributing Update Connectives

For $\alpha \in \mathit{Ter}_{AS} \cup \mathit{For}_{AS} \cup \mathit{Upd}_{AS}$:

$$\{u_1\}\,\{u_2\}\,\alpha \equiv \{u_1\,;\,u_2\}\,\alpha \tag{R51}$$

$$u_1 \mid (u_2 \mid u_3) \equiv (u_1 \mid u_2) \mid u_3 \tag{R52}$$

$$u_1\,;\,(u_2\,;\,u_3) \equiv (u_1\,;\,u_2)\,;\,u_3 \tag{R53}$$

$$u_1 \mid u_2 \equiv \textsc{reject}(u_1, \overline{u_2}) \mid u_2 \tag{R54}$$

$$u_1 \mid u_2 \equiv u_2 \mid \textsc{reject}(u_1, \overline{u_2}) \tag{R55}$$

$$u \equiv u \mid \texttt{if}\ \phi\ \{u\} \tag{R56}$$

$$u_1 \equiv u_1 \mid \textsc{reject}(u_1, \overline{u_2}) \quad (u_2 \text{ arbitrary}) \tag{R57}$$

$$\texttt{if}\ \phi\ \{u_1 \mid u_2\} \equiv \texttt{if}\ \phi\ \{u_1\} \mid \texttt{if}\ \phi\ \{u_2\} \tag{R58}$$

$$\texttt{if}\ \phi_1\ \{\texttt{if}\ \phi_2\ \{u\}\} \equiv \texttt{if}\ \phi_1 \wedge \phi_2\ \{u\} \tag{R59}$$

$$\texttt{for}\ x\ \{\texttt{if}\ \phi\ \{u\}\} \equiv \texttt{if}\ \phi\ \{\texttt{for}\ x\ \{u\}\} \quad (x \notin \mathrm{fv}(\phi)) \tag{R60}$$

$$\texttt{for}\ x\ \{\texttt{if}\ \phi\ \{u\}\} \equiv \texttt{if}\ \exists x.\,\phi\ \{u\} \quad (x \notin \mathrm{fv}(u)) \tag{R61}$$

$$\texttt{for}\ x\ \{u_1 \mid u_2\} \equiv \texttt{for}\ x\ \{u_1\} \mid u_2 \quad (x \notin \mathrm{fv}(u_2)) \tag{R62}$$

For $u = \texttt{for}\ z\ \{\texttt{if}\ z \doteq x\ \{\{x/z\}\ u_1\}\}$ and $z \neq x$, $z \notin \mathrm{fv}(u_1)$:

$$\texttt{for}\ x\ \{u_1\} \equiv \texttt{for}\ x\ \{\textsc{reject}(u_1, \overline{u})\} \tag{R63}$$

$$\texttt{for}\ x\ \{u_1 \mid u_2\} \equiv \texttt{for}\ x\ \{u_1\} \mid \texttt{for}\ x\ \{\textsc{reject}(u_2, \overline{u})\} \tag{R64}$$

For $u = \texttt{for}\ z\ \{\texttt{if}\ z \doteq x\ \{\{x/z\}\ \texttt{for}\ y\ \{u_1\}\}\}$ and $\{x, y, z\} = 3$, $z \notin \mathrm{fv}(u_1)$:

$$\texttt{for}\ x\ \{\texttt{for}\ y\ \{u_1\}\} \equiv \texttt{for}\ y\ \{\texttt{for}\ x\ \{\textsc{reject}(u_1, \overline{u})\}\} \tag{R65}$$

given an arbitrary update u, there will always be an equivalent update $u' \equiv u$ of the following shape: (in which ϕ_i, s_i, t_i do not contain further updates)

$$\begin{array}{l} \texttt{for}\ x_{1,1}\ \{\texttt{for}\ x_{1,2}\ \{\texttt{for}\ \cdots\ \{\texttt{if}\ \phi_1\ \{s_1 := t_1\}\}\}\} \\ \mid \cdots \\ \mid \texttt{for}\ x_{k,1}\ \{\texttt{for}\ x_{k,2}\ \{\texttt{for}\ \cdots\ \{\texttt{if}\ \phi_k\ \{s_k := t_k\}\}\}\} \end{array} \tag{1}$$

It is usually advantageous to establish this shape: (i) Obvious clashes, like in the update $g := 1 \mid g := 2$, can easily be eliminated. (ii) The update can easily be read and directly tells about the values of variables or heap contents. (iii) When applying updates syntactically using the rewriting system of Sect. 5, this form is more efficient than most other shapes, because it supports the search for matching assignments. (iv) It is possible to define more specialised and efficient rewriting rules for update application (than the ones given in Sect. 5). This has been done for the implementation of updates in KeY.

Table 5 gives, besides others, identities that enable to establish form (1) by turning sequential composition into parallel composition, distributing `if` and `for` through parallel composition and commuting `if` and `for`. Another important application of the identities is the optimisation of parallel composition,

Table 6. Simplification Rules for Updates based on Neutral and Extremal Elements

`if` ϕ {`skip`}	$\to$	`skip`		(R66)	`skip` $\mid u$	$\to$	u	(R71)
`if` *false* $\{u\}$	$\to$	`skip`		(R67)	$u \mid$ `skip`	$\to$	u	(R72)
`if` *true* $\{u\}$	$\to$	u		(R68)	`skip`; u	$\to$	u	(R73)
`for` x {`skip`}	$\to$	`skip`		(R69)	u; `skip`	$\to$	u	(R74)
`for` x $\{u\}$	$\to$	u	$(x \notin \mathrm{fv}(u))$	(R70)				

which involves ordering updates ((R52), (R55)) and removing updates that are overridden by other updates ((R54), see Sect. 2). Table 6 contains a set of rewriting rules for eliminating neutral or extremal elements. The soundness of all rules and identities, based on the semantics of Sect. 4, has been proven using the Isabelle/HOL proof assistant.

Lemma 5. *The rules of Table 6 are correct: if* $\alpha \to \alpha'$ *then* $\alpha \equiv \alpha'$.

For formulating the transformation rules, we need a further operator from Def. 7: the expression $\textsc{reject}(u_1, \overline{u_2})$ denotes an update that carries out exactly those assignments of u_1 that do *not* define locations that are also assigned to by u_2. This enables us to make updates disjoint, i.e., to prevent updates from assigning to the same locations, which is often a premise for permuting updates.

