Thesis for the Degree of Doctor of Philosophy

Calculi for Program Incorrectness and Arithmetic

Philipp Rümmer

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2008

Calculi for Program Incorrectness and Arithmetic
Philipp Rümmer
ISBN 978-91-628-7242-7

Technical Report no. 50D
Department of Computer Science and Engineering
Research Group: Formal Methods

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, 2008

Abstract

This thesis is about the development and usage of deductive methods in two main areas: (i) the deductive dis-verification of programs, i.e., how techniques for deductive verification of programs can be used to detect program defects, and (ii) reasoning modulo integer arithmetic, i.e., how to prove the validity (and, in special cases, satisfiability) of first-order formulae that involve integer arithmetic.

The areas of program verification and of testing are traditionally considered as complementary: the former searches for a formal proof of program correctness, while the latter searches for witnesses of program incorrectness. Nevertheless, deductive verification methods can discover bugs indirectly: the failure to prove the absence of bugs is interpreted as a sign for the incorrectness of the program. This approach is bound to produce "false positives" and bugs can be reported also for correct programs. To overcome this problem, I investigate how techniques that are normally used for verification can be used to directly prove the incorrectness of programs. This covers both the detection of partial incorrectness (a program produces results that are inconsistent with a declarative specification), and the detection of total incorrectness (a program diverges erroneously).

As a prerequisite for both program correctness and incorrectness proofs, I investigate and extend the concept of updates, 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. Further, I formulate a calculus for integer arithmetic that is tailored to program verification. While restricted to ground problems, the calculus can handle both linear and nonlinear arithmetic (to some degree) and is useful both for automated and interactive reasoning.

The calculus for integer arithmetic can naturally be generalised to a stand-alone procedure for Presburger arithmetic with uninterpreted predicates, which is a logic that subsumes both Presburger arithmetic and first-order logic. The procedure has similarities both with SMT-solvers and with the methods used in automated first-order theorem provers. It is complete for theorems of first-order logic, decides Presburger arithmetic, and is complete for a substantial fragment of the combination of both.

Acknowledgements

There are many people that contributed in one way or another to this thesis. First of all, I want to thank my supervisor Wolfgang Ahrendt for his advice and guidance, without which the thesis would not have been possible, and for the uncountable hours of discussions. Also my examiner Reiner Hähnle helped with many discussions and comments to solve problems or to recognise their true nature. I am also grateful to the further members of my advisory committee, Graham Kemp and Mary Sheeran, and to Koen Claessen for providing feedback on my work and on the thesis.

It was a great experience to work with the people that coauthored articles in the thesis: the master students Muhammad Ali Shah and Helga Velroyen, as well as Wolfgang Ahrendt, Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov, Steffen Schlager, and Peter H. Schmitt. Likewise, I am thankful to all other members of the KeY project, in particular to Richard Bubel and Angela Wallenburg, for creating a fantastic group to work in.

I also want to thank all PhD students and the employees at the department of Computer Science and Engineering that established such a warm and pleasant working environment, that helped me to get started in a new country, and that constantly strived to discover and get everyone into new and exciting spare time activities.

Finally, I am grateful to my parents and my family for supporting me over many years and preparing me for Computing Science, and to Cigdem for always being there and helping me to stay away from Computing Science once in a while.

Table of Contents

Introduction

Advances in hardware technology over the last decades have turned computers from expensive, large, and cryptic machines into small, highly specialised, and nearly ubiquitous tools or accessories that are an unavoidable part of daily life. An aspect of this development is an increasing demand for software programs that become more complex, more safety or security critical, and that are expected to be produced in shorter time and with less effort. Surprisingly and despite this development, the prevalent methods to produce software have changed only little over the last 30 years.

One approach to improve software quality is to analyse programs with the help of deductive and formal methods. Deduction is the principle of rigorously deriving conclusions from assumptions by means of syntactic arguments (also called rules). While originally devised in the areas of mathematics and philosophy, deduction has, again due the spread of computers, become a universal (and often automatic) tool that is applied in various areas to analyse systems and to manage information. This thesis is about the development and usage of deductive methods in two areas:

– *Deductive dis-verification of programs:* How can techniques for deductive verification of programs be used to detect program defects? This covers both the detection of partial incorrectness, i.e., the case that a program produces results that are inconsistent with a declarative specification, and the detection of total incorrectness, i.e., the situation that a program erroneously diverges.
– *Reasoning modulo integer arithmetic:* The problem of proving the validity (and, in special cases, satisfiability) of first-order formulae that involve integer arithmetic is investigated. The two examined settings are the case of quantifier-free linear and nonlinear arithmetic, and the case of Presburger arithmetic augmented with uninterpreted predicates (which subsumes both Presburger arithmetic and first-order logic).

Both topics are closely related: on the one hand, integers are the most common datatype in programs, and any deductive verification method has to offer a solution to reasoning about integers. On the other hand, we approach program dis-verification and reasoning about integers using techniques that are very similar in nature: in both cases, we start from tableau-style theorem proving with free variables and constraints.

This thesis is a collection of seven papers that were presented at conferences and workshops on deductive methods and their application. While each of the papers is a self-contained document, the thesis starts with a broader introduction to the field.

Contributions of the Thesis

Deductive dis-verification of programs. The areas of (deductive) program verification and of testing are traditionally considered as complementary: the former works under the hypothesis of program correctness and searches for a formal proof, while the latter assumes program incorrectness and searches for concrete witnesses. In the context of software development, the more realistic hypothesis is that of program incorrectness, and the usefulness of a tool primarily depends on its ability to discover program defects.

Deductive verification methods normally discover bugs indirectly: the failure to prove the absence of bugs is interpreted as a sign for the incorrectness of the program. Due to the inherent incompleteness of deductive methods, this approach is bound to produce "false positives," i.e., bugs can be suspected also in correct programs. A large number of false positives can make it nearly impossible to identify the actual defects and is commonly considered as one of the main obstacles preventing a broad usage of deductive methods in software development.

To overcome this problem, I investigate how techniques that are normally used for verification can be used to prove the (partial or total) incorrectness of programs. Because the presence of bugs is actually proven in this approach, no false positives can occur (but, vice versa, the incompleteness of the method makes it in general impossible to find all bugs in a program). The usage of symbolic reasoning allows to derive whole classes of inputs simultaneously for which a program behaves incorrectly, or to detect bugs like non-termination that are not accessible for testing.

As a prerequisite for both program correctness and incorrectness proofs, I investigate and extend the concept of *updates*, which is the central component for performing symbolic execution in Java dynamic logic ("Sequential, Parallel, and Quantified Updates of First-Order Structures," page 115). 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.

Theorem proving modulo integer arithmetic. A second ingredient for reasoning about program correctness and incorrectness is the treatment of integer arithmetic. Automatic verification systems that use SMT-solvers as back-end typically handle arithmetic with the help of integer linear programming techniques (possibly extended with axioms about simple properties of multiplication like commutativity and distributivity) and cannot be applied even to simple programs that involve nonlinear arithmetic. The paper "A Sequent Calculus for Integer Arithmetic with Counterexample Generation" (page 149) formulates a calculus for integer arithmetic that is tailored to program verification. While restricted to ground problems, the calculus can handle both linear and nonlinear arithmetic (to some degree) and is useful both for automated and interactive reasoning.

In the paper "A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic" (page 173), I develop an approach to theorem proving modulo linear integer arithmetic that is an alternative to that of SMT-solvers. The problem, in both cases, is to handle a logic in which validity is not a semi-decidable property. SMT-solvers approach this issue by starting with the (decidable) ground problem and augment the reasoning with heuristics to add quantifiers. The result are powerful reasoners, but there is no simple description of the fragment on which completeness is achieved, and there are simple examples of quantified problems where all heuristics fail.

The alternative approach described here is to start with a simple and idealised calculus for Presburger arithmetic with arbitrary uninterpreted predicates. To prove the validity of a formula in this logic, the calculus iteratively computes approximating formulae in Presburger arithmetic, which in turn can be checked using an arbitrary decision procedure for Presburger arithmetic. The result is a calculus that is complete for a rather regular fragment of Presburger arithmetic with predicates. It is easy to reason about the fragment and to show that it subsumes, e.g., both the universal and the existential fragment of the logic, as well as the whole of first-order logic.

As the next step, the idealised calculus is optimised to achieve greater efficiency without changing the set of provable formulae. A major step to this end is to add capabilities for more efficient reasoning about ground problems (similar to the approach in "A Sequent Calculus for Integer Arithmetic with Counterexample Generation," page 149), which brings the calculus closer to the architecture of SMT-solvers.

Background

1 First-Order Theorem Proving and Integer Arithmetic

Because all chapters of this thesis are written in the context of classical first-order logic, and most of them in the context of the theory of integer arithmetic, we start with an introduction into these topics.

1.1 First-Order Logic (FOL)

The definition of classical logics consists of three parts: the definition of a language as the *syntax* in which assumptions or conjectures are stated; a *semantics* that gives meaning to the elements of this language; and *calculi* that allow to reason on the syntactic level in a manner that is faithful to the semantics.

Classical first-order logic (FOL, also called predicate calculus, see [1] for a more detailed introduction) is an example for such a logic. FOL is an extension of propositional logic that allows to talk about *individuals* or *objects*, which are syntactically represented by *terms*. Quantifiers allow to state properties that are supposed to hold for all or for some individuals. A simple version of the syntax of FOL is defined by the following grammar for formulae:

$$\phi \;::=\; \phi \land \phi \;\big|\; \phi \lor \phi \;\big|\; \neg\phi \;\big|\; \forall x.\phi \;\big|\; \exists x.\phi \;\big|\; p(x, \ldots, x)$$

in which x ranges over a predefined and fixed set of *variables* and p over a fixed vocabulary of *predicates*. The first three constructors $\land$ (and), $\lor$ (or), and $\neg$ (negation) are the connectives that are also present in propositional logic, while the quantifiers $\forall$ (for all), $\exists$ (there exists), and the application of predicates are specific to FOL. A common further propositional connective is implication $\to$, but in classical logic the formula $\phi \to \psi$ can be considered as an abbreviation of $\neg\phi \lor \psi$.

As an example, the following formulae are the first five axioms of Tarski's first-order axiomatisation of Euclidean geometry [2]:

$$\forall x.\forall y.\; cg(x, y, y, x) \tag{Ta1}$$

$$\forall x.\forall y.\forall z.\; \big(cg(x, y, z, z) \to x \doteq y\big) \tag{Ta2}$$

$$\forall x.\forall y.\forall z.\forall u.\forall v.\forall w.\; \big(cg(x, y, z, u) \land cg(x, y, v, w) \to cg(z, u, v, w)\big) \tag{Ta3}$$

$$\forall x.\forall y.\; \big(bw(x, y, x) \to x \doteq y\big) \tag{Ta4}$$

$$\forall x.\forall y.\forall z.\forall u.\forall v.\; \begin{pmatrix} bw(x, u, z) \land bw(y, v, z) \\ \to \exists a.\; (bw(u, a, y) \land bw(v, a, x)) \end{pmatrix} \tag{Ta5}$$

The predicate cg represents *congruence*: $cg(a, b, c, d)$ means that the distance between the points a and b is the same as the distance between c and d. The predicate bw represents *betweenness*: $bw(a, b, c)$ means that the point b is on the line segment *ac*. We also need the binary predicate $\doteq$ to represent *equality* between two points.

It is usually required (and the case for the five formulae from above) that axioms are *closed*, which means that every occurrence of a variable x in such a formula has to be in the scope of a quantifier $\forall x$ or $\exists x$.

Functions. Traditionally, it is common to understand the notions of *equality* and *functions* not as first-class members in predicate calculus, but rather as special cases of predicates $\doteq$, f that satisfy (explicitly stated) axioms. In the case of functions, the required properties are *totality* and *functionality* (or, vice versa, the *congruence* property of equality). A unary function can be represented as a binary predicate f with the help of the following axioms (similarly for functions of higher arity):

$$\forall x.\exists y.\ f(x,y), \qquad \forall x,y,y'.\ (\neg f(x,y) \vee \neg f(x,y') \vee y \doteq y')$$

In the classical exposition of tableaux by Smullyan [3], for instance, functions are not treated at all. Due to the importance of the theory of functions and equality for applications and as more focus is put on automatic deduction, this conception of FOL has shifted: modern approaches (like theorem proving based on ordered paramodulation, see [4] for an overview) often consider functions and equality as the primary concepts of FOL, whereas predicates can be introduced as boolean-valued functions if needed.

Semantics. The semantics for FOL that is most frequently used nowadays is model-theoretic and goes back to Tarski [5]. Evaluation of formulae in this semantics is defined over *structures* (U, I) that consist of a non-empty set U, the *universe* of individuals, and an *interpretation* I that maps every predicate symbol to a subset of U^* (the set of U-tuples) on which the predicate is considered to hold. Given a structure (U, I), an *evaluation function* $val_{(U,I)}$ can then recursively be defined that maps every formula to one of the boolean values *tt* or *ff*. In the first case, (U, I) is called a *model* of the formula.

Formulae that evaluate to *tt* for *every* structure are called *valid*, while formulae that evaluate to *tt* for *some* structure are called *satisfiable.* To determine whether a given formula belongs to one of these two classes is usually considered the most important problem of reasoning in a logic: many other questions can be reduced to the question whether a certain formula is valid or satisfiable (and what a model for the formula is). It is not necessary to actually refer to the (model-theoretic) semantics of a logic in order to check validity or satisfiability: the by far more common approach is to reason on the syntactic level with the help of calculi. The correctness of a calculus, in turn, has to be justified using the semantics. In fact, the idea of calculi is much older than the concept of semantics and goes back as far as Aristotle's syllogisms.

It is well-known that the validity of a first-order formula is not a decidable problem, although the valid formulae in FOL are recursively enumerable (which implies that the satisfiable formulae are not even recursively enumerable). FOL is therefore strictly more expressive than propositional logic, in which validity is 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 and is one of the most popular logics for applications (although often in combination with various theories). A good overview of state-of-the-art FOL reasoners is the annual CADE ATP System Competition.[1]

Theories. Working with pure FOL can be too uncomfortable, too inefficient, or simply insufficient because FOL is not expressive enough: to apply the logic it is often necessary to have further concepts or datatypes like lists, arrays, ordering relations, integer or rational numbers, etc. available. Also functions and equality can be counted as such *theories* (see [6] for an overview).

More formally, a theory is a satisfiable (finite or infinite) set $\mathcal{T}$ of closed formulae, which are called the *axioms* of the theory. The theory $\mathcal{A}$ of (non-extensional) arrays [7], for instance, is defined by the following axioms:

$$\forall a, x, y.\ \mathit{select}(\mathit{update}(a, x, y), x) \doteq y,$$
$$\forall a, x_1, x_2, y.\ \big(x_1 \doteq x_2 \lor \mathit{select}(\mathit{update}(a, x_1, y), x_2) \doteq \mathit{select}(a, x_2)\big)$$

Given a theory $\mathcal{T}$, we define a *$\mathcal{T}$-structure* to be a structure (U, I) in which each element of $\mathcal{T}$ evaluates to tt. Adapting the definitions from above, formulae that evaluate to tt for every $\mathcal{T}$-structure are called *$\mathcal{T}$-valid*, while formulae that evaluate to tt for some $\mathcal{T}$-structure are called *$\mathcal{T}$-satisfiable*. As an example, the following formula is $\mathcal{A}$-valid (valid in the theory of arrays), but it is not valid because there are non-$\mathcal{A}$-structures in which it does not hold:

$$\mathit{update}(a, x, y) \doteq a \to \mathit{select}(a, x) \doteq y$$

If a theory $\mathcal{T}$ is finite, then it is in principle possible to reason about the $\mathcal{T}$-validity of a formula ϕ by examining the validity of the implication $\bigwedge \mathcal{T} \to \phi$. Because this approach is often too inefficient, however, much research is put into the design of dedicated theory-reasoners. Efficient reasoning in the presence of theories is considered as one of the major challenges of the field.

A theory that is particularly interesting (and exceptionally intricate) is the theory of integer arithmetic with the operations 0, *succ*, $+$, $\cdot$, $\doteq$, $\dot{\leq}$. The most commonly used axioms for this theory are due to Peano [8]. The first-order version of the Peano axiomatisation is incomplete, however, in the sense that there are formulae ϕ (that do not contain any operations apart from 0, *succ*, $+$, $\cdot$, $\doteq$, $\dot{\leq}$, variables, and quantifiers) for which neither ϕ nor $\neg\phi$ are implied by the Peano axioms [9]. In fact, the situation is even worse: the famous first incompleteness theorem by Gödel [10] states that there is no complete recursively enumerable (and consistent) axiomatisation of the integers in FOL.

[1] `http://www.cs.miami.edu/~tptp/CASC/`

A much weaker (in fact, decidable) system is the theory of integer arithmetic without multiplication, which is known as *Presburger arithmetic* (PA) [11] and discussed in more detail in Sect. 1.3. There is no finite axiomatisation of Presburger arithmetic, which is the reason why the notion of PA-structures (U, I) is usually defined semantically (informally) by postulating that the universe $U = \mathbb{Z}$ of such a structure are the integers, and that the operations 0, *succ*, $+$, $\doteq$, $\dot{\leq}$ are interpreted by I in the "canonical" way on $\mathbb{Z}$.

Syntactic methods. There are two main concepts to determine syntactically whether a formula in FOL (or equivalently in propositional logic) is valid: one is to reason about a set of formulae that were assumed to hold (i.e., about a *conjunction* of formulae) and to perform inferences in order to *synthesise* further formulae from these assumptions; the other one is to *analyse* the structure of a formula by repeatedly performing case distinctions such that each case becomes simpler than the whole problem (i.e., to analyse by generating a *disjunction*). Traditional calculi realise rather pure versions of these two concepts:

– *Resolution* [12] operates on a set of clauses (formulae in normal form) and works by deriving new clauses until eventually the empty clause and thus a contradiction (unsatisfiability of the clause set) can be derived.
– *Tableaux* [13] are trees that are constructed by analysing and taking apart a formula. In order to show that the analysed formula is unsatisfiable, the tree has to be expanded to a point at which an obvious contradiction occurs on every branch of the tree.

In the last years it has been generally recognised, however, that the two techniques are complementary and have to be combined to obtain powerful calculi [14–18]. The first technique is usually more successful for problems in pure FOL (without additional theories) and can handle quantifiers, functions, and equality in a natural manner. The second technique yields more efficient procedures for problems with a complex propositional structure and for quantifier-free problems modulo various theories. Because the work in this thesis primarily builds on tableaux, we give an introduction to them in Sect. 1.2 (see [13] for a more detailed exposition).

Relationship to this thesis. All papers in this thesis build on FOL as base logic. In the paper "A Sequent Calculus for Integer Arithmetic with Counterexample Generation" (page 149), we introduce a calculus for quantifier-free reasoning in the theory of integer arithmetic (both linear and nonlinear) that is designed for program verification systems. This calculus is developed further in the paper "A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic" (page 173) to a calculus for full FOL (including quantifiers) combined with the theory of Presburger arithmetic.

1.2 Tableaux and Sequent Calculi

Tableau-like calculi were first introduced by Gentzen [19] in the form of *sequent calculi*, a kind of calculi that has remained in use until today and is applicable to a large variety of logics. Most chapters of this thesis use sequent calculi as the format for proofs. Proofs in such a calculus are trees (growing upwards) in which each node is labelled with a *sequent* $\Gamma \vdash \Delta$ consisting of two lists Γ, Δ of formulae. Furthermore, each node that is not a leaf has to be labelled with a *proof rule* that determines how the parent sequents (the *premisses*) are related with the child sequent (the *conclusion*). While original versions of the calculus provided explicit rules for rearranging and contracting formulae in Γ, Δ, it has since then become more common to consider the two collections of formulae as sets right away. An example of a proof in sequent calculus is given in Fig. 1.

Gentzen's calculus was reformulated independently by Beth [20] as *semantic tableaux* and by Hintikka [21] as *model sets*, although their notations did not become successful (the name *tableau* stems from the representation of proofs as tables that was introduced by Beth). The version of tableaux that is almost exclusively used nowadays was introduced by Smullyan [3]: a tableau can be understood as a tree (usually growing downwards) in which each node is labelled with a signed formula, i.e., with a formula preceded by T or F to indicate whether the formula is negated (there are also unsigned versions of the calculus). Proof rules are represented in *unifying notation* that comprises the categories α, β for non-splitting and splitting propositional rules and γ, δ for universal and existential quantifier instantiation.

Tableaux differ in one main point from sequent calculi: while each node in a sequent calculus proof can be interpreted independently from all other nodes because it repeats assumptions and conjectures, in a tableau the formulae of a whole branch (the path between the proof root and a leaf) are available for inferences. This makes the sequent notation somewhat more flexible when formulating calculi for non-classical logics, for instance modal logics. While the tableau representation is more popular in the area of automated theorem proving, interactive proof assistants are more often based on sequent calculi.

DPLL. A calculus for propositional logic that is strongly related to tableaux is the Davis-Putnam-Logemann-Loveland (DPLL) procedure [22] that forms the basis of most propositional theorem provers (SAT-solvers). DPLL is analytic and follows the approach of analysing formulae (in clause normal form) through a case analysis. As the main difference between propositional tableaux and DPLL, the only rule that causes proof splitting in the latter calculus is the *cut-rule*, which splits over the cases that an atomic formula evaluates to *tt* or to *ff* (the principle of bivalence). On each branch, DPLL simplifies formulae by performing unit resolution steps.[2]

DPLL has recently become a popular basis to build automated reasoners for FOL and various theories (SMT-solvers): in the DPLL(T) architecture [27] the DPLL method to handle propositional problems is combined with decision procedures for ground problems in theories like equality, uninterpreted functions,

[2] The cut-rule is also central in Gentzen's sequent calculus, although its importance for avoiding redundancy in proofs was only recognised much later [23]. Both the cut-rule and formula simplification can be carried over to tableaux and are crucial from the efficiency point of view, e.g., [24–26].

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{*}{cg(b,a,a,b) \;\vdash\; cg(a,b,a,b), cg(b,a,a,b)} \qquad \cdots
          }{cg(b,a,a,b) \;\vdash\; cg(a,b,a,b), cg(b,a,a,b) \wedge cg(b,a,a,b)}\ \text{AND-RIGHT}
          \qquad
          \dfrac{*}{cg(b,a,a,b), cg(a,b,a,b) \;\vdash\; cg(a,b,a,b)}
        }{\ldots, cg(b,a,a,b), cg(b,a,a,b) \wedge cg(b,a,a,b) \rightarrow cg(a,b,a,b) \;\vdash\; cg(a,b,a,b)}\ \text{IMP-LEFT}
      }{(\mathrm{Ta1}), (\mathrm{Ta3}), cg(b,a,a,b) \;\vdash\; cg(a,b,a,b)}\ \text{ALL-LEFT} \times 6 \text{ on } (\mathrm{Ta3})
    }{(\mathrm{Ta1}), (\mathrm{Ta3}) \;\vdash\; cg(a,b,a,b), \ldots}\ \text{ALL-LEFT} \times 2 \text{ on } (\mathrm{Ta1})
  }{(\mathrm{Ta1}), (\mathrm{Ta3}) \;\vdash\; \forall y. cg(a,y,a,y), \ldots}\ \text{ALL-RIGHT}
}{(\mathrm{Ta1}), (\mathrm{Ta3}) \;\vdash\; \forall x. \forall y. cg(x,y,x,y)}\ \text{ALL-RIGHT}
$$

Fig. 1. Proof in a sequent calculus for FOL: the geometry axioms of Sect. 1.1 imply the formula $\forall x, y.\ cg(x,y,x,y)$.

integers, etc. The resulting provers are currently among the most efficient decision procedures for quantifier-free FOL and can also be combined with heuristics to treat (simple cases of) quantified formulae [28, 29]. For an overview of state-of-the-art SMT-solvers, see [30].

Quantifiers. Handling quantified formulae is the primary problem when building FOL theorem provers: to show the validity of a formula like $(\forall x.\phi) \rightarrow \psi$, it is necessary to discover suitable instances $\phi[x/t_1]$, $\phi[x/t_2]$, ... of the quantified part so that ψ is entailed. This issue is more difficult to handle in tableaux than in resolution (intuitively, because universal quantifiers distribute through conjunctions but not through disjunctions), which is why quantifier handling in tableau calculi is often inspired by or based on resolution.

Three main techniques to treat quantifiers in tableaux are:

– *ground approaches* (also called *instance-based*), which work by generating instances $\phi[x/t]$ that are added to the problem so that reasoning is reduced to the quantifier-free case. Because quantifier-free reasoning is usually very efficient, this technique can offer a good performance even if a large number of unnecessary instances is generated. The terms t to produce instances can be determined using heuristics (this is mostly common in SMT-solvers, e.g., [28, 29]), by a complete enumeration of all terms up to some redundancy [31–33], or in various ways using unification [34–36].
– *free-variable approaches* [37] resemble instance-based methods, with the difference that variables are used as placeholder symbols to generate instances $\phi[x/X_1]$, $\phi[x/X_2]$, ... The terms that are denoted by the placeholders X_1, X_2, ... are at a later point determined using unification [37] and either substituted into the proof or remembered using constraints (a more detailed description is given below). The name "free variable" for a placeholder is mostly used in the tableau community, in other areas the term "metavariable" is more common and denotes the same concept.
– *quantifier elimination* (QE) is possible for certain theories, including Presburger arithmetic [11] and real-closed fields [38], which means that for every formula in these theories there is an equivalent quantifier-free formula (in the mentioned cases, this formula can also be computed effectively). While QE methods are mostly popular for interactive proof assistants (e.g., [39–41]) and less for automated theorem provers, also the SMT-solver Yices [42] and the tableau calculus described in [43] make use of QE.