12 Normalisation and Equivalence Modulo Definedness

The identities given in Sect. 11 are sufficient for turning updates into shape (1). In the implementation of updates in KeY, this kind of rewriting[13] is performed immediately whenever updates occur, and updates are stored or shown only in shape (1). Often, this is already enough for making equivalent updates syntactically equal. One of the counterexamples are the following equivalent updates that are not rewritten to the same expression:

$$\texttt{for}\ x\ \{\texttt{if}\ a \dot{<} x\ \{u\}\} \mid \texttt{for}\ x\ \{\texttt{if}\ \neg a \dot{<} x\ \{u\}\} \quad \equiv \quad \texttt{for}\ x\ \{u\}$$

Because updates can contain arbitrary terms and formulas, we cannot hope for a general procedure that decides the equivalence of two updates or that establishes a real normal form. On the other hand, reasoning about the equivalence of updates is *not more difficult* than reasoning about the equivalence of terms without updates (which can contain formulas, however, because of the constructors $\min x.\, \phi$ and if ϕ then t_1 else t_2). We describe a procedure that turns every update u into an equivalent update

$$\begin{array}{l} \texttt{for}\ x_{1,1}\ \{\texttt{for}\ x_{1,2}\ \{\texttt{for}\ \cdots\ \{f_1(x_{1,1}, x_{1,2}, \ldots) := t_1\}\}\} \\ \mid \cdots \\ \mid \texttt{for}\ x_{k,1}\ \{\texttt{for}\ x_{k,2}\ \{\texttt{for}\ \cdots\ \{f_k(x_{k,1}, x_{k,2}, \ldots) := t_k\}\}\} \end{array} \tag{2}$$

[13] Application of (R51), (R52), (R58), (R59), (R60), (R64) from left to right, Table 6 as well as ordering sequences of parallel updates.

Table 7. Compatibilities between $\equiv_{\mathrm{md}}$ and Update Operators

$f(\bar{t}) := f(\bar{t}) \equiv_{\mathrm{md}} \mathtt{skip}$	(R75)
$u_1 \equiv_{\mathrm{md}} u_1'$ implies $u_1 \mid u_2 \equiv_{\mathrm{md}} u_1' \mid u_2$	(R76)
$u_1 \equiv_{\mathrm{md}} u_1'$, $u_2 \equiv_{\mathrm{md}} u_2'$ implies $u_1 \,;\, u_2 \equiv_{\mathrm{md}} u_1' \,;\, u_2'$	(R77)
$u \equiv_{\mathrm{md}} u'$ implies $\mathtt{if}\ \phi\ \{u\} \equiv_{\mathrm{md}} \mathtt{if}\ \phi\ \{u'\}$	(R78)
$u \equiv_{\mathrm{md}} \mathtt{skip}$ implies $\mathtt{for}\ x\ \{u\} \equiv_{\mathrm{md}} \mathtt{skip}$	(R79)
For $\alpha \in \mathit{Ter}_{AS} \cup \mathit{For}_{AS} \cup \mathit{Upd}_{AS}$:	
$u \equiv_{\mathrm{md}} u'$ implies $\{u\}\,\alpha \equiv \{u'\}\,\alpha$	(R80)

where the set $\{f_1, \ldots, f_k\}$ contains all function symbols that are assigned to by u (but possibly more symbols). Establishing a normal form is then reduced to normalising the terms $t_1, \ldots, t_k$. We need a bit of equipment:

Assignments vs. Modifications: Given a well-ordered algebra $S = (U, <, I)$, there are three ways in which a partial interpretation J (for instance, the value of an update) can behave at a location $\mathit{loc} = \langle f, \bar{a} \rangle$: (i) J can be undefined at point loc (i.e., $J(\mathit{loc}) = \bot$), (ii) J can agree with the interpretation I at point loc (i.e., $J(\mathit{loc}) = I(\mathit{loc}) \neq \bot$), which means that it assigns to the location without changing the stored value, or (iii) J can assign a value to loc that is different from the value assigned by the interpretation I (i.e., $\bot \neq J(\mathit{loc}) \neq I(\mathit{loc})$). Although the behaviours (i) and (ii) mostly cannot be distinguished when working with updates, the relation $\equiv$ is fine enough for separating the two cases. For arbitrary terms, formulas or updates α, we have:

$$\{a := a\}\,\alpha \quad \equiv \quad \{\mathtt{skip}\}\,\alpha \qquad \text{but} \qquad a := a \quad \not\equiv \quad \mathtt{skip}$$

We define a coarser equivalence relation that identifies the cases (i) and (ii):

Definition 9. *Two updates $u_1, u_2 \in \mathit{Upd}_{AS}$ are called* equivalent modulo definedness*, $u_1 \equiv_{md} u_2$, if for all well-ordered algebras $S = (U, <, I)$ and all variable assignments β over S*

$$I \oplus \mathrm{val}_{S,\beta}(u_1) = I \oplus \mathrm{val}_{S,\beta}(u_2)$$

Two examples for updates that are equivalent modulo definedness are:

$$a := a \quad \equiv_{\mathrm{md}} \quad \mathtt{skip}, \quad (\mathtt{for}\ x\ \{f(x) := f(x)\} \mid f(a) := b) \quad \equiv_{\mathrm{md}} \quad f(a) := b$$

It has to be stressed, however, that $\equiv_{\mathrm{md}}$ is *not* a congruence relation for all of the update constructors. The critical constructors are parallel composition and quantification:

$$a := a \quad \equiv_{\mathrm{md}} \quad \mathtt{skip} \qquad \text{but} \qquad a := b \mid a := a \quad \not\equiv_{\mathrm{md}} \quad a := b \mid \mathtt{skip}$$

More generally, Table 7 contains a number of implications that enable to derive equivalence modulo definedness syntactically.

Normalisation of Updates: Surprisingly, the notion of equivalence modulo definedness allows to perform normalisation of updates u. By Table 7, we have:

$$v \;=\; \texttt{for}\; x_1\; \{\texttt{for}\; x_2\; \{\texttt{for}\; \cdots\; \{f(x_1, x_2, \ldots) := f(x_1, x_2, \ldots)\}\}\} \;\equiv_{\mathrm{md}}\; \texttt{skip}$$

for arbitrary function symbols f. Because equivalence modulo definedness is a congruence relation for sequential composition, assignments to f in the update u can then be "split off":

$$\begin{array}{lll} & & u \\ (\mathrm{R74}) & \equiv & u\,;\, \texttt{skip} \\ (\mathrm{R77}) & \equiv_{\mathrm{md}} & u\,;\, v \\ (\mathrm{R45}) & \equiv & u \mid \{u\}\, v \\ (\mathrm{R54}) & \equiv & \textsc{reject}(u, \overline{\{u\}\, v}) \mid \{u\}\, v \end{array}$$

Simplifying the update $\{u\}\, v$ by applying rewriting rules will turn the right-hand side of the assignment in v into an explicit representation of the values that u assigns to f. In contrast, simplifying $\textsc{reject}(u, \overline{\{u\}\, v})$ eliminates all assignments to the symbol f from the update u. By iterating the splitting procedure (or by choosing an update v that contains more assignments) the normalform (2) will eventually be established. The resulting update u' is equivalent to the original update u modulo definedness, which means that u and u' have the same effect when being applied to terms, formulas or updates (R80).

13 Related Work

Symbolic execution of programs is introduced in [11] in form of a symbolic interpreter for imperative, deterministic programs. The considered programming language only provides integer variables, although arrays are shortly mentioned. Execution states of the interpreter consist of a symbolic variable assignment (a mapping from program variables to polynomials over the initial variable contents) and a path condition (a quantifier-free formula over the initial variable contents).

There are different approaches to extend symbolic execution to heap structures and arrays, two of them are: In [12], an explicit model of the heap is maintained during execution of the program, which is extended each time a variable or attribute is accessed the first time. Eager case distinctions are performed in order to cover different initial shapes of the heap. A more implicit representation is achieved by describing the state of the heap as a formula, which, for instance, is done in [13] for separation logic. Because updates describe *heap modifications*, i.e., not necessarily the complete heap state, they can be seen as an orthogonal approach and could be combined with both methods.