Free variables and constraints. The standard approach to discover the terms that are denoted by free variables is to use unification for finding substitution candidates, apply such candidates speculatively to a proof, and possibly backtrack at a later point to undo substitutions that appear misleading. An example is the proof attempt (1) in Fig. 2, in which the variable X is used as a place-holder for the witness that is needed to prove the existentially quantified formula. 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,

$$\dfrac{\dfrac{\vdash\ X = c,\ X = d}{\vdash\ X = c \lor X = d}\ \textsc{or-right} \qquad \vdash\ f(c) = f(X)}{\dfrac{\vdash\ (X = c \lor X = d) \land f(c) = f(X),\ \ldots}{\vdash\ \exists x.\ ((x = c \lor x = d) \land f(c) = f(x))}\ \textsc{ex-right}}\ \textsc{and-right} \tag{1}$$

$$\dfrac{\dfrac{\dfrac{[X \equiv c],\ [X \equiv d]}{\vdash\ X = c,\ X = d}}{\vdash\ X = c \lor X = d}\ \textsc{or-right} \qquad \dfrac{[f(c) \equiv f(X)]}{\vdash\ f(c) = f(X)}}{\dfrac{\vdash\ (X = c \lor X = d) \land f(c) = f(X),\ \ldots}{\vdash\ \exists x.\ ((x = c \lor x = d) \land f(c) = f(x))}\ \textsc{ex-right}}\ \textsc{and-right} \tag{2}$$

Fig. 2. Two example proofs in a sequent calculus with free variables

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 variables and instances of the quantified formula.

In [1, 44], 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 [44] show that it can also be a basis for realising automated state-of-the-art theorem provers.

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, as shown in (2) in Fig. 2. 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 $X \equiv c \land f(c) \equiv f(X)$ is consistent and is solved by the substitution (the *unifier*) $\{X \mapsto c\}$.

Relationship to this thesis. Most chapters of this thesis use sequent calculi combined with free variables and constraints to reason in various first-order logics. We generalise the solution described in [44] to other kinds of constraints to handle the theory of integer arithmetic more efficiently: in the papers “Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic” (page 39) and “Non-Termination Checking for Imperative Programs” (page 61), unification constraints modulo linear arithmetic are used; in “A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic” (page 173), constraints are arbitrary formulae in Presburger arithmetic. The latter paper also uses quantifier elimination in Presburger arithmetic.

1.3 Reasoning about Presburger Integer Arithmetic (PA)

Due to its convenient properties and the omnipresence of integers in computer programs, quantifier-free linear integer arithmetic (LIA, [45]) is widely used for applications and supported by virtually all SMT-solvers. The dominant decision procedure for LIA in such solvers is the Simplex method, which has to be combined with branch-and-bound or cutting-plane methods to ensure completeness over the integers (some SMT-solvers also use the Fourier-Motzkin method, see [46] for such a solver and [47] for an overview). Support for quantifiers is in this setting (normally) only provided by the general heuristic instantiation methods of SMT-solvers.

In contrast, support for full first-order linear integer arithmetic (which includes quantifiers and is known as *Presburger arithmetic*) (PA) is mostly present in interactive theorem provers. Two possible reasons for this are: (i) the worst-case complexity of decision procedures for PA is at least doubly exponential [48], and the worst-case complexity of quantifier elimination is triply exponential [49], which is often considered as prohibitive for automated reasoners; (ii) pure PA is too weak for many applications and has to be combined with uninterpreted functions or predicates. Adding only a single uninterpreted unary predicate to PA is enough, however, to create a logic in which valid formulae are no longer recursively enumerable [50]. There has recently been renewed interest in using (decidable) extensions of Presburger arithmetic for program verification [51, 52].

The languages of terms and formulae in Presburger arithmetic can be defined by the following grammar:

$$\begin{aligned} t &::= \alpha \bigm| x \bigm| \alpha t + \cdots + \alpha t \\ \phi &::= \phi \wedge \phi \bigm| \phi \vee \phi \bigm| \neg\phi \bigm| \forall x.\phi \bigm| \exists x.\phi \bigm| t \doteq t \bigm| t \mathrel{\dot{\leq}} t \end{aligned}$$

where x ranges over variables and α over integer literals. Compared to the grammar for FOL in Sect. 1.1, the only allowed predicates are equality $\doteq$ and the ordering $\dot{\leq}$ on integers. Besides variables, the term language of PA also provides integer literals and linear combinations of terms, but no multiplication of variables. Formulae are always evaluated over the universe $\mathbb{Z}$ of integers (there is a corresponding logic PNA of Presburger arithmetic over the natural numbers, which has similar properties as the integer version).

The first proof of decidability of PA was given by Presburger [11] in the form of a QE procedure, which was later improved by Cooper [53] and is one of the standard decision procedures for PA. A second common QE procedure that was originally developed for compiler optimisation purposes is the Omega test [54]. The latter approach is based on the Fourier-Motzkin elimination method [45] and, thus, requires that formulae are put into disjunctive normal form before elimination is possible. This implies that the worst-case complexity of the Omega test is non-elementary (every quantifier alternation can lead to an exponential blowup), while the complexity of Cooper's method is triply exponential [55]. Nevertheless, the Omega test is a popular method to decide PA, and the (little) experimental data that is available indicates that the practical runtime of both approaches is comparable [56, 57] (in the latter paper, Cooper's method is compared with Hodes' procedure, which resembles the Fourier-Motzkin method on rationals).

The Omega test. Quantifier elimination is usually formulated for the special case of a formula $\exists x.\phi$ that only contains a single quantifier $\exists x$ (i.e., no further quantifiers occur in ϕ). Given such a formula, a QE procedure determines an equivalent quantifier-free formula ψ. This is sufficient to eliminate the quantifiers in arbitrary formulae in the considered logic: in general, quantifiers can be treated one by one, starting with the innermost quantifiers. Universal quantifiers can first be turned into existential ones using the equivalence $\forall x.\phi \Leftrightarrow \neg\exists x.\neg\phi$. Because it is normally easy to decide closed quantifier-free formulae, a QE procedure practically gives rise to a decision procedure.

It is possible (but less efficient) to consider an even more restricted case, namely that $\exists x.\phi$ is a quantified conjunction of literals. To handle general formulae, the matrix of an innermost quantifier is then first turned into disjunctive normal form and the quantifier is distributed over all disjuncts. In the case of PA, further assumptions can be made: because negated equations can be turned into disjunctions of inequalities, it can be assumed that ϕ only contains inequalities and positive equations (we ignore the issue of divisibility statements for sake of brevity). This is the way in which the Omega test works: the central transformation step of the test is the elimination of the existential quantifier in a formula

$$\exists x.\ (L(x) \wedge U(x) \wedge E(x))$$

where $L(x) = \bigwedge_i a_i \dot{\leq} \alpha_i x$ is a conjunction of lower bounds, $U(x) = \bigwedge_j \beta_j x \dot{\leq} b_i$ is a conjunction of upper bounds, and $E(x) = \bigwedge_k \gamma_k x \doteq c_k$ is a conjunction of equations (all coefficients α_i, β_j, γ_k are positive). This consists of two subproblems: (i) the equations $E(x)$ have to be eliminated, which can be done, e.g., using the algorithm described in [58, Chapter 4.5.2], and (ii) the existential quantifier has to be eliminated from the remaining formula over inequalities.

The Fourier-Motzkin elimination method [45] provides a solution for the latter problem over the rationals. In this case, the following equivalence holds:

$$\exists x.\ (L(x) \wedge U(x)) \quad \Leftrightarrow \quad \bigwedge_{i,j} a_i\beta_j \dot{\leq} \alpha_i b_j \tag{3}$$

Over the integers, the implication $\Rightarrow$ is still true, but $\Leftarrow$ is violated: a counterexample is the formula $\exists x.\ (1 \dot{\leq} 2x \wedge 2x \dot{\leq} 1)$ that is not implied by $2 \dot{\leq} 2$. The main contribution of the Omega test is a version of (3) that also works over the integers through an additional case analysis over the border cases, see page 167 for the exact theorem.

Relationship to this thesis. The paper "A Sequent Calculus for Integer Arithmetic with Counterexample Generation" (page 149) describes a sequent calculus that covers quantifier-free Presburger arithmetic as well as non-linear integer arithmetic (incompletely). In the paper "A Constraint Sequent Calculus for

First-Order Logic with Linear Integer Arithmetic" (page 173), a sequent calculus is introduced that decides Presburger arithmetic and can also handle arbitrary predicate symbols (again, in an incomplete manner). Both calculi are partially based on the Omega test. The calculus given in the latter paper is also complete for LIA* [52] (but not a decision procedure).

2 Program Analysis and Deductive Program Verification

The main application of FOL and theorem proving in this thesis is the deductive analysis of programs, with the particular goal to detect program defects. We focus on *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. As a second choice, we investigate mostly *object-oriented* programs, which on the one hand means that programs can store data as a graph, the *heap*, and on the other hand that the language conceptually 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 [59]. We do not consider issues like concurrency, so that the treated fragment of Java mostly corresponds to the JavaCard language [60].

2.1 Approaches to Find Bugs in Programs

In the context of software development, the primary usage of program analysis techniques is to reveal bugs: unfinished software is with high likelihood incorrect, and any technique to discover bugs can be of great help for a developer. The following paragraphs give a short overview of existing approaches that do not require human interaction. An experimental comparison of related tools is given in [61].

Ill-formed programs: The most basic step is to ensure that a piece of code actually is a well-formed program according to some language specification. This is done by *syntactic* and *type-based* analyses that are part of interpreters and compilers for programming languages (see, e.g., [62] for an overview) and normally has a very low complexity.

Unsafe programs: There is a variety of safety properties that are commonly not considered as part of functional correctness, although they are undecidable and their verification is in general not simpler than full functional correctness. *Safety* means that it is supposed to be guaranteed that a program never performs undesired, harmful, or illegal operations during runtime, which can include (i) absence of dereferentiation of undefined pointers or accesses to data-structures outside of their bounds, (ii) absence of arithmetic errors (like division by zero), (iii) correct usage of the functionality that is provided by libraries or frameworks, (iv) correct usage of concurrency, (v) bounded usage of resources, and (vi) secure handling of information. Methods to detect such defects are:

- *Abstract interpretation-based techniques*, which are often known or represented as type systems, data-flow analysis, or constraint-based analyses [63, 64]. Such techniques derive safety properties by approximating the set of possible program states at the various points in a program.
- *Software model checking*, which proves the safety of a program by completely exploring the set of reachable program states [65–70]. Because software programs often have a large or even infinite state space, model checking is usually combined with abstract interpretation in order to reduce the number of states. In case of an incorrect program, model checking is able to produce concrete examples (in terms of the program inputs) that demonstrate the incorrectness.
- *Heuristic methods* search for patterns in program code that indicate the presence of bugs. Such methods are often integrated in compilers, but there are also stand-alone tools such as FindBugs [71] or JLint [72].
- *Deductive verification and testing*, see below.

Functionally incorrect programs: Going beyond "generic" defects as described in the previous point, it is also possible to investigate whether a program is correct wrt. a given functional specification. Such a functional specification can in principle be as complex as the program itself, which entails that verification is more difficult than for generic properties. The methods mentioned so far can to some degree be used to find violations of specifications, but are often too weak and have to be supplemented with more expensive approaches such as *deductive verification* and *testing* (which are introduced in more detail on the next pages).

It can be observed that the many of the described techniques work indirectly by verifying the *correctness* of a program; in case of an incorrect program, the inability to conduct this verification leads to information about the cause of the failure. Because verification techniques are usually incomplete and unable to verify all correct programs (either because the considered properties are not semi-decidable, or to achieve a better performance), this can lead to *false positives*: bugs can be reported even if the program in question actually is correct.

Relationship to this thesis. The papers "Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic" (page 39) and "Non-Termination Checking for Imperative Programs" (page 61) of this thesis discuss how deductive verification can be used to find bugs in programs. The particular approach described in the papers implements a direct search for bugs and never generates false positives.

2.2 Semantics and Analysis of Programs

The behaviour of an imperative program can be investigated on different levels of abstraction. 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* [73].

A second view on the semantics of programs is *operational semantics.* Describing the operational meaning of an imperative programming language essentially means to write 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.

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 FOL (see Sect. 1.1), extended with algebraic theories like integers 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) [74] or the Object Constraint Language (OCL) [75]. For reasoning about programs and specifications, this is mostly irrelevant. How first-order logic is used in specifications is illustrated in the paper "Verifying Object-Oriented Programs with KeY: A Tutorial" (page 81) in this thesis.

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, for instance about termination or exception-freeness. Such properties are one of the main application areas for deduction-based verification systems and software model checkers.

Relationship to this thesis. 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 paper "Sequential, Parallel, and Quantified Updates of First-Order Structures" (page 115) discusses the topic of capturing the operational semantics of an imperative language as rules of dynamic logic.

2.3 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 back-end. 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 defect (in the program or in the specification). Unfortunately, if the verification of a program fails, the reason is not necessarily the presence of a bug: because the correctness of programs is undecidable (and not even semi-decidable), sound verification methods are incomplete and can fail even for correct programs.