A theory that is very similar to updates are abstract state machines (ASMs) [14]. While there are different versions of ASMs, all update constructors of this paper can in similar form also be found in [15]. The main difference is the notion

of "consistent updates" that exists for ASMs and that demands clash-freeness. In contrast, the present paper describes a semantics in which clashes are resolved by a last-win strategy or a well-ordering strategy, which we consider as better suited for representing imperative programs. This topic is discussed in Sect. 10 (and also in [16]): intuitively, clashes in updates are caused by multiple assignments to the same location in an imperative program. Because it is generally not decidable whether clashes occur—due to aliasing—case distinctions can be postponed by resolving clashes deterministically.

Substitutions in B [17] have character similar to updates. Like ASMs, they are used for modelling systems and are a complete programming language that also provides loops and non-determinism. Updates are deliberately kept less expressive, focussing on automated simplification and application.

The guarded command language [18] is used as intermediate language in the verification systems ESC/Java2 [19] and Boogie [20]. In contrast to updates, guarded commands are used to represent *complete* object-oriented programs—which requires concepts like loops or non-determinism—and are eliminated using wp-calculus.

Many proof assistants, for instance Isabelle/HOL [3] or PVS [21], provide notations for function updates. The main differences to the updates in the present paper is that function updates are directly attached to functions, and, thus, do not have a substitution-like character. At the same time, function updates usually provide fewer constructors and are less expressive.

Explicit substitutions [8], i.e., substitutions that are applied in multiple steps and in a delayed manner, are a refinement of λ-calculi. Explicit substitutions are a basis for programming language features like closures, but are also relevant when studying logics. The step-wise application of explicit substitution is similar to the application of updates and substitutions in the present paper. Updates go beyond explicit substitutions concerning the provided constructors, and are given an independent semantics in the style of an imperative programming language. A further difference is that updates are designed as a component of first-order logic, whereas the style in which explicit substitutions can be used to define or to modify functions appears more natural in higher-order logics.

In the context of the KeY system, updates turn up in [9] for the first time, where the only update constructor are assignments. Parallel updates are described in [16, 22] for the first time, and have the same last-win semantics as in this paper.

14 Conclusions and Future Work

The update language described in this paper has been implemented in the KeY prover. Quantified updates, added most recently, have mostly improved the ability of the prover to handle arrays, as operations like `arrayCopy` (Sect. 10) can now be specified and symbolically executed very efficiently. Compared to the rules in Sect. 5 and 11 (which are more general), KeY also contains further optimisations for applying updates that have been found to be important in practice.

An interesting step would be the combination of ordinary substitutions and updates. This would require developing a concept of bound renaming for updates. Another appealing improvement would be the possibility of non-deterministic updates, which would allow to handle object creation (or, generally, underspecification of language features) more naturally.

Acknowledgements

I want to thank Reiner Hähnle for bringing up the idea of extending the update language by adding quantification, as well as for discussions. I am also grateful for discussions and comments from Wolfgang Ahrendt and Richard Bubel, and for comments from the anonymous referees.

References

1. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
2. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY Tool. Software and System Modeling **4** (2005) 32–54
3. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
4. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
5. Zermelo, E.: Beweis dass jede Menge wohlgeordnet werden kann. Mathematische Annalen **59** (1904) 514–516
6. Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. Prentice Hall (1992)
7. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
8. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming **1** (1991) 375–416
9. Beckert, B.: A dynamic logic for the formal verification of JavaCard programs. In Attali, I., Jensen, T., eds.: Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France. Volume 2041 of LNCS., Springer (2001) 6–24
10. Gedell, T., Hähnle, R.: Automating verification of loops by parallelization. In: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNAI, Springer (2006) To appear.
11. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
12. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: Proceedings, TACAS. Volume 2619 of LNCS., Springer (2003) 553–568
13. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In Yi, K., ed.: Proceedings, APLAS. Volume 3780 of LNCS., Springer (2005) 52–68
14. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36

15. Stärk, R.F., Nanchen, S.: A logic for abstract state machines. Journal of Universal Computer Science **7** (2001) 981–1006
16. Platzer, A.: An object-oriented dynamic logic with updates. Master's thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems (2004)
17. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (1996)
18. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
19. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings, PLDI. (2002) 234–245
20. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: Post Conference Proceedings, CASSIS, Marseille. Volume 3362 of LNCS., Springer (2005) 49–69
21. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In Alur, R., Henzinger, T.A., eds.: Proceedings, CAV. Volume 1102 of LNCS., Springer (1996) 411–414
22. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In Furbach, U., Shankar, N., eds.: Proceedings, IJCAR, Seattle, USA. LNCS, Springer (2006) To appear.

Paper 3

Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic

Philipp Rümmer[1] and Muhammad Ali Shah[2]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
philipp@cs.chalmers.se
[2] Avanza Solutions ME, Dubai - 113116, United Arab Emirates
muhammad.ali@avanzasolutions.com

Abstract. Program verification is concerned with proving that a program is correct and adheres to a given specification. Testing a program, in contrast, means to search for a witness that the program is incorrect. In the present paper, we use a program logic for Java to prove the *incorrectness* of programs. We show that this approach, carried out in a sequent calculus for dynamic logic, creates a connection between calculi and proof procedures for program verification and test data generation procedures. In comparison, starting with a program logic enables to find more general and more complicated counterexamples for the correctness of programs.

1 Introduction

Testing and program verification are techniques to ensure that programs behave correctly. The two approaches start with complementary assumptions: when we try to verify correctness, we implicitly expect that a program *is* correct and want to confirm this by conducting a proof. Testing, in contrast, expects incorrectness and searches for a witness (or *counterexample* for correctness):

"Find program inputs for which something bad happens."

In the present paper, we want to reformulate this endeavour and instead write it as an existentially quantified statement:

"There are program inputs for which something bad happens." (1)

Written like this, it becomes apparent that we can see testing as a proof procedure that attempts to eliminate the quantifier in statements of form (1). When considering functional properties, many program logics that are used for verification are general enough to formalise (1), which entails that calculi for such program logics can in fact be identified as testing procedures.

The present paper discusses how the statement (1), talking about a Java program and a formal specification of safety-properties, can be formalised in

dynamic logic for Java [1, 2]. Through the usage of algebraic datatypes, this formalisation can be carried out without leaving first-order dynamic logic. Subsequently, we use a sequent calculus for automatic reasoning about the resulting formulas. The component of the calculus that is most essential in this setting is quantifier elimination. Depending on the way in which existential quantifiers are eliminated—by substituting ground terms, or by using metavariable techniques—we either obtain proof procedures that much resemble automated white-box test generation methods, or we arrive at procedures that can find more general and more complicated solutions (program inputs) of (1), but that are less efficient for "obvious" bugs. We believe that this viewpoint to incorrectness proofs can both lead to a better understanding of testing and to more powerful methods for showing that programs are incorrect.