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 to capture 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. Examples in which deep embeddings *are* used for verification are the deep embedding of the Java virtual machine in ACL2 [76], the LOOP tool [77], and the EVT tool for verifying Erlang programs [78] (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) [79], which is a backwards analysis but still very near to the operational semantics.[3] 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 [80], Boogie/Spec# [81], Jack [82], and Why/Krakatoa [83, 84].

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) [85]. 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 [85], 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.

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 often chosen to be *true*) and finally produces a strongest post-condition. Because the values of variables are stored explicitly during SE, however, it is also possible to use symbolic execution for deriving weakest pre-conditions in a natural manner.

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., [86]) or test data generation (see [87] for a survey). 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 [88], which exist for many imperative languages. Examples of verification systems that are based on Hoare logics for Java are Jive [89] and the system developed as part of Bali [90]. A further program logic is dynamic logic [91], which strongly

[3] 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.

resembles Hoare logics and is described in more detail in the paper "Verifying Object-Oriented Programs with KeY: A Tutorial" (page 81) in this thesis. 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[4] 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. 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 heaps in a first-order logic are:

– Because a heap has the property of being unbounded, but finite, it can be modelled through algebraic datatypes like lists, arrays [93, 7], or through more specialised types. This approach is used in ESC/Java2 [80], Boogie [81], Krakatoa [84], and KIV [94].
– 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 [82] and KeY [95], but also the memory model of separation logic [96] falls into this category.

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, which is convenient for interactive verification.

Relationship to this thesis. Several chapters in the thesis use dynamic logic to reason about programs. In "Sequential, Parallel, and Quantified Updates of First-Order Structures" (page 115), a formalism for symbolic execution as well as

[4] An optimisation of the wp-calculus that leads to a similar effect is described in [92].

heap representation and modification in dynamic logic is developed. In this setting, symbolic execution is used as a method to compute weakest pre-conditions. The paper also proposes to use a more general representation of the symbolic program state in order to handle certain kinds of loops (more details are given in [97]).

A method to circumvent the limitation of not being able to quantify over program states is described in the paper "Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic" (page 39). Conceptually, the paper shows how the first and the second approach to heap representation can be related using updates.

2.4 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. [98, 99]). Also the generation of random program inputs is common (e.g. [100]).

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 [87].

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 exceptions or reaches violated assertions, a specification is reintroduced through the back door.

Relationship to this thesis. The papers "Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic" (page 39) and "Non-Termination Checking for Imperative Programs" (page 61) of this thesis discuss how deductive verification (based on dynamic logic) can be used to find bugs in programs. Depending on the proof strategy that is used, both specification-based and implementation-based testing can be simulated. Deductive methods also allow to find classes of program inputs that reveal bugs instead of only concrete program inputs, or to find bugs like non-termination that are inaccessible for testing approaches.

Conclusions

In the course of the PhD work presented in this thesis, deduction systems for recognising incorrectness in software programs and for deriving theorems in integer arithmetic have been developed. The method investigated in the first field has some similarities with testing, but is based on symbolic reasoning and allows to generate more general counterexamples for a larger class of possible defects than normal testing approaches do. Like in testing, no false positives are produced. The method is fully automatic and has been designed with a tight integration into development environments in mind, where a checker could run in the background and provide online error messages, similarly to existing tools for extended static checking. While the feasibility of the approach has been demonstrated, it is planned to improve and optimise its prototypical implementation to enable such an integration.

As the theory of integer arithmetic is particularly important when analysing programs, a sequent calculus for ground problems in arithmetic was developed that can be integrated in software verification systems. This calculus can naturally be generalised to a stand-alone procedure for Presburger arithmetic with uninterpreted predicates. The procedure has similarities both with SMT-solvers and with automated first-order theorem provers, but it can be shown to be complete for a larger and more regular fragment of the accepted logic than SMT-solvers. It is an ongoing task to find and eliminate efficiency problems in the calculus. In order to obtain more experimental data, it is also planned to integrate the (prototypical) implementation of the procedure as a prover back-end into systems such as KeY and Boogie.

All calculi in the thesis are based on the tableau approach (in the notation of Gentzen-style sequent calculi), which is augmented with incremental closure when necessary to handle quantifiers. The idea of incremental closure has been extended, compared to earlier work, by integrating more powerful constraint languages that allow to treat arithmetic more efficiently. It seems obvious that this approach can be generalised and be investigated independently of the particular constraint language and theory, which is planned as future work.

Overview of the Papers

The following pages summarise the papers that are part of the thesis. In addition, my contributions to papers of which I am not the sole author are listed. Each of the papers has been peer-reviewed and accepted to a conference or workshop.

Program Incorrectness Analysis

Paper 1: Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic 39
Philipp Rümmer, Muhammad Ali Shah

We show how Java dynamic logic can be used to prove the *incorrectness* of programs. To this end, we use the concept of quantified updates together with existential quantification over algebraic datatypes. We show that the 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 is in parts based on the Master's thesis [101] of Muhammad Ali Shah, which was supervised by the author. The paper has appeared in the proceedings of the First International Conference on Tests and Proofs, Zurich, Switzerland, 2007 [102]. The version in this thesis contains minor modifications.

My Contributions: I developed the main ideas to characterise and verify incorrectness in dynamic logic. My coauthor Muhammad Ali Shah did most of the implementation on top of the KeY system and evaluated the approach on examples (including the example in the paper). The writing of the paper was almost completely done by me.

Paper 2: Non-Termination Checking for Imperative Programs 61
Helga Velroyen, Philipp Rümmer

Building on the techniques from paper 1, we present an approach to automatic non-termination checking for Java programs that relates to termination checking in the same way as symbolic testing does to program verification. Our method is based on the automated generation of invariants that show that a program cannot reach terminating states when given certain program inputs. The existence of such critical inputs is shown using constraint-solving techniques. We fully implemented the method on top of the KeY system, the implementation is available for download. We also give an empirical evaluation of the approach using a collection of non-terminating example programs.

The only non-termination checker for imperative programs that is comparable to our work (to the best of our knowledge) is described in [103] and was developed independently and simultaneously to our system.

This paper is in parts based on the Master's thesis [104] of Helga Velroyen, which was supervised by the author. The paper has appeared in the proceedings of the Second International Conference on Tests and Proofs, Prato, Italy, 2008 [105]. The version in this thesis contains only minor modifications.

My Contributions: Most ideas for proving non-termination were developed in discussions with my coauthor Helga Velroyen, details for generating invariants and the invariant generator tool were worked out by Helga Velroyen. The interface and proof strategies for KeY were written by me. Helga Velroyen collected the examples and performed the experiments. I wrote Sect. 2, 3, and 6 of the paper, Helga Velroyen wrote Sect. 4 and 5, and Sect. 1 and 7 were jointly written.

Technical background: Proving and Architecture in KeY

Following a small case study, the paper gives an introduction to the logic and usage of the KeY system. The tutorial aims to fill the gap between elementary introductions using toy examples and state-of-art case studies by going through a self-contained, yet non-trivial, example.

The paper is an updated version of the paper that appeared in the post-proceedings of the 5th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands, 2006 [106]. It was presented by Wolfgang Ahrendt and me in the form of a tutorial at the symposium.

My Contributions: I developed the calendar application used as case study and did all proofs with KeY that are described in the paper. Sect. 3 and parts of Sect. 4.3 and 5.2 are written by me.

This paper describes the concept of *updates*, which is the central artifact 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. All results and the complete theory of updates have been formalised and proven using the Isabelle/HOL proof assistant [40].

This paper is an extended version of the paper that appeared in the proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Phnom Penh, Cambodia, 2006 [107].

Paper 5: The KeY System (Deduction Component) 141
Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov, Philipp Rümmer, Steffen Schlager, and Peter H. Schmitt

We give an overview of the theorem prover that is at the heart of the KeY system: the logic that the prover accepts as input language, the proof-confluent free-variable sequent calculus that is used to handle first-order logic, the notation available to specify inference rules (both pre- and user-defined), and reasoning capabilities in linear and nonlinear integer arithmetic. The deduction component is complemented with a frontend to translate specifications given in OCL or JML into the logic used by the prover.

This system abstract has appeared in the proceedings of the 21st International Conference on Automated Deduction, Bremen, Germany, 2007 [108]. The version in this thesis contains only minor modifications.

My Contributions: I wrote the section on arithmetic handling and also implemented the described parts of KeY.

Reasoning modulo Integer Arithmetic

Paper 6: A Sequent Calculus for Integer Arithmetic with Counterexample Generation 149
Philipp Rümmer

I introduce a calculus for handling integer arithmetic in first-order logic. The method is tailored to Java program verification and meant to be used both as a supporting procedure and simplifier during interactive verification and as an automated tool for discharging (ground) proof obligations. There are four main components: a complete procedure for linear equations, a complete procedure for linear inequalities, an incomplete procedure for nonlinear (polynomial) equations, and an incomplete procedure for nonlinear inequalities. The calculus is complete for the generation of counterexamples for invalid ground formula in integer arithmetic. All parts described here have been implemented as part of the KeY verification system.

This paper is an extended version of the paper that has appeared in the proceedings of the 4th International Verification Workshop at CADE 21, Bremen, Germany, 2007 [109].

I introduce a sequent calculus that combines ideas from free-variable constraint tableaux with the Omega quantifier elimination procedure. The calculus is complete for theorems of first-order logic (without functions, but with arbitrary uninterpreted predicates), can decide Presburger arithmetic, and is complete for a substantial fragment of the combination of both. In this sense, the calculus handles a similar logic as SMT-solvers that are commonly used in program verification, but is complete for a larger and more regular fragment of the logic. A prototypical implementation of the calculus is available, whose performance is evaluated on benchmarks.

This paper is an extended version of a paper accepted at the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Doha, Qatar, 2008 [110].

Summary of the Contributions

1. I develop a general concept to combine verification in dynamic logic with constraint solving to discover program states with certain properties. This was instantiated to an automatic search for:
   - violations of partial-correctness specifications in programs (page 39), and for
   - divergence of programs (page 61).
2. As part of 1, an approach to loop invariant synthesis is given that targets non-termination proofs. The approach is based on refinement and invariant templates (page 61).
3. I give a foundation of the update concept in dynamic logic, including different semantics and a complete formalisation in Isabelle/HOL. The considered notion of updates extends earlier work by including guards and quantification of updates (page 115).
4. I formulate a calculus for reasoning about ground problems in linear and nonlinear integer arithmetic that targets program verification (page 149).
5. I develop a novel approach for reasoning about first-order problems in linear integer arithmetic with uninterpreted predicates. Results about soundness, completeness, proof confluence, and refinements are derived (page 173).

Further Work Relevant for this Thesis

- The architecture and usage of the KeY tool[5] was introduced in two published articles, one of which is based on a case study (page 81 and 141).
- The approaches 1 and 2 were implemented and evaluated using benchmarks (page 39 and 61).[6]
- The version of updates considered in 3 and the calculus developed in 4 were implemented in the KeY tool and are part of the standard KeY distribution.
- The calculus developed in 5 was implemented as a standalone theorem prover[7] and evaluated using benchmarks (page 173).

[5] `http://www.key-project.org`
[6] `http://www.key-project.org/nonTermination/`
[7] `http://www.cse.chalmers.se/~philipp/princess`

References

1. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
2. Tarski, A.: What is elementary geometry? The Axiomatic Method, with Special Reference to Geometry and Physics (1959)
3. Smullyan, R.M.: First-Order Logic. Second corrected edn. Dover Publications, New York (1995) First published 1968 by Springer-Verlag.
4. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier Science B.V. (2001) 371–443
5. Tarski, A.: Pojęcie prawdy w językach nauk dedukcyjnych (the concept of truth in the languages of the deductive sciences). Prace Towarzystwa Naukowego Warszawskiego, Wydzial III Nauk Matematyczno-Fizycznych **34** (1933) 13–172
6. Beckert, B.: Equality and other theories. In D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer, Dordrecht (1999) 197–254
7. McCarthy, J.: Towards a mathematical science of computation. In Popplewell, C.M., ed.: Information Processing 1962: Proceedings IFIP Congress 62, Amsterdam, North Holland (1963) 21–28
8. Peano, G.: Arithmetices Principia, Nova Methodo Exposita. Fratres Bocca, Turin (1889) English translation, "The Principles of Arithmetic, presented by a new method," in [111], pages 20–55.
9. Paris, J., Harrington, L.A.: A mathematical incompleteness in Peano arithmetic. In Barwise, J., ed.: Handbook of Mathematical Logic. North-Holland, Amsterdam (1977) 1133–1142
10. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatschefte für Mathematik und Physik **38** (1931) 173–198 English translation, "On Formally Undecidable Propositions of Principia Mathematica and Related Systems," in [111].
11. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. (On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation). In: Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929, Warsaw, Poland (1930) 92–101,395
12. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier Science B.V. (2001) 19–99
13. D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer, Dordrecht (1999)
14. Weidenbach, C.: Combining superposition, sorts and splitting. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume II. Elsevier Science B.V. (2001) 1965–2013
15. Riazanov, A., Voronkov, A.: Splitting without backtracking. In Nebel, B., ed.: Proceedings, 17th International Joint Conference on Artificial Intelligence, Seattle, Washington, Morgan Kaufmann (2001) 611–617
16. Lynch, C., Tran, D.K.: SMELS: satisfiability modulo equality with lazy superposition. In: Proceedings, 6th International Symposium on Automated Technology for Verification and Analysis. LNCS, Springer (2008) To appear.