Organisation of the Paper Sect. 2 introduces dynamic logic for Java and describes how (1) can be formalised. In Sect. 3, we show how different versions of a sequent calculus for dynamic logic can be used to reason about (1). Sect. 4 discusses how solutions of (1) can be represented. Sect. 5 discusses related work, and Sect. 6 gives future work and concludes the paper.

Running Example: Erroneous List Implementation The Java program shown in Fig. 1 is used as example in the whole paper. It is interesting for our purposes because it operates on a heap datastructure and contains unbounded loops, although it is not difficult to spot the bug in method `delete`.

2 Formalisation of the Problem in Dynamic Logic

In the scope of this paper, the only "bad things" that we want to detect are violated post-conditions of programs. Arbitrary broken safety-properties (like assertions) can be reduced to this problem, whereas the violation of liveness-properties (like looping programs) falls in a different class and the techniques presented here are not directly applicable. This section describes how the statement that we want to prove can be formulated in dynamic logic:

> There is a pre-state—possibly subject to pre-conditions—such that the program at hand violates given post-conditions. (2)

Dynamic Logic First-order dynamic logic (DL) [1] is a multi-modal extension of first-order predicate logic in which modal operators are labelled with programs. There are primarily two kinds of modal operators that are dual to each other: a diamond formula $\langle\alpha\rangle\phi$ expresses that ϕ holds in at least one final state of program α. Box formulae can be regarded as abbreviations $[\alpha]\phi \equiv \neg\langle\alpha\rangle\neg\phi$ as usual. The DL formulae that probably appear most often have the form $\phi \rightarrow \langle\alpha\rangle\psi$ and state, for a deterministic program α, the total correctness of α concerning a precondition ϕ and a postcondition ψ. In this paper, we will only use dynamic logic for Java [2] (JavaDL) and assume that α is a list of Java statements.

```
public class IntList {
  private ListNode head;
  public void add (int n) { ... }

  /*@
    @ public normal_behavior
    @ ensures !contains(n);
    @*/
  public void delete(int n) {
    ListNode cur = head, prev = head;
    while (cur != null) {
      if (cur.val == n) prev.next = cur.next;
      else              prev = cur;
      cur = cur.next;
    }
  }

  public /*@ pure @*/ boolean contains(int n) {
    ListNode temp = head;
    while (temp != null) {
      if (temp.val == n) return true;
      temp = temp.next;
    }
    return false;
  }
}
```

```
class ListNode {
  public int      val;
  public ListNode next;
}
```

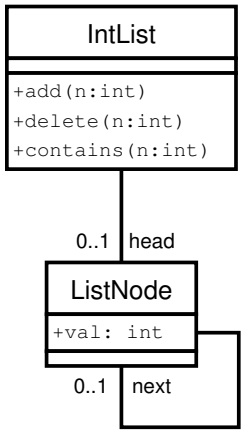


Fig. 1. The running example, a simple implementation of singly-linked lists, annotated with JML [3] constraints. We concentrate on the method `delete` for removing all elements with a certain value, which contains bugs.

Updates JavaDL features a notation for updating functions in a substitution-like style [4], which is primarily useful because it allows for a natural representation of symbolic execution. For our purposes, *updates* can be seen as a simplistic programming language and are defined by the grammar:

$$Upd \;::=\; \texttt{skip} \;\big|\; f(s_1,\ldots,s_n) := t \;\big|\; Upd\,|\,Upd \;\big|\; \texttt{if}\ \phi\ \{Upd\} \;\big|\; \texttt{for}\ x\ \{Upd\}$$

in which $s_1, \ldots, s_n, t$ range over terms, f over function symbols, ϕ over formulae and x over variables. The update constructors denote effect-less updates, assignments, parallel composition, guarded updates and quantified updates. Updates u can be attached to terms and formulae (like in $\{u\}\ t$) for changing the state in which the expression is supposed to be evaluated:

$$\{a := g(3)\}\ f(a) \;\rightsquigarrow\; f(g(3)), \quad \{a := 3 \mid \texttt{for}\ x\ \{f(x) := 2{\cdot}x{+}1\}\}\ f(f(a)) \;\rightsquigarrow\; 15$$

As shown here, it is always possible to apply updates to terms and formulae like a substitution, unless a formula contains further modal operators. In the latter case, the application has to be delayed until the modal operator is eliminated.

2.1 Heap Representation in Dynamic Logic for Java

Reasoning in JavaDL always takes place in the context of a system of Java classes, which is supposed to be free of compile-time-errors. From this context, a vocabulary of sorts and function symbols is derived that represents variables and the heap of the program in question [2].

Most importantly, objects of classes are in JavaDL identified with natural numbers. For each class C, a sort with the same name and a (injective) function $C.get : nat \to C$ are introduced. $C.get(i)$ is the ith object of class C (i is the index or "address"). For distinct classes C and D, $C.get(i)$ and $D.get(j)$ never are the same object. Each sort C representing a class also contains a distinguished individual denoted by *null*, which is used to represent undefined references. Attributes of type T of a class C are modelled by functions $C \to T$. Instead of the infix notation $attr(o)$, we will mostly write $o.attr$ for attribute accesses.

C can be seen as a reservoir containing both those objects that are already created and those that can possibly be created later by a program: JavaDL uses a constant-domain semantics in which modal operators never change the domains of existing individuals. In order to distinguish existing and non-existing objects, for each class C also a constant $C.nextToCreate : nat$ is declared that denotes the lowest index of a non-created object. All objects $C.get(i)$ with $i < C.nextToCreate$ are created, all others are not.

For the program in Fig. 1, the vocabulary is as follows:

Sorts:	Functions:	
$IntList, ListNode,$	$IntList.get$	$: nat \to IntList$
$int, nat, \ldots$	$ListNode.get$	$: nat \to ListNode$
	$IntList.nextToCreate$	$: nat$
	$ListNode.nextToCreate$	$: nat$
	$head$	$: IntList \to ListNode$
	$next$	$: ListNode \to ListNode$
	val	$: ListNode \to int$

2.2 Formalising the Violation of Post-Conditions

We go back to (2). It is almost straightforward to formalise the part of (2) that comes after the existential quantifier "there is a pre-state":

$$\neg\big(\mathit{pre\text{-}conditions} \to \langle\, \mathit{statements} \,\rangle\ \mathit{post\text{-}conditions}\big) \tag{3}$$

Formula (3) is true if and only if the pre-conditions hold, the program fragment does not terminate, or terminates and the post-conditions do not hold in the final state.

Property (2) does not mention termination, which could be interpreted in different ways. If in (3) the box operator $[\alpha]\phi$ was used instead of a diamond, we would also specify that the program has to terminate for the inputs that we search for. JavaDL does, however, not distinguish between non-termination due

to looping and abrupt termination due to exceptions (partial correctness semantics). Because we, most likely, will consider abrupt termination as a violation of the post-condition, the diamond operator appears more appropriate.

2.3 Quantification over Program States

In order to continue formalising (2), it is necessary to close the statement (3) existentially and to add quantifiers that express "there is a pre-state":

$$\exists\, \mathit{pre\text{-}state}.\ \{\mathit{pre\text{-}state}\}\ \neg\big(\mathit{pre\text{-}conditions} \rightarrow \langle\, \mathit{statements}\,\rangle\ \mathit{post\text{-}conditions}\big) \quad (4)$$

Because state quantification is not directly possible in JavaDL, we use an update $\{\mathit{pre\text{-}state}\}$ to define the state in which (3) is to be evaluated. For a Java program, the pre-state covers (i) variables that turn up in a program, and (ii) the heap that the program operates on. Following Sect. 2.1, at a first glance this turns out to be a second-order problem, because the heap is modelled by functions like *head*, *next*, etc.[3] A second glance reveals, fortunately, that a proper Java program (and proper pre- and post-conditions)[4] will only look at the values $C.\mathit{get}(i).\mathit{attr}$ of attributes for $i < C.\mathit{nextToCreate}$: the state of non-existing objects is irrelevant. Quantification of $C.\mathit{nextToCreate}$ and the finite prefix

$$C.\mathit{get}(0).\mathit{attr},\ C.\mathit{get}(1).\mathit{attr},\ \ldots,\ C.\mathit{get}(C.\mathit{nextToCreate} - 1).\mathit{attr}$$

can naturally be realised through algebraic datatypes, like through lists. Note, that the number of quantified locations is finite, but unbounded.

Attributes of Primitive Types In the most direct case, the type of *attr* would be *int* and the quantification as follows (other primitive Java types can be handled in the same way):

$$\exists\, \mathit{attr}_V : \mathit{intList}.\ \ \{\texttt{for}\ x : \mathit{nat}\ \{C.\mathit{get}(x).\mathit{attr} := \mathit{attr}_V \downarrow x\}\}\ \ldots$$

Apart from the actual quantifier, an update is used for copying the contents of the list variable attr_V to the attribute. The expression also contains an operator for accessing lists $[a_0, \ldots, a_n]$, which we define by

$$[a_0, \ldots, a_n]\downarrow i\ :=\ \begin{cases} a_i & \text{for } i \leq n \\ 0 & \text{otherwise} \end{cases} \qquad (i : \mathit{nat})$$

The fact that the operator returns a default value (0, but any other value would work equally well) for accesses outside of the list bounds simplifies the overall treatment and basically renders the length of lists irrelevant.[5]

[3] JavaDL does not provide higher-order quantification.

[4] In the whole paper, we assume that pre- and post-conditions only talk about the program state, and only about created objects.

[5] Instead of lists, one could also talk about functions with finite support.

Attributes of Reference Types The quantification is a bit more involved for attributes *attr* of type D, where D is a reference type, e.g., a class: (i) attributes can be undefined, i.e., have value *null*, (ii) attributes of created objects must not point to non-created objects, and (iii) attributes of type D can also point to objects of type D', provided that D' is a subtype of D. We capture these requirements by overloading the function $D.get$. Assuming that $D_0\,(=D), \ldots, D_k$ is an arbitrary, but fixed enumeration of D's subtypes, we define:

$$D.get(s,i) \;:=\; \begin{cases} D_s.get(i) & \text{for } i < D_s.nextToCreate,\ s \le k \\ null & \text{otherwise} \end{cases} \qquad (i, s : nat)$$

Apart from the object index i, we also pass $D.get(s,i)$ the index s of the requested subtype of D. The result of $D.get(s,i)$ is either a created object (if i and s are within their bounds $D_s.nextToCreate$ and k) or *null*. With this definition, the quantification part for a reference attribute boils down to

$$\exists\, a_S, a_V : natList.\ \ \{\texttt{for}\ x : nat\ \{C.get(x).attr := D.get(a_S \downarrow x, a_V \downarrow x)\}\}\ \ldots$$

In case of a class D that does not have proper subclasses, the list a_S can of course be left out (and the first argument of $D.get$ can be set to 0).

Example We show the formalisation of (2) for the method `delete` in the program of Fig. 1. Apart from the values of the attributes *head*, *next* and *val*, which are treated as discussed above, one also has to quantify over the number of created objects (*IntList.nextToCreate* and *ListNode.nextToCreate*), over the receiver o of the method invocation and over the argument n. o is assumed to be either an arbitrary created object or *null* ($IntList.get(0, o_V)$). The pre- and post-conditions correspond to the JML specification: initially, o is not *null*, and `delete` really removes the elements with value n.

$$\begin{aligned} &\exists\, k_{IL}, k_{LN}, o_V : nat.\ \ \exists\, n_V : int.\ \ \exists\, head_V, next_V : natList.\ \ \exists\, val_V : intList. \\ &\ \{IntList.nextToCreate := k_{IL} \mid ListNode.nextToCreate := k_{LN}\} \\ &\ \{\texttt{for}\ x : nat\ \{IntList.get(x).head := ListNode.get(0, head_V \downarrow x)\} \mid \\ &\ \ \texttt{for}\ x : nat\ \{ListNode.get(x).next := ListNode.get(0, next_V \downarrow x)\} \mid \\ &\ \ \texttt{for}\ x : nat\ \{ListNode.get(x).val := val_V \downarrow x\} \mid \\ &\ \ o := IntList.get(0, o_V) \mid n := n_V\} \\ &\quad \neg(\ o \neq null \rightarrow \langle\, o.\texttt{delete}(n)\,\rangle\ \langle\, b = o.\texttt{contains}(n)\,\rangle\ b = FALSE\ ) \end{aligned} \qquad (5)$$

3 Constructing Proofs for Program Incorrectness

A Gentzen-style sequent calculus for JavaDL is introduced in [2], which has been implemented in the KeY system [5] and is used by us as test-bed. Fig. 2 shows a small selection of the rules. Relevant for us are the following groups of rules: (i) rules for a sequent calculus for first-order predicate logic with metavariables (the first 5 rules of Fig. 2), (ii) rules that implement symbolic execution [6] for

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}\ \wedge\text{R} \qquad \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \wedge\text{L} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \neg\text{R}$$

$$\frac{\Gamma \vdash \phi[x/f(X_1, \ldots, X_n)], \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \forall\text{R} \qquad (X_1, \ldots, X_n \text{ all metavariables in } \phi)$$

$$\frac{\Gamma \vdash \phi[x/X], \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \exists\text{R} \qquad (X \text{ a fresh metavariable})$$

$$\frac{\Gamma, \{u\}\ \{r := l\}\ \langle \ldots \rangle \phi \vdash \Delta}{\Gamma, \{u\}\ \langle r = l; \ldots \rangle \phi \vdash \Delta}\ \text{ASSIGN-L} \qquad (r, l \text{ side-effect-free})$$

$$\frac{\begin{array}{c}\Gamma, \{u\}\ \langle \alpha_1; \ldots \rangle \phi, \{u\}\ b \vdash \Delta \\ \Gamma, \{u\}\ \langle \alpha_2; \ldots \rangle \phi \vdash \{u\}\ b, \Delta\end{array}}{\Gamma, \{u\}\ \langle \texttt{if}\ (b)\ \alpha_1\ \texttt{else}\ \alpha_2\ \ldots \rangle \phi \vdash \Delta}\ \text{IF-L} \qquad (b \text{ side-effect-free})$$

Fig. 2. Examples of (simplified) sequent calculus rules for JavaDL. In the last two rules, the update u can also be empty (`skip`) and disappear.