17. de Moura, L., Bjørner, N.: Engineering DPLL(T) + saturation. In: Proceedings, Fourth International Joint Conference on Automated Reasoning, Sydney, Australia. Volume 5195 of LNCS, Springer (2008) 410–425
18. Fietzke, A., Weidenbach, C.: Labelled splitting. In: Proceedings, Fourth International Joint Conference on Automated Reasoning, Sydney, Australia. Volume 5195 of LNCS, Springer (2008) 459–474
19. Gentzen, G.: Untersuchungen über das Logische Schliessen. Mathematische Zeitschrift **39** (1935) 176–210, 405–431 English translation, "Investigations into Logical Deduction," in [112].
20. Beth, E.W.: Semantic entailment and formal derivability. Mededelingen van de Koninklijke Nederlandse Akademie van Wetenschappen, Afdeling Letterkunde, N.R. **18** (1955) 309–342 Partially reprinted in [113].
21. Hintikka, K.J.J.: Form and content in quantification theory. Acta Philosohica Fennica **8** (1955) 7–55
22. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
23. D'Agostino, M.: Tableaux methods for classical propositional logic. In D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer, Dordrecht (1999) 45–123
24. Mondadori, M.: Classical analytical deduction. Annali dell' Università di Ferrara, Nuova Serie, sezione III, Filosofia, discussion paper, n. 1, Università degli Studi di Ferrara (1988)
25. Mondadori, M.: Classical analytical deduction, part II. Annali dell' Università di Ferrara, Nuova Serie, sezione III, Filosofia, discussion paper, n. 5, Università degli Studi di Ferrara (1989)
26. Massacci, F.: Simplification: A general constraint propagation technique for propositional and modal tableaux. In de Swart, H., ed.: Proceedings, International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oosterwijk, The Netherlands. Volume 1397 of LNCS, Springer (1998) 217–232
27. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM **53** (2006) 937–977
28. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52** (2005) 365–473
29. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In Pfenning, F., ed.: Proceedings, International Conference on Automated Deduction, Bremen, Germany. Volume 4603 of LNCS, Springer (2007) 167–182
30. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smt-lib.org` (2008)
31. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 201–215
32. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper tableaux. In Alferes, J.J., Pereira, L.M., Orlowska, E., eds.: Proceedings, European Workshop on Logics in Artificial Intelligence. LNCS, Springer (1996) 1–17
33. Baumgartner, P., Furbach, U., Pelzer, B.: Hyper tableaux with equality. In Pfenning, F., ed.: Proceedings, International Conference on Automated Deduction, Bremen, Germany. Volume 4603 of LNCS, Springer (2007) 492–507
34. Lee, S.J., Plaisted, D.A.: Eliminating duplication with the hyper-linking strategy. Journal of Automated Reasoning **9** (1992) 25–42

35. Billon, J.P.: The disconnection method: a confluent integration of unification in the analytic framework. In Miglioli, P., Moscato, U., Mundici, D., Ornaghi, M., eds.: Proceedings, 5th International Workshop on Theorem Proving with Tableaux and Related Methods, Terrasini, Italy. Volume 1071 of LNCS, Springer (1996) 110–126
36. Letz, R., Stenz, G.: The disconnection tableau calculus. Journal of Automated Reasoning **38** (2007) 79–126
37. Letz, R.: First-order tableau methods. In D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer, Dordrecht (1999) 125–196
38. Tarski, A.: A decision method for elementary algebra and geometry. 2nd edn. University of California Press, Berkeley-Los Angeles (1951)
39. 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.
40. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
41. Harrison, J.: The HOL light manual (1.1) (2000)
42. Dutertre, B.: System description: Yices 1.0.10. In: SMT-COMP'07. (2007)
43. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning **41** (2008) 143–189
44. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, First International Joint Conference on Automated Reasoning, Siena, Italy. Volume 2083 of LNCS, Springer (2001) 545–560
45. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
46. Ganesh, V.: Decision procedures for bit-vectors, arrays and integers. PhD thesis, Stanford, CA, USA (2007) Adviser: David L. Dill.
47. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Proceedings, 18th International Conference on Computer-Aided Verification, Seattle, USA. Volume 4144 of LNCS, Springer (2006) 81–94
48. Fisher, M.J., Rabin, M.O.: Superexponential complexity of Presburger's arithmetic. Proceedings, AMS Symposium on the Complexity of Computational Processes **7** (1974) 27–41
49. Weispfenning, V.: Complexity and uniformity of elimination in Presburger arithmetic. In: Proceedings, International Symposium on Symbolic and Algebraic Computation, ACM Press (1997) 48–53
50. Halpern, J.Y.: Presburger arithmetic with unary predicates is Π^1_1 complete. Journal of Symbolic Logic **56** (1991)
51. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding boolean algebra with Presburger arithmetic. Journal of Automated Reasoning **36** (2006) 213–239
52. Piskac, R., Kuncak, V.: Linear arithmetic with stars. In: Proceedings, 20th International Conference on Computer-Aided Verification, Princeton, USA. Volume 5123 of LNCS, Springer (2008) 268–280
53. Cooper, D.: Theorem proving in arithmetic without multiplication. Machine Intelligence **7** (1972) 91–99
54. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM **8** (1992) 102–114
55. Oppen, D.C.: A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. Journal of Computer and System Sciences **16** (1978) 323–332

56. Norrish, M.: Complete integer decision procedures as derived rules in HOL. In: Proceedings, Theorem Proving in Higher Order Logics. Volume 2758 of LNCS, Springer (2003) 71–86
57. Janičić, P., Green, I., Bundy, A.: A comparison of decision procedures in Presburger arithmetic. In: Proceedings, 8th International Conference on Logic and Computer Science, Novi Sad, Yugoslavia (1997) 99–101
58. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms. Addison-Wesley (1997) Third edition.
59. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. 2nd edn. Addison Wesley (2000)
60. Sun Microsystems, Inc. Palo Alto/CA, USA: Java Card 2.2 Platform Specification. (2002)
61. Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for Java. In: Proceedings, 15th International Symposium on Software Reliability Engineering, Saint-Malo, France, IEEE Computer Society (2004) 245–256
62. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
63. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings, Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, USA, ACM Press (1977) 238–252
64. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (2005) Corrected second printing.
65. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
66. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer **2** (1998)
67. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10** (2003) 203–232
68. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: Proceedings, 22nd International Conference on Software engineering, Limerick, Ireland, ACM Press (2000) 439–448
69. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings, 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, USA, ACM Press (2002) 1–3
70. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. International Journal on Software Tools for Technology Transfer **9** (2007) 505–525
71. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Notices **39** (2004) 92–106
72. Artho, C.: JLint (2001) `http://artho.com/jlint`.
73. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
74. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. (2007)
75. Object Modeling Group: Object Constraint Language Specification, version 1.1. (1997)

76. Liu, H., Moore, J.S.: Java program verification via a JVM deep embedding in ACL2. In Slind, K., Bunker, A., Gopalakrishnan, G., eds.: Proceedings, 17th International Conference on Theorem Proving in Higher Order Logics, Park City, USA. Volume 3223 of LNCS, Springer (2004) 184–200
77. Jacobs, B., van den Berg, J., Huisman, M., van Berkum, M., Hensel, U., Tews, H.: Reasoning about Java classes: Preliminary report. In: Proceedings, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Vancouver, Canada, ACM Press (1998) 329–340
78. Arts, T., Chugunov, G., Dam, M., Fredlund, L., Gurov, D., Noll, T.: A tool for verifying software written in Erlang. International Journal of Software Tools for Technology Transfer **4** (2003) 405–420
79. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
80. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings, ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, ACM Press (2002) 234–245
81. 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.: Proceedings, International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, France. Volume 3362 of LNCS, Springer (2005) 49–69
82. Burdy, L., Requet, A., Lanet, J.: Java applet correctness: a developer-oriented approach. In: Proceedings, 12th International FME Symposium, Pisa, Italy. Volume 2805 of LNCS, Springer (2003) 422–439
83. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (2003)
84. 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
85. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
86. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: Proceedings, 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Warsaw, Poland. Volume 2619 of LNCS, Springer (2003) 553–568
87. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings, Second Conference on Computer Science and Engineering, Linköping. (1999) 21–28
88. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580
89. Meyer, J., Poetzsch-Heffter, A.: An architecture for interactive program provers. In Graf, S., Schwartzbach, M.I., eds.: Proceedings, 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems, Berlin, Germany. LNCS, Springer (2000) 63–77
90. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience **13** (2001) 1173–1214
91. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
92. Leino, K.R.M.: Efficient weakest preconditions. Information Processing Letters **93** (2005) 281–288
93. Wirsing, M.: Algebraic specification. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). (1990) 675–788

94. Stenzel, K.: A formally verified calculus for full JavaCard. In Rattray, C., Maharaj, S., Shankland, C., eds.: Proceedings, 10th International Conference on Algebraic Methodology and Software Technology, Stirling, Scotland. Volume 3116 of LNCS, Springer (2004) 491–505
95. 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
96. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings, 17th IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark, IEEE Computer Society (2002) 55–74
97. Gedell, T., Hähnle, R.: Automating verification of loops by parallelization. In Hermann, M., Voronkov, A., eds.: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia. Volume 4246 of LNCS, Springer (2006) 332–346
98. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: Proceedings, International Symposium on Software Testing and Analysis, Rome, Italy. (2002) 123–133
99. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: Proceedings, 16th IEEE International Conference on Automated Software Engineering, Coronado Island, USA, IEEE Computer Society (2001)
100. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35** (2000) 268–279
101. Shah, M.A.: Generating counterexamples for Java dynamic logic. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden (2005)
102. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In Gurevich, Y., Meyer, B., eds.: Proceedings, First International Conference on Tests and Proofs, Zurich, Switzerland. Volume 4454 of LNCS, Springer (2007) 41–60
103. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In Necula, G.C., Wadler, P., eds.: ACM Symposium on Principles of Programming Languages, San Francisco, USA, ACM Press (2008) 147–158
104. Velroyen, H.: Automatic non-termination analysis of imperative programs. Master's thesis, Chalmers University of Technology, Aachen Technical University, Göteborg, Sweden and Aachen, Germany (2007)
105. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In Beckert, B., Hähnle, R., eds.: Proceedings, Second International Conference on Tests and Proofs, Prato, Italy. Volume 4966 of LNCS, Springer (2008) 154–170
106. Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P., Schmitt, P.H.: Verifying object-oriented programs with KeY: A tutorial. In: 5th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands. Volume 4709 of LNCS, Springer (2007) 70–101
107. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In Hermann, M., Voronkov, A., eds.: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia. Volume 4246 of LNCS, Springer (2006) 422–436
108. Beckert, B., Giese, M., Hähnle, R., Klebanov, V., Rümmer, P., Schlager, S., Schmitt, P.H.: The KeY System 1.0 (deduction component). In Pfenning, F.,

ed.: Proceedings, International Conference on Automated Deduction, Bremen, Germany. Volume 4603 of LNCS, Springer (2007) 379–384

109. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In Beckert, B., ed.: Proceedings, Fourth International Verification Workshop, Bremen, Germany. Volume 259 of CEUR (`http://ceur-ws.org/`) (2007)
110. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Doha, Qatar. LNCS, Springer (2008) To appear.
111. van Heijenoort, J., ed.: From Frege to Gödel: a Source Book in Mathematical Logic, 1879–1931. Harvard University Press (1967) Reprinted 1971, 1976.
112. Szabo, M.E., ed.: The Collected Papers of Gerhard Gentzen. North-Holland, Amsterdam (1969)
113. Berka, K., Kreiser, L., eds.: Logik-Texte. Kommentierte Auswahl zur Geschichte der modernen Logik. Akademie-Verlag, Berlin (1986)

Paper 1

Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic

Philipp Rümmer and Muhammad Ali Shah

Proceedings of the First International Conference on Tests and Proofs
Zurich, Switzerland, 2007
Volume 4454 of LNCS, Springer (2007) 41–60

The version in this thesis contains minor updates.