Java, and (iii) rewriting rules for applying and simplifying updates (not shown here, see [4]).

The fact that the calculus directly integrates symbolic execution—and covers all important features of Java like dynamic object creation and exceptions—is most central for us. When symbolically executing a program, the proof tree resembles the *symbolic execution tree* of the program [6] and reflects the (feasible) paths through the program. Branch predicates that describe, in terms of the pre-state, when a certain path is taken are accumulated as formulae in a sequent. JavaDL introduces such predicates for conditional statements, when unwinding loops, or for statements that might raise exceptions. A simple example is the following proof:

$$\frac{\dfrac{\dfrac{\dfrac{\vdots}{p+1 \leq 0, p \geq 0 \vdash}}{\{p := p+1\}\ \langle\rangle p \leq 0, p \geq 0 \vdash}}{\langle p = p+1; \rangle p \leq 0, p \geq 0 \vdash}\ \text{ASSIGN-L} \qquad \frac{\dfrac{\dfrac{\vdots}{-p \leq 0 \vdash p \geq 0}}{\{p := -p\}\ \langle\rangle p \leq 0 \vdash p \geq 0}}{\langle p = -p; \rangle p \leq 0 \vdash p \geq 0}}{\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p; \rangle p \leq 0 \vdash}\ \text{IF-L}$$

Symbolic execution and update application can usually be automated easily, because in each proof situation only few rules are applicable, and because the application order does not matter.

This section discusses how the sequent calculus can be used to prove formulae (4). The first and essential task is always to eliminate the existential quantifiers, i.e., to provide the programs inputs, which can be concrete or symbolic.[6]

[6] Assuming that pre- and post-conditions only talk about the program state, it is sufficient to apply $\exists$R once (and not multiple times) for each quantifier in $\exists$ *pre-state*,

We focus on and propose two methods for constructing proofs: the usage of metavariables and depth-first search (Sect. 3.2) and the usage of metavariables and backtracking-free search with constraints (Sect. 3.3). In our experiments, we have concentrated on the latter method, because the implementation KeY follows this paradigm. As a comparison, Sect. 3.1 shortly discusses how a ground calculus would handle (4), which resembles common test generation techniques.

3.1 Using a Ground Proof Procedure

The simplest approach is *ground reasoning*, i.e., to not use metavariables. Therefore, a ground version of $\exists\text{R}$ can be used: (t is a term)

$$\frac{\Gamma \vdash \phi[x/t], \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \exists\text{R}_g$$

Equivalently, also the normal rule $\exists\text{R}$ can be applied, immediately followed by a *substitution step* that replaces the introduced metavariable X with a concrete term t. For (4), the usage of rule $\exists\text{R}_g$ encompasses that a concrete pre-state has to be chosen up-front that satisfies the pre-condition and makes the program violate its post-condition. If we consider (5), for instance, we see that a proof can be conducted with the following instantiations:

k_{IL}	k_{LN}	o_V	n_V	$head_V$	$next_V$	val_V
1	1	0	5	[0]	[7]	[5]

(6)

The instantiations express that the classes *IntList* and *ListNode* have one created object each (k_{IL}, k_{LN}), that the object *IntList.get*(0) receives the method invocation (o_V) with argument 5 (n_V), that *IntList.get*(0).*head* points to the object *ListNode.get*(0) ($head_V$), that *ListNode.get*(0).*next* is *null* ($next_V$, because of $7 \geq k_{LN}$), i.e., that the receiving list has only one element, and that *ListNode.get*(0).*val* is 5 (val_V).

A ground proof of a formula (4) is the most specific description of an erroneous situation that is possible. For debugging purposes, this is both an advantage and a disadvantage: (i) it is possible to concretely follow a program execution that leads to a failure, but (ii) the description does not distinguish between those inputs (or input features) that are relevant for causing a failure and those that are irrelevant. The disadvantage can partly be undone by looking at more than one ground proof, and by searching for proofs with "minimal" input data (e.g., [7]). Technically, the main advantage of a ground proof is that program execution (and checking pre- and post-conditions) is most efficient for a concrete pre-state. The difficulty, of course, it to find the *right* pre-state, which is subject of techniques for automated test data generation. Common approaches are the generation of *random* pre-states (e.g., [7]), or the usage of backtracking, symbolic execution and constraint techniques in order to optimise coverage criteria and to reach the erroneous parts of a program (see, e.g., [8]).

because the validity of (4) only depends on the program fragment and the pre- and post-conditions, not on the values of other symbols.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{[\,P \mapsto 2\,]}}{P+1>3, P\geq 0 \vdash}}{\{p:=P+1\}\ \langle\rangle p>3, p\geq 0 \vdash}}{\{p:=P\}\ \langle p=p+1;\rangle p>3, P\geq 0 \vdash} \qquad \dfrac{\dfrac{*}{[\,P \mapsto 2\,]}}{\{p:=P\}\ \langle p=-p;\rangle p>3 \vdash\ P\geq 0}}{\{p:=P\}\ \langle \texttt{if}\ (p\geq 0)\ p=p+1;\ \texttt{else}\ p=-p;\rangle p>3 \vdash}\ \text{IF-L}}{\vdash\ \neg\{p:=P\}\ \langle \texttt{if}\ (p\geq 0)\ p=p+1;\ \texttt{else}\ p=-p;\rangle p>3,\ \ldots}\ \neg\text{R}}{\vdash\ \exists p_V : \mathit{int}.\ \{p:=p_V\}\ \neg\langle \texttt{if}\ (p\geq 0)\ p=p+1;\ \texttt{else}\ p=-p;\rangle p>3}\ \exists\text{R}$$

Fig. 3. Proof that a program violates its post-condition $p > 3$. The initial (quantified) formula is derived as described in Sect. 2. The application of updates is not explicitly shown in the proof.

3.2 Construction of Proofs using Metavariables and Backtracking

The most common technique for efficient automated proof search in tableau or sequent calculi are rigid metavariables (also called free variables) and backtracking (depth-first search), for an overview see [9]. The rules shown in Fig. 2, together with a global substitution rule that allows to substitute terms for metavariables in a proof tree, implement a corresponding sequent calculus. Because, in particular, the substitution rule is destructive and a wrong decision can hinder the subsequent proof construction, proof procedures usually carry out a depth-first search with iterative deepening and backtrack when earlier rule applications appear misleading.

The search space of a proof procedure can be seen as an and/or search tree: (i) And-nodes occur when the proof branches, for instance when applying $\wedge\text{R}$, because each of the new proof goals has to be closed at some point. (ii) Or-nodes occur when a decision has to be drawn about which rule to apply next, or about a substitution that should be applied to a proof; in general, only one of the possible steps can be taken.

Metavariables and backtracking can be used to prove formulae like (4). The central difference to the ground approach is that metavariables can be introduced as place-holders for the pre-state, which can later be refined and made concrete by applying substitutions. A simple example is shown in Fig. 3, where the initial value of the variable p is represented by a metavariable P. After symbolic execution of the program, it becomes apparent that the post-condition $p > 3$ can be violated in the left branch by substituting 2 for P. The right branch can then be closed immediately, because this path of the program is not executed for $P = 2$: the branch predicate $P \geq 0$ allows to close the branch. Generally, the composition of the substitutions that are applied to the proof can be seen as a description of the pre-state that is searched for. A major difference to the ground case is that a substitution also can describe *classes* of pre-states, because it is not necessary that concrete terms are substituted for all metavariables.