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@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. 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 formulae. 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 provides further details about incorrectness proofs using the incremental closure approach. Sect. 6 discusses related work, and Sect. 7 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 the 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 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;
}
```

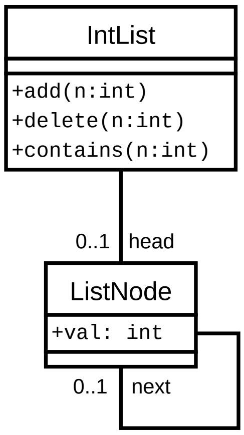


Fig. 1. The running example, a simple implementation of singly-linked lists, annotated with JML [3] constraints. We concentrate on the method `delete` for removing all elements with a certain value, which contains bugs.

Updates. JavaDL features a notation for updating functions in a substitution-like style [4] (page 115), which is primarily useful because it allows for a simple and natural memory representation during symbolic execution. For our purposes, *updates* can be seen as a simplistic programming language and are defined by the grammar:

$$\mathit{Upd} \ ::= \ \texttt{skip} \ \big| \ f(s_1,\ldots,s_n) := t \ \big| \ \mathit{Upd}|\mathit{Upd} \ \big| \ \texttt{if}\ \phi\ \{\mathit{Upd}\} \ \big| \ \texttt{for}\ x\ \{\mathit{Upd}\}$$

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:

Expression with update:	Equivalent update-free expr.:
$\{a := g(3)\}\,f(a)$	f(g(3))
$\{x := y \mid y := x+1\}\,(x < y)$	$y < x + 1$
$\{a := 3 \mid \texttt{for}\ x\ \{f(x) := 2 \cdot x + 1\}\}\,f(f(a))$	15

As illustrated 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 compilation 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. The term $C.get(i)$ denotes the ith object of class C (i is the index or "address"). For distinct classes C and D, the expressions $C.get(i)$ and $D.get(j)$ never have the same value. Each sort C representing a class also contains a distinguished individual denoted by *null* to represent undefined references. Attributes of type T of a class C are modelled by functions $C \to T$. Instead of $attr(o)$, we usually 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*, ...	*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 model). Because we, most likely, will consider abrupt termination as a violation of the post-conditions, the diamond operator is more useful at this point.

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 quantification over algebraic datatypes like lists. The number of quantified locations is unbounded, but finite.

Attributes of primitive types. The simplest case is an attribute *attr* of a primitive Java type. If *attr* has type *int*, the quantification can be performed as follows:

$$\exists\, \mathit{attr}_V : \mathit{intList}.\ \ \{\texttt{for}\ x : \mathit{nat}\ \{C.\mathit{get}(x).\mathit{attr} := \mathit{attr}_V \downarrow x\}\}\ \ldots$$

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

$$[a_0, \ldots, a_n] \downarrow i \ := \begin{cases} a_i & \text{for } i \le n \\ 0 & \text{otherwise} \end{cases} \qquad (i : \mathit{nat})$$

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. Instead of lists, one could also talk about functions with finite support.

[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.

Attributes of reference types. The quantification is a bit more involved for attributes *attr* of type D, where D is a reference type like 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:

$$D.get(s,i) \;:=\; \begin{cases} D_s.get(i) & \text{for } i < D_s.nextToCreate,\ s \le k \\ null & \text{otherwise} \end{cases} \qquad (s,i : nat)$$

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. The receiver 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` in fact removes the elements with value n.

$$\begin{array}{l}
\exists\, k_{IL}, k_{LN}, o_V : nat.\ \exists\, n_V : int.\ \exists\, head_V, next_V : natList.\ \exists\, val_V : intList. \\
\ \{IntList.nextToCreate := k_{IL} \mid ListNode.nextToCreate := k_{LN}\} \\
\ \{\texttt{for}\ x : nat\ \{IntList.get(x).head := ListNode.get(0, head_V \downarrow x)\} \mid \\
\quad \texttt{for}\ x : nat\ \{ListNode.get(x).next := ListNode.get(0, next_V \downarrow x)\} \mid \\
\quad \texttt{for}\ x : nat\ \{ListNode.get(x).val := val_V \downarrow x\} \mid \\
\quad o := IntList.get(0, o_V) \mid n := n_V\} \\
\qquad \neg\big(\ o \neq null \rightarrow \langle\, o.\texttt{delete}(n)\,\rangle\, \langle\, b = o.\texttt{contains}(n)\,\rangle\, b = FALSE\ \big)
\end{array} \qquad (5)$$

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 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 [5] for

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \land \psi, \Delta}\ \land\text{R} \qquad \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \land \psi \vdash \Delta}\ \land\text{L} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \neg\text{R}$$

$$\frac{\Gamma \vdash \phi[x/f(X_1, \ldots, X_n)], \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \forall\text{R} \qquad (X_1, \ldots, X_n \text{ all metavariables in } \phi)$$

$$\frac{\Gamma \vdash \phi[x/X], \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \exists\text{R} \qquad (X \text{ a fresh metavariable})$$

$$\frac{\Gamma, \{u\}\,\{r := l\}\,\langle \ldots \rangle\,\phi \vdash \Delta}{\Gamma, \{u\}\,\langle r = l; \ldots \rangle\,\phi \vdash \Delta}\ \text{ASSIGN-L} \qquad (r, l \text{ side-effect-free})$$

$$\frac{\begin{array}{c}\Gamma, \{u\}\,\langle \alpha_1; \ldots \rangle\,\phi, \{u\}\,b \vdash \Delta \\ \Gamma, \{u\}\,\langle \alpha_2; \ldots \rangle\,\phi \vdash \{u\}\,b, \Delta\end{array}}{\Gamma, \{u\}\,\langle \texttt{if}\ (b)\ \alpha_1\ \texttt{else}\ \alpha_2\ \ldots \rangle\,\phi \vdash \Delta}\ \text{IF-L} \qquad (b \text{ side-effect-free})$$

$$\frac{\Gamma, \{u\}\,\langle \texttt{if}\ (b)\ \{\alpha;\ \texttt{while}\ (b)\ \alpha\} \ldots \rangle\,\phi \vdash \Delta}{\Gamma, \{u\}\,\langle \texttt{while}\ (b)\ \alpha\ \ldots \rangle\,\phi \vdash \Delta}\ \text{WHILE-L}$$

Fig. 2. Examples of (simplified) sequent calculus rules for JavaDL. In the last three rules, the update u can also be empty (`skip`) and disappear. Γ and Δ denote arbitrary sets of formulae (side-formulae).

Java (the last three rules of Fig. 2), and (iii) rewriting rules for applying and simplifying updates (not shown here, see [4] on page 115). The rule ASSIGN-L turns a Java assignment into an update, which can subsequently be merged with the preceding update u and simplified. In IF-L, a case analysis for an if-statement is performed by splitting on the branch predicate b evaluated in the current program state u. Both rules require that expressions with side-effects are simplified first. Finally, the rule WHILE-L unwinds a loop once.

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 [5] 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 and for statements that might raise exceptions. A simple example is the following proof:

$$\frac{\dfrac{\dfrac{\vdots}{p+1 \leq 0, p \geq 0 \vdash}}{\dfrac{\{p := p+1\}\,\langle\rangle\,p \leq 0, p \geq 0 \vdash}{\langle p = p+1; \rangle\,p \leq 0, p \geq 0 \vdash}}\ \text{ASSIGN-L} \qquad \dfrac{\dfrac{\vdots}{-p \leq 0 \vdash p \geq 0}}{\dfrac{\{p := -p\}\,\langle\rangle\,p \leq 0 \vdash p \geq 0}{\langle p = -p; \rangle\,p \leq 0 \vdash p \geq 0}}}{\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p; \rangle\,p \leq 0 \vdash}\ \text{IF-L}$$

Symbolic execution and update application can usually be automated easily—in contrast to reasoning in first-order logic—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. Assuming that pre- and post-conditions only talk about the program state, it is sufficient to apply $\exists\text{R}$ once for each quantifier in $\exists\, \textit{pre-state}$, because the validity of (4) only depends on the program fragment and the pre- and post-conditions in question, not on the values of other symbols.

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, Sect. 5). 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 Construction of Proofs using a Ground Procedure

The simplest approach to prove (4) is *ground reasoning*, i.e., to not use metavariables at all. A ground version of $\exists\text{R}$ is sufficient in this case:

$$\frac{\Gamma \;\vdash\; \phi[x/t], \exists x.\phi, \Delta}{\Gamma \;\vdash\; \exists x.\phi, \Delta}\;\exists\text{R}_g \qquad (t \text{ an arbitrary term})$$

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 $\textit{IntList.get}(0)$ receives the method invocation (o_V) with argument 5 (n_V), that $\textit{IntList.get}(0).\textit{head}$ points to the object $\textit{ListNode.get}(0)$ ($head_V$), that $\textit{ListNode.get}(0).\textit{next}$ is *null* ($next_V$, because of $7 \geq k_{LN}$), i.e., that the receiving list has only one element, and that $\textit{ListNode.get}(0).\textit{val}$ 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

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{[\,P \mapsto 2\,]}}{P+1>3, P \geq 0 \;\vdash}}{\{p := P+1\}\,\langle\rangle\, p > 3, p \geq 0 \;\vdash}}{\{p := P\}\,\langle p = p+1;\rangle\, p > 3, P \geq 0 \;\vdash} \qquad \dfrac{\dfrac{*}{[\,P \mapsto 2\,]}}{\{p := P\}\,\langle p = -p;\rangle\, p > 3 \;\vdash\; P \geq 0}}{\{p := P\}\,\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\rangle\, p > 3 \;\vdash}\ \textsc{if-l}}{\vdash\; \neg\{p := P\}\,\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\rangle\, p > 3,\ \ldots}\ \neg\textsc{r}}{\vdash\; \exists p_V : \mathit{int}.\ \{p := p_V\}\,\neg\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\rangle\, p > 3}\ \exists\textsc{r}
$$

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.

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., [6]). 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, is 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., [6]), 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., [7]).

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 [8]. 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 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 $\wedge$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 made 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 principal 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 be generalised further.

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 [9] and worked out in detail in [10]. While backtracking-free proof search is, in principle, also possible when immediately applying substitutions, removing this destructive operation vastly simplifies proving without backtracking. Because KeY implements this technique, it is used in our experiments.

The approach of [10] 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 never commits 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 inequality $2 \geq 0$ 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 [10], 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 $X \geq 0$.[5]

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)

[5] Depending on the representation of integers or natural numbers, certain inequalities like $X \geq 1 \Leftrightarrow X \equiv \mathrm{succ}(X')$ might be expressible, but this concept is rather restricted.

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, [9]):

Definition 1. *A constraint C is called* consistent *if for each arithmetic structure (interpreting the symbols $+$, $-$, $\neq$, $<$, $\leq$ and literals as is common over the integers, and all other function symbols arbitrarily), there is an assignment of values to metavariables such that C is evaluated to tt.*

Example 2. Of the following constraints, C_1, C_2 and C_3 are consistent, while the others are not. C_4 is inconsistent because the ranges of f and g could be disjoint, C_5 because f could be the identity, and C_6 because 5 could be outside of the range of the function $\cdot\downarrow\cdot$. Our constraint language does not know about lists, so that $\cdot\downarrow\cdot$ is just an arbitrary function symbol in this regard.

$$\begin{array}{lllllll} C_1 & := & X = 5 \wedge 2 = Y + 1 & \quad & C_2 & := & h(A,2) = h(h(c,Y), Y+1) \\ C_3 & := & c < X \wedge d \leq X & & C_4 & := & f(X) = g(Y) \\ C_5 & := & X < f(X) & & C_6 & := & (ATTR\downarrow O) = 5 \end{array}$$

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}	K_{LN}	0	N_V	$[0,\ldots]$	$[E,\ldots]$	$[N_V,\ldots]$

$$\begin{array}{l} K_{IL} > 0 \ \wedge \\ K_{LN} > 0 \ \wedge \\ E \geq K_{LN} \end{array}$$

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 Reasoning about Lists and Arithmetic

The next pages give more (implementation) details and treat some further aspects of the backtracking-free method from Sect. 3.3. As incremental closure works by enumerating the closing constraints of all proof branches, the central issue is to design suitable goal-local rules that produce such constraints, and to develop an application strategy that defines which rule should be applied at which point in a proof. The solutions shown here are tailored to the constraint language of the previous section.

5.1 Rules for the Theory of Lists

For proof obligations of the form (4), the closing constraints of a goal mostly describe the values of metavariables X_1, X_2, … over lists—the lists that in Sect. 2.3 are used to represent program states—and usually have the form:

$$\begin{aligned} &X_1 = \mathit{cons}(X_1^1, \mathit{cons}(X_1^2, \ldots)) \;\wedge\; X_2 = \mathit{cons}(X_2^1, \mathit{cons}(X_2^2, \ldots)) \;\wedge\; \cdots \\ \wedge\; &C(X_1^1, X_1^2, \ldots, X_2^1, X_2^2, \ldots) \end{aligned}$$

Such constraints consist of a first part that determines to which depth the lists $X_1, X_2, \ldots$ have been "expanded," and of a part $C(X_1^1, X_1^2, \ldots, X_2^1, X_2^2, \ldots)$ (which is again a constraint, e.g. in the language from Sect. 4) that describes the values of list elements. Following Sect. 2.3, each of the list elements $X_1^1, X_1^2, \ldots$ belongs to one object of a class.

The expansion of lists is handled by a single rule that introduces fresh metavariables H, T for the head and the tail of a list. We use the *constrained formula* approach from [10] to remember this decomposition of a list L into two parts. A constrained formula is a pair $\phi \ll C$ consisting of a formula ϕ and a constraint C. The semantics of a formula $\phi \ll C$ that occurs in the antecedent of a sequent is (roughly) the same as of the implication $C \to \phi$, and in the succedent the semantics is $C \wedge \phi$: intuitively, the presence of ϕ can only be assumed if the constraint C holds. C has to be kept and propagated to all formulae that are derived from $\phi \ll C$ during the course of a proof. If $\phi \ll C$ is used to close a proof branch, the closing constraint that is created has to be conjoined with C.