Branch Predicates Strictly speaking, the proof branching that is caused by the rule IF-L (or by similar rules for symbolic execution) falls into the "and-node" category: all paths through the program have to be treated in the proof. The situation differs, however, from the branches introduced by $\wedge$R, because IF-L performs a cut (a case distinction) on the branch predicate $\{u\}\ b$. As the program is executed with symbolic inputs (metavariables), it is possible to turn $\{u\}\ b$ into *true* or *false* (possibly into both, as one pleases), by applying substitutions and choosing the pre-state appropriately. Coercing $\{u\}\ b$ in this way will immediately close one of the two branches.

There are, consequently, two principle ways to close (each of) the proof branches after executing a conditional statement: (i) the program execution can be continued until termination, and the pre-state can be chosen so that the post-condition is violated, or (ii) one of the two branches can be closed by making the branch predicate *true* or *false*, which means that the program execution is simply forced *not* to take the represented path. Both cases can be seen in Fig. 3, in which the same substitution $P \mapsto 2$ leads to a violation of the post-condition in the left branch and turns the branch predicate in the right branch into *true*.

Proof Strategy The proof construction consists of three parts: (i) pre-conditions have to be proven, (ii) the program has to be executed symbolically in order to find violations of the post-conditions, and (iii) it has to be ensured that the program execution takes the right path by closing the remaining proof branches with the help of branch predicates. These steps can be performed in different orders, or also interleaved. Furthermore, it can in all phases be necessary to backtrack, for instance when a violation of the post-conditions was found but the pre-state does not satisfy the pre-condition, or if the path leading to the failure is not feasible.

Example Formula (5) can be proven by choosing the following values, which could be found using metavariables and backtracking:

k_{IL}	k_{LN}	o_V	n_V	$head_V$	$next_V$	val_V
1	1	0	N_V	$[0, \ldots]$	$[7, \ldots]$	$[N_V, \ldots]$

(7)

Comparing this solution to (6), the main difference is that no concrete value has to be chosen for n_V. It suffices to state that the value of n_V coincides with the first element of the list val_V: when calling `delete`, the actual parameter coincides with the first element of the receiving linked list. Likewise, the parts of the pre-state that are described by lists do not have to be determined completely: the tail of lists can be left unspecified by applying substitutions like $VAL_V \mapsto \mathrm{cons}(N_V, VAL_{tail})$ (which is written as $[N_V, \ldots]$ in the table). Sect. 4 discusses how the representation of solutions can further be generalised.

3.3 Construction of Proofs using Incremental Closure

There are alternatives to proof search based on backtracking: one idea is to work with metavariables, but to delay the actual application of substitutions to the

proof tree until a substitution has been found that closes all branches. The idea is described in [10] and worked out in detail in [11]. While backtracking-free proof search is, in principle, also possible when applying substitutions immediately, removing this destructive operation vastly simplifies proving without backtracking. Because KeY implements this technique, it is used in our experiments.

The approach of [11] works by explicitly enumerating and collecting, for each of proof goals, the substitutions that would allow to close the branch. Substitutions are represented as *constraints*, which are conjunctions of unification conditions $t_1 \equiv t_2$. A generalisation is discussed in Sect. 4. For the example in Fig. 3, the "solutions" of the left branch could be enumerated as $[\,P \equiv 2\,]$, $[\,P \equiv 1\,]$, $[\,P \equiv 0\,]$, $[\,P \equiv -1\,]$, ..., and the solutions of the right branch as $[\,P \equiv 0\,]$, $[\,P \equiv 1\,]$, $[\,P \equiv 2\,]$, ... In this case, we would observe that, for instance, the substitution represented by $[\,P \equiv 0\,]$ closes the whole proof. Generally, the conjunction of the constraints for the different branches describes the substitution that allows to close a proof (provided that it is consistent).

When proving formulae (4) using metavariables, a substitution (i.e., pre-state) has to be found that simultaneously satisfies the pre-conditions, violates the post-conditions in one (or multiple) proof branches and invalidates the branch predicates of all remaining proof branches. The constraint approach searches for such a substitution by enumerating the solutions of all three in a fair manner. In our experiments, we also used breadth-first exploration of the execution tree of programs, which simply corresponds to a fair selection of proof branches and formulae that rules are applied to. For formula (5), the method could find the same solution (7) as the backtracking approach of Sect. 3.2.

Advantages Compared to backtracking, the main benefits of the constraint approach are that duplicated rule applications (due to removed parts of the proof tree that might have to be re-constructed) are avoided, and that it is possible to search for different solutions in parallel. Because large parts of the proofs in question—the parts that involve symbolic execution—can be constructed algorithmically and do not require search, the first point is particularly significant here. The second point holds because the proof search does never commit to one particular (partial) solution by applying a substitution. Constraints also naturally lead to more powerful representations of classes of pre-states (Sect. 4).

Disadvantages Destructively applying substitutions has the effect of *propagating* decisions that are made in one proof branch to the whole proof. While this is obviously a bad strategy for wrong decisions, it is by far more efficient to *verify* a substitution that leads to a solution (by applying it to the whole proof and by closing the remaining proof branches) than to hope that the remaining branches can independently come up with a compatible constraint. In Fig. 3, after applying the substitution $[\,P \mapsto 2\,]$ that is found in the left branch, the only work left in the right branch is to identify the inequation $2 \geq 0$ as valid. Finding a common solution of $P + 1 \not> 3$ and $P \geq 0$ by enumerating partial solutions, in contrast, is more naive and less efficient. One aspect of this problem is that unification

constraints are not a suitable representation of solutions when arithmetic is involved (Sect. 4).

3.4 A Hybrid Approach: Backtracking and Incremental Closure

Backtracking and non-destructive search using constraints do not exclude each other. The constraint approach can be seen as a more fine-grained method for generating substitution candidates: while the pure backtracking approach always looks at a single goal when deriving substitutions, constraints allow to compare the solutions that have been found for multiple goals. The number of goals that can simultaneously be closed by one substitution, for instance, can be considered as a measure for how reasonable the substitution is. Once a good substitution candidate has been identified, it can also be applied to the proof destructively and the proof search can continue focussing on this solution candidate. Because the substitution could, nevertheless, be misleading, backtracking might be necessary at a later point. Such hybrid proof strategies have not yet been developed or tested, to the best of our knowledge.