The rule for expanding lists is essentially a case distinction on whether the head ($i = 0$) or a later element ($i > 0$) of a list is accessed. An attached constraint $[\, L = \mathit{cons}(H, T)\,]$ expresses that the name H is introduced for the head of the list and T for its tail. In practice, the rule is only applied if an expression $L \downarrow i$ occurs in the sequent $\Gamma \;\vdash\; \Delta$, where L is a metavariable. As described in Sect. 2.3, the length of lists is irrelevant, so that the case $L = \mathit{nil}$ does not have to be taken into account:

$$\frac{\begin{array}{c}\Gamma, (i = 0 \wedge (L \downarrow 0) = H) \ll [\, L = \mathit{cons}(H, T)\,] \;\vdash\; \Delta \\ \Gamma, (i > 0 \wedge (L \downarrow i) = (T \downarrow (i-1))) \ll [\, L = \mathit{cons}(H, T)\,] \;\vdash\; \Delta\end{array}}{\Gamma \;\vdash\; \Delta}\;\textsc{expand-list}$$

(H, T fresh metavariables)

Fig. 4 shows an example how EXPAND-LIST is used to enumerate the solutions of the formula $L \downarrow X > 3$. By repeated application of EXPAND-LIST, all list access

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} * \\ [\,H > 3 \land C\,] \end{array}}{X = 0 \ll C, L{\downarrow}0 = H \ll C, H \le 3 \ll C \;\vdash\;}\,\le\text{L}}{X = 0 \ll C, L{\downarrow}0 = H \ll C \;\vdash\; H > 3 \ll C}}{X = 0 \ll C, L{\downarrow}0 = H \ll C \;\vdash\; L{\downarrow}X > 3}}{(X = 0 \land L{\downarrow}0 = H) \ll C \;\vdash\; L{\downarrow}X > 3}}{\mathcal{D}}$$

$$\dfrac{\mathcal{D} \qquad \dfrac{\dfrac{\dfrac{\begin{array}{c} * \\ [\,X < 1 \land C\,] \end{array}}{X \ge 1 \ll C, L{\downarrow}X = T{\downarrow}(X-1) \ll C \;\vdash\; L{\downarrow}X > 3}\,\ge\text{L}}{X > 0 \ll C, L{\downarrow}X = T{\downarrow}(X-1) \ll C \;\vdash\; L{\downarrow}X > 3}}{(X > 0 \land L{\downarrow}X = T{\downarrow}(X-1)) \ll C \;\vdash\; L{\downarrow}X > 3}}{\vdash\; L{\downarrow}X > 3}\;\textsc{expand-list}$$

Fig. 4. Example for a proof involving lists and metavariables $L, T : \mathit{intList}$, $H : \mathit{int}$, $X : \mathit{nat}$. We write C as abbreviation for the constraint $[\,L = \mathit{cons}(H, T)\,]$. The first solution (shown here) that is produced by the proof is $[\,L = \mathit{cons}(H, T) \land X < 1 \land H > 3\,]$ and stems from the formulae $X \ge 1 \ll C$ and $H \le 3 \ll C$ in the two branches. When applying further rules to the proof—instead of closing it—and expanding the list more than once, further solutions like $[\,L = \mathit{cons}(H, \mathit{cons}(H', T')) \land X = 1 \land H' > 3\,]$ can be generated. Concerning the handling of inequalities in the proof, see Sect. 5.3.

expressions $L{\downarrow}i$ in a sequent can be replaced with scalar metavariables, which subsequently can be handled with other rules for first-order logic and arithmetic.

Because EXPAND-LIST splits on the value of the list index i, it can happen that different isomorphic heap arrangements are explored in different goals of a proof. For a larger number of objects, this can obviously leads to a combinatorial explosion. Generally, two possibilities to handle this issue (which we have not investigated yet) are (i) to work with a constraint language that directly supports the theory of lists, or to (ii) use the approach suggested in Sect. 3.4 to focus on one particular heap arrangement, ignoring isomorphic heaps. In this manner, it is, for instance, possible to simulate the lazy-initialisation approach from [11].

5.2 Fairness Conditions

As the different branches (and formulae) of a proof are expanded completely independently when using incremental closure, it is important to choose a fairness strategy that ensures an even distribution of rule applications. When proving program incorrectness, there are two primary parameters that describe how far a problem has been explored: (i) how often loops have been unwound on a branch (the number of applications of the rule WHILE-L from Fig. 2), and the (ii) the depth to which lists have been expanded (the size of the heap under consideration, or the number of applications of the rule EXPAND-LIST from the previous section).

In the KeY prover, automatic reasoning is controlled by *strategies*, which are basically cost computation functions that assign each possible rule application in a proof an integer number as cost. The rule application that has been given the least cost (for the whole proof) is carried out first. In this setting, we achieve fairness in the following way:

– Applications of WHILE-L are given the cost $c_w = \alpha_w \cdot k_w + o_w$, where k_w is the number of applications of WHILE-L that have already been performed on a proof branch, and $\alpha_w > 0$, o_w are constants. This means that the cost for unwinding a loop a further time grows linearly with the number of earlier loop unwindings.
– Applications of EXPAND-LIST are given the cost $c_e = \alpha_e \cdot k_e + o_e$, where k_e is the sum of the depths to which each of the list metavariables has been expanded on a proof branch. This sum can be computed by considering the constraints C that are attached to formulae $\phi \ll C$ in a sequent that contain list access expressions $L \downarrow i$: one can simply count the occurrences of *cons* in the terms that have to be substituted for the original list metavariables when solving the constraint C.[6]

Good values for the constants α_w, α_e are in principle problem-dependent, but in our experience it is meaningful to choose α_e (a lot) bigger than α_w. When proving the formula (5), yielding the constraint shown in Sect. 4, we had chosen $\alpha_w = 50$, $o_w = 200$, $\alpha_e = 2500$, $o_e = -2000$.

A slightly different approach is to choose a fixed upper bound either for the number of loop unwindings or for the heap size, and to let only the other parameter grow unboundedly within one proof attempt. If the proof attempt fails, the bound can be increased and a new proof is started. In the experiments so far, we have not found any advantages of starting multiple proof attempts over the method described first, however.

5.3 Arithmetic Handling in KeY

The heap representation that is introduced in Sect. 2.3 heavily uses arithmetic (both natural and integer numbers). After the elimination of programs using symbolic execution, of updates and of list expressions, the construction of solutions or closing constraints essentially boils down to handling arithmetic formulae. Although KeY is in principle able to use the theorem prover Simplify [12] as a back-end for discharging goals that no longer contain modal operators and programs, this does not provide any support when reasoning with metavariables. In this section, we shortly describe the native support for arithmetic that we, thus, have added to KeY (see page 149 for a more detailed account).

[6] The actual computation of c_e is more complicated, because smaller costs are chosen when applying EXPAND-LIST for terms $L \downarrow i$ in which i is a concrete literal, or when the rule has already been applied for the same list L earlier.

Linear arithmetic. Equations and inequalities over linear polynomials is the most common and most important fragment of integer arithmetic. We use Fourier-Motzkin variable elimination to handle such formulae—inspired by the Omega test [13], which is an extension of Fourier-Motzkin. Although Fourier-Motzkin does not yield a complete procedure over the integers, in contrast to the Omega test, we have so far not encountered the need to create a full implementation of the Omega test.

As a pre-processing step, the equations and inequalities of a sequent are always moved to the antecedent and are transformed into inequalities $c \cdot x \leq s$ or $c \cdot x \geq s$, where c is a positive number and s is a term. In order to ensure termination, we assume the existence of a well-ordering on the set of variables of a problem and require that x is strictly bigger than all variables in s. Fourier-Motzkin variable elimination can then be realised by the following rule:

$$\frac{\Gamma, c \cdot x \geq s, d \cdot x \leq t, d \cdot s \leq c \cdot t \ \vdash \ \Delta}{\Gamma, c \cdot x \geq s, d \cdot x \leq t \ \vdash \ \Delta}\ \textsc{transitivity} \qquad (c > 0, d > 0)$$

Apart from the rule for eliminating variables from inequalities, we also have to provide rules for generating closing constraints (using the constraint language from Sect. 4):

$$\frac{[\, s = t \,]}{\Gamma \ \vdash \ s = t, \Delta}\ {=}\textsc{r} \qquad \frac{[\, s \neq t \,]}{\Gamma, s = t \ \vdash \ \Delta}\ {=}\textsc{l} \qquad \frac{[\, s > t \,]}{\Gamma, s \leq t \ \vdash \ \Delta}\ {\leq}\textsc{l} \qquad \frac{[\, s < t \,]}{\Gamma, s \geq t \ \vdash \ \Delta}\ {\geq}\textsc{l}$$

Non-linear arithmetic. In order to handle multiplication, division- and modulo-operations that frequently occur in programs, we have also added some support for non-linear integer arithmetic to KeY. Our approach is similar to that of the ACL2 theorem prover [14] and is based on the following rule (together with the rules for handling linear arithmetic):

$$\frac{\Gamma, s \leq s', t \leq t', 0 \leq (s' - s) \cdot (t' - t) \ \vdash \ \Delta}{\Gamma, s \leq s', t \leq t' \ \vdash \ \Delta}\ \textsc{mult-inequalities}$$

Often, it is also necessary to perform a systematic case analysis. The rule MULT-INEQUALITIES alone is, for instance, not sufficient to prove simple formulae like $x \cdot x \geq 0$. Case distinctions can be introduced with the following rules:

$$\frac{\begin{array}{l}\Gamma, x < 0 \ \vdash \ \Delta \\ \Gamma, x = 0 \ \vdash \ \Delta \\ \Gamma, x > 0 \ \vdash \ \Delta\end{array}}{\Gamma \ \vdash \ \Delta}\ \textsc{sign-cases} \qquad \frac{\begin{array}{l}\Gamma, s < t \ \vdash \ \Delta \\ \Gamma, s = t \ \vdash \ \Delta\end{array}}{\Gamma, s \leq t \ \vdash \ \Delta}\ \textsc{strengthen}$$

We can now prove $x \cdot x \geq 0$ by first splitting on the sign of x. The rules SIGN-CASES and STRENGTHEN are in principle sufficient to find solutions for arbitrary solvable polynomial equations and inequalities. Combined with the rules $=$R, $=$L, $\leq$L, $\geq$L from above, this guarantees that the calculus can always produce solutions and closing constraints for satisfiable sequents that (only) contain such formulae.

6 Related Work

Proof strategies based on metavariables and backtracking are related to common approaches to test data generation with symbolic execution, see, e.g., [5, 7]. 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. [15]. 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 relationship between ground proof procedures and test data generation using actual program execution. Constructing proofs using metavariables can be seen as exhaustive testing, because the behaviour of a program is examined (simultaneously) for all possible inputs. When using the fairness approach of limiting the size of the initial heap that is described in Sect. 5.2, the method is related to bounded exhaustive testing, because only program inputs up to a certain size are considered.

A technique that can be used both for proving programs correct and incorrect is abstraction-refinement model checking (e.g., [16–18]). 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 [19] and Spec#/Boogie [20] are verification systems for object-oriented languages that use the prover Simplify [12] as back-end. Simplify is able to derive counterexamples from failed proof attempts, which are subsequently used to create warnings about possible erroneous behaviour of a program for certain concrete situations. Another example is [21], 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.

7 Conclusions and Future Work

The development of the proposed method and of its prototypical implementation has been driven by working with (small) examples [22], 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 [23, 22]. 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. As explained in Sect. 5.1, 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 (page 61).

Acknowledgements

We want to thank Wolfgang Ahrendt for many discussions that eventually led to this paper, and Tobias Nipkow for comments on an older version of the paper. Thanks are also due to the anonymous referees for helpful comments and hints.

References

1. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
2. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2007)
3. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. (2007)
4. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In Hermann, M., Voronkov, A., eds.: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia. Volume 4246 of LNCS, Springer (2006) 422–436
5. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35** (2000) 268–279
7. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings, Second Conference on Computer Science and Engineering, Linköping. (1999) 21–28
8. 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

9. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
10. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, First International Joint Conference on Automated Reasoning, Siena, Italy. Volume 2083 of LNCS, Springer (2001) 545–560
11. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: Proceedings, 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Warsaw, Poland. Volume 2619 of LNCS, Springer (2003) 553–568
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52** (2005) 365–473
13. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM **8** (1992) 102–114
14. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common lisp. IEEE Transactions on Software Engineering **23** (1997) 203–213
15. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. In: Proceedings, 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, USA, ACM Press (1998) 231–244
16. 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
17. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10** (2003) 203–232
18. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings, ACM SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, USA, ACM Press (2001) 203–213
19. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings, ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, ACM Press (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.: Proceedings, International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, France. Volume 3362 of LNCS, Springer (2005) 49–69
21. Reif, W., Schellhorn, G., Thums, A.: Flaw detection in formal specifications. In: Proceedings, First International Joint Conference on Automated Reasoning, Siena, Italy. LNCS, Springer (2001) 642–657
22. Shah, M.A.: Generating counterexamples for Java dynamic logic. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden (2005)
23. Parasoft: JTest (2006) `http://www.parasoft.com`.

Paper 2

Non-Termination Checking for Imperative Programs

Helga Velroyen and Philipp Rümmer

Proceedings of the Second International Conference on Tests and Proofs
Prato, Italy, 2008
Volume 4966 of LNCS, Springer (2008) 154–170

The version in this thesis contains minor typographic changes.

Non-Termination Checking for Imperative Programs

Helga Velroyen[1] and Philipp Rümmer[2]

[1] Department of Computer Science
RWTH Aachen University of Technology
helga.velroyen@rwth-aachen.de
[2] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
philipp@chalmers.se

Abstract. While termination checking tailored to real-world library code or frameworks has received ever-increasing attention during the last years, the complementary question of *disproving* termination properties as a means of debugging has largely been ignored so far. We present an approach to automatic non-termination checking that relates to termination checking in the same way as symbolic testing does to program verification. Our method is based on the automated generation of invariants that show that terminating states of a program are unreachable from certain initial states. Such initial states are identified using constraint-solving techniques. The method is fully implemented on top of a program verification system and available for download. We give an empirical evaluation of the approach using a collection of non-terminating example programs.

1 Introduction

Termination properties of programs are crucial for liveness and safety: a piece of software which does not terminate can have vast consequences, especially when employed in critical environments or wide-spread. The latter concerns in particular library code or frameworks, whose specific use is often unknown at the time of development. Non-termination bugs can be very subtle and hide long before they take effect in productivity situations.

Although the concept of formally proving termination properties has been known and investigated for a long time, the last years have seen intensified research on how to check the termination of real-world code [1, 2]. During the same time, however, the complementary field of showing the potential *non-termination* of programs as a means of debugging has largely been ignored. This is a surprising situation, because programs under development are prone to contain defects. In this context, direct attempts to find those bugs might be more successful and more useful than to learn from failed correctness or termination proofs.

Traditional dynamic techniques of testing program behavior by means of concrete execution are not adequate to show non-termination (they can nevertheless provide valuable hints). As a consequence, although the purpose of non-termination analysis is more related to testing than to program verification, in most cases the usage of symbolic reasoning cannot be avoided. In the present paper, we introduce an approach to automatic non-termination checking that relates to termination checking in the same way as symbolic testing does to program verification. The method has been implemented on top of a general-purpose program verification system. Experiments using a database of non-terminating programs indicate that it can be a useful tool for detecting termination defects early during software development.

Showing the non-termination of a program consists of two parts: (i) to prove that a potential loop in a program is reachable from some initial state, and (ii) to prove that the potential loop can indeed cause non-termination. We use constraint solving techniques to achieve the first part, following the approach in [3] (page 39). For the second part, we introduce an algorithm to synthesise invariants that show that the found loop is never exited and that terminating states of the program are therefore unreachable. Our approach is based on two main techniques, a template method for generating invariants (together with constraint solving) and refinement (strengthening) of invariants based on counterexamples. Because our experiments show that invariants for proving non-termination are typically much smaller than invariants for proving partial correctness, we believe that this yields a practical procedure for constructing non-termination proofs.

The paper is organised as follows: In Sect. 2 we define the programming language that is analysed in the whole paper. Sect. 3 introduces the logic and the calculus that we use to reason about programs, which is the basis for an effective algorithm in Sect. 4. An empirical evaluation of our approach is given in Sect. 5. Finally, we list related work in Sect. 6 and conclude in Sect. 7.

2 Preliminaries

We assume that the reader is familiar with classical first-order logic and Gentzen-style sequent calculi, see [4] for an introduction. For sake of simplicity, all considerations of this paper are done in the context of a simple while-language that operates on the (infinite) domain of integers. The generalisation to other imperative languages is mostly straightforward, and, in our experience, occurring problems tend to be orthogonal to the task of proving non-termination. More details are given in [5, 3] (page 39).

In order to introduce the while-language, we first assume a fixed vocabulary Σ of functions and predicates (with fixed arity) that describe the native side-effect-free operations that are available, as well as a fixed set V_p of program variables. The set Σ is supposed to contain at least literals and the standard operations on integers $(0, 1, -1, \ldots, +, -, \cdot, =, <, \leq)$. Ground terms, ground formulae and programs are then inductively defined by the following grammars:

$$\begin{aligned}
t_\mathrm{g} &::= v \mid f(t_\mathrm{g}, \ldots, t_\mathrm{g}) \\
\phi_\mathrm{g} &::= \mathit{true} \mid \mathit{false} \mid \phi_\mathrm{g} \wedge \phi_\mathrm{g} \mid \neg\phi_\mathrm{g} \mid \cdots \mid p(t_\mathrm{g}, \ldots, t_\mathrm{g}) \\
\alpha &::= \alpha \,;\, \ldots \,;\, \alpha \mid v = t_\mathrm{g} \mid \mathtt{if}\ (\phi_\mathrm{g})\ \alpha\ \mathtt{else}\ \alpha \mid \mathtt{while}\ (\phi_\mathrm{g})\ \alpha
\end{aligned}$$

where $f \in \Sigma$ ranges over functions, $p \in \Sigma$ over predicates and $v \in V_p$ over program variables.

Semantics of Programs. Because only the integers are considered as domain, a *structure* is a pair $S = (\mathbb{Z}, I)$ consisting of the set $\mathbb{Z}$ of integers and an *interpretation* I with $I(f) : \mathbb{Z}^n \to \mathbb{Z}$ if $f \in \Sigma$ is a function of arity n and $I(p) \subseteq \mathbb{Z}^n$ if $p \in \Sigma$ is a predicate of arity n. Only those structures are considered in which the standard integer operations from above (like $0, 1, -1, +, \ldots$) have their usual meaning. A *program variable assignment* is a mapping $\gamma : V_p \to \mathbb{Z}$. The space of all program variable assignments is denoted by $PA = V_p \to \mathbb{Z}$. While-programs α are evaluated in structures S and denote partial mappings $[\![\alpha]\!]^S : PA \rightharpoonup PA$ from program variable assignments to program variable assignments:

$$[\![\alpha]\!]^S(\gamma) \;=\; \begin{cases} \gamma' & \alpha \text{ terminates in state } \gamma' \text{ when started in } \gamma \\ \bot & \alpha \text{ does not terminate when started in } \gamma \end{cases}$$

Given an evaluation function $val_{S,\gamma}$ for ground terms and formulae, which is defined as is common for first-order logic (e.g., [4]), the concrete definition of $[\![\alpha]\!]^S$ follows the lines of denotational semantics (for instance, [6]).

3 Proving Non-Termination: The Calculus Level

We introduce our approach to non-termination detection in two parts: in this section, we describe the logic and the calculus to reason about programs. Based on this declarative framework, Sect. 4 defines an algorithm (a proof procedure) for automatically detecting non-termination.

Dynamic Logic for the While-Language (WhileDL). First-order dynamic logic (DL) [7] is a multi-modal extension of first-order predicate logic, in which modal operators are labelled with programs. Most importantly, given a program α and a formula ϕ, a *box-formula* $[\alpha]\,\phi$ expresses that ϕ holds in each final state of α. This paper uses a version of dynamic logic for the simple while-language [7] that is enriched with an explicit operator for simultaneous substitutions called *updates* [8, Sect. 3] (also see [9] on page 115). Updates allow us to present some of the techniques of this papers in a simpler way, but also simplify the generalisation to more involved languages like Java [5, 3, 8].

We assume the same vocabulary Σ and the same set V_p of program variables as in Sect. 2, but in addition we define a disjoint set V_l of *logical variables* that can occur in formulae and terms (outside of programs). Because some of our rules need to introduce fresh function symbols, we assume that Σ contains infinitely many functions for each arity n. Extending the grammar from Sect. 2, arbitrary terms, formulae and updates are then defined by:

$$\begin{aligned} t \;&::=\; t_{\mathrm{g}} \mid x \mid f(t, \ldots, t) \mid \{\, U \,\}\, t \\ \phi \;&::=\; \phi_{\mathrm{g}} \mid \phi \wedge \phi \mid \neg\phi \mid \;\cdots\; \mid p(t, \ldots, t) \mid [\alpha]\,\phi \mid \{\, U \,\}\, \phi \\ U \;&::=\; v := t \mid U, \ldots, U \end{aligned}$$

where $f \in \Sigma$ ranges over functions, $p \in \Sigma$ over predicates, $x \in V_l$ over logical variables and $v \in V_p$ over program variables.

In order to define the semantics of terms, formulae and updates, besides structures $S = (\mathbb{Z}, I)$ and program variable assignments $\gamma \in PA$ we also need *logical variable assignments* $\beta : V_l \to \mathbb{Z}$. The denotation $[\![U]\!]^{S,\beta} : PA \to PA$ of an update U is a total operation on program variable assignments:

$$[\![v_1 := t_1, \dots, v_k := t_k]\!]^{S,\beta}(\gamma)(w) \quad = \quad \begin{cases} val_{S,\beta,\gamma}(t_i) & w = v_i \text{ and} \\ & \quad w \notin \{v_{i+1}, \dots, v_k\} \\ \gamma(w) & w \notin \{v_1, \dots, v_k\} \end{cases}$$

This means that the assignments of an update are executed in parallel, and that assignments that syntactically occur later can override the effects of earlier assignments ($v_j := t_j$ will override $v_i := t_i$ for $v_i = v_j$ and $j > i$).

The evaluation $val_{S,\beta,\gamma}$ of terms and formulae is mostly defined as it is common for first-order predicate logic. Formulae are mapped into a Boolean domain, where tt stands for semantic truth. The cases for programs and updates are:

$$\begin{aligned} val_{S,\beta,\gamma}([\,\alpha\,]\,\phi) \quad &= \quad \begin{cases} val_{S,\beta,[\![\alpha]\!]^S(\gamma)}(\phi) & \text{if } [\![\alpha]\!]^S(\gamma) \text{ is defined} \\ \mathrm{tt} & \text{otherwise} \end{cases} \\ val_{S,\beta,\gamma}(\{\,U\,\}\;\phi) \quad &= \quad val_{S,\beta,[\![U]\!]^{S,\beta}(\gamma)}(\phi) \end{aligned}$$

We interpret free logical variables $x \in V_l$ existentially: a formula ϕ is *valid* iff for each structure S and each program variable assignment $\gamma \in PA$ there is a variable assignment $\beta : V_l \to D$ such that $val_{S,\beta,\gamma}(\phi) = \mathrm{tt}$. Likewise, a sequent $\Gamma \vdash \Delta$ is called valid iff $\bigwedge \Gamma \to \bigvee \Delta$ is valid. Free variables are used to express symbolic program inputs and as parameters in loop invariants and serve as an interface to constraint solving (see below for more details).

Characterisation of Non-Termination. Because box-formulae $[\alpha]\,\phi$ are trivially rendered true by a diverging program α, we can express non-termination by asserting *false* as post-condition: $[\alpha]\,\mathit{false}$. This means that, given a structure S, $val_{S,\gamma}([\alpha]\,\mathit{false}) = \mathrm{tt}$ holds for exactly those initial states $\gamma \in PA$ for which α diverges.

In order to express non-termination for some *arbitrary* initial state, it is necessary to quantify the variables occurring in α existentially, following the approach from [3] (page 39). For the while-language, this is done by prefixing the formula from above with an update that assigns arbitrary values to all program variables in α:

$$\{\, v_1 := x_1, \dots, v_n := x_n \,\}\; [\alpha]\,\mathit{false} \tag{1}$$

where $v_1, \dots, v_n \in V_p$ are the variables occurring in α and $x_1, \dots, x_n \in V_l$ are fresh logical variables. (1) is valid iff there are initial states from which α diverges.

A Sequent Calculus for WhileDL. To reason formally about the non-termination of programs, we introduce a Gentzen-style sequent calculus for WhileDL that follows closely the calculi in [3, 8]. Fig. 1 contains the most important calculus rules, which can be categorised as program-independent *first-order rules* (the upper part of the figure) and *symbolic execution rules*.

The rule ASSIGN turns assignments into updates, which subsequently can be merged with the former preceding update U and simplified. The simplification and application of updates is performed by the rewriting rules in Fig. 2, which propagate updates in formulae or terms downwards until they can be applied to program variables like substitutions.

In IF, a case analysis for an if-statement is performed by splitting on the branch predicate ψ evaluated in the current program state U. The invariant rule WHILE is a simplified version of the rule for Java defined in [8, Chap. 3]. In WHILE, the erasure of side formulae is avoided with the help of *anonymising updates* A_1, A_2 that assign unspecified values to all variables that can be modified by the loop body α. More formally, given that (i) $v_1, \ldots, v_n \in V_p$ are the variables that occur as left-hand