4 Representation of Solutions: Constraint Languages

In Sect. 3.2 and 3.3, classes of pre-states are represented as substitutions or unification constraints. These representations are well-suited for pure first-order problems [11], but they are not very appropriate for integers (or natural numbers) that are common in Java: (i) Syntactic unification does not treat interpreted functions like $+$, $-$ or literals in special way. This rules out too many constraints, for instance $[\, X + 1 \equiv 2 \,]$, as inconsistent. (ii) Unification conditions $t_1 \equiv t_2$ cannot describe simple classes of solutions that occur frequently, for instance classes that can be described by linear conditions like $X \geq 0$.[7]

The constraint approach of Sect. 3.3 is not restricted to unification constraints: we can see constraints in a more semantic way and essentially use any sub-language of predicate logic (also in the presence of theories like arithmetic) that is closed under the connective $\wedge$ as constraint language. For practical purposes, validity should be decidable in the language, although this is not strictly necessary. The language that we started using in our experiments is a combination of unification conditions (seen as equations) and linear arithmetic:

$$C \;::=\; C \wedge C \;\big|\; t_{int} = t_{int} \;\big|\; t_{int} \neq t_{int} \;\big|\; t_{int} < t_{int} \;\big|\; t_{int} \leq t_{int} \;\big|\; t_{oth} = t_{oth}$$

in which t_{int} ranges over terms of type *int* and t_{oth} over terms of other types. The constraints are given the normal model-theoretic semantics of first-order formulae (see, for instance, [10]):

[7] Depending on the representation of integers or natural numbers, certain inequations like $X \geq 1 \Leftrightarrow X \equiv \mathrm{succ}(X')$ might expressible, but this concept is rather restricted.

Definition 1. *A linear constraint C is called* consistent *if (i) for each arithmetic structure (interpreting the symbols $+$, $-$, $\neq$, $<$, $\leq$ and literals as is common over the integers, and all other function symbols arbitrarily), (ii) there is an assignment of values to metavariables such that C is evaluated to tt.*

We are in the process of working out details of this language—so far, we do not know whether consistency of constraints is decidable. Using a prototypical implementation of the constraints in KeY (as part of the constraint approach of Sect. 3.3), it is possible to find the following solution of (5) automatically:

k_{IL}	k_{LN}	o_V	n_V	$head_V$	$next_V$	val_V	$K_{IL} > 0 \wedge$
K_{IL}	K_{LN}	0	N_V	$[0, \ldots]$	$[E, \ldots]$	$[N_V, \ldots]$	$K_{LN} > 0 \wedge$
							$E \geq K_{LN}$

Compared to (7), this description of pre-states is more general and no longer contains the precise number of involved objects of *IntList* and *ListNode*. It is enough if at least one object of each class is created ($K_{IL} > 0$, $K_{LN} > 0$). Further, the solution states that $IntList.get(0)$ receives the invocation of `delete` with arbitrary argument N_V, that $IntList.get(0).head$ points to the object $ListNode.get(0)$, that the attribute $ListNode.get(0).next$ is *null* ($E \geq K_{LN}$), i.e., the receiving list has only one element, and that the value of this element coincides with N_V.

5 Related Work

Proof strategies based on metavariables and backtracking are related to common approaches to test data generation with symbolic execution, see, e.g., [6, 8]. Conceiving the approach as *proving* provides a semantics, but also opens up for new optimisations like backtracking-free proof search. Likewise, linear arithmetic is frequently used to handle branch predicates in symbolic execution, e.g. [12]. This is related to Sect. 4, although constraints are in the present paper not only used for branch predicates, but also for the actual pre- and post-conditions.

As discussed in Sect. 3.1, there is a close relation between ground proof procedures and test data generation using actual program execution.

A technique that can be used both for proving programs correct and incorrect is abstract-refinement model checking (e.g., [13–15]). Here, the typical setup is to abstract from precise data flow and to prove an abstract version of a program correct. If this attempt fails, usually symbolic execution is used to extract a precise witness for program incorrectness or to increase the precision of the employed abstraction. Apart from abstraction, a difference to the method presented here is the strong correlation between paths in a program (reachability) and counterexamples in model checking. In contrast, our approach can potentially produce classes of pre-states that cover multiple execution paths.

Related to this approach is the general idea of extracting information from failing verification attempts, which can be found in many places. ESC/Java2 [16] and Boogie [17] are verification systems for object-oriented languages that use the prover Simplify as back-end. Simplify is able to derive counterexamples from failed prove attempts, which are subsequently used to create warnings

about possible erroneous behaviour of a program for certain concrete situations. Another example is [18], where counterexamples are created from unclosed sequent calculus proofs. Making use of failing proof attempts has the advantage of reusing work towards verification that has already been performed, which makes it particularly attractive for interactive verification systems. At the same time, it is difficult to obtain completeness results and to guarantee that proofs explicitly "fail", or that counterexamples can be extracted. In this sense, our approach is more systematic.

6 Conclusions and Future Work

The development of the proposed method and of its prototypical implementation has been driven by working with (small) examples [19], but we cannot claim to have a sufficient number of benchmarks and comparisons to other approaches yet. It is motivating, however, that our method can handle erroneous programs like in Fig. 1 (and similar programs operating on lists) automatically, which we found to be beyond the capabilities of commercial test data generation tools like JTest [20, 19]. This supports the expectation that the usage of a theorem prover for finding bugs (i) is most reasonable for "hard" bugs that are only revealed when running a program with a non-trivial pre-state, and (ii) has the further main advantage of deriving more general (classes of) counterexamples than testing methods. The method is probably most useful when combined with other techniques, for instance with test generation approaches that can find "obvious" bugs more efficiently.

For the time being, we consider it as most important to better understand the constraint language of Sect. 4 for representing solutions, and, in particular, to investigate the decidability of consistency. Because of the extensive use of lists in Sect. 2.3, it would also be attractive to have constraints that directly support the theory of lists. Such constraints would introduce a notion of *heap isomorphism*, which is a topic that we also plan to address. Further, we want to investigate the combination of backtracking and incremental closure (as sketched in Sect. 3.4). A planned topic that conceptually goes beyond the method of the present paper are proofs about the termination behaviour of programs.

References

1. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
2. Beckert, B.: A dynamic logic for the formal verification of JavaCard programs. In Attali, I., Jensen, T., eds.: Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France. Volume 2041 of LNCS., Springer (2001) 6–24
3. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C.: JML Reference Manual. (2002)
4. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNAI, Springer (2006) To appear.

5. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY Tool. Software and System Modeling **4** (2005) 32–54
6. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
7. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35** (2000) 268–279
8. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the Second Conference on Computer Science and Engineering in Linkping, ECSEL (1999) 21–28
9. Hähnle, R.: Tableaux and related methods. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier Science B.V. (2001) 101–178
10. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
11. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, IJCAR, Siena, Italy. Volume 2083 of LNAI., Springer (2001) 545–560
12. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. In: SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (1998) 231–244
13. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. International Journal on Software Tools for Technology Transfer **2** (2000) 410–425
14. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10** (2003) 203–232
15. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings, PLDI. (2001) 203–213
16. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings, PLDI. (2002) 234–245
17. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: Post Conference Proceedings, CASSIS, Marseille. Volume 3362 of LNCS., Springer (2005) 49–69
18. Reif, W., Schellhorn, G., Thums, A.: Flaw detection in formal specifications. In: Proceedings, IJCAR, Siena, Italy. LNAI, Springer (2001) 642–657
19. Shah, M.A.: Generating counterexamples for Java dynamic logic. Master's thesis (2005)
20. Parasoft: JTest (2006) `www.parasoft.com/jsp/products/home.jsp?product=Jtest`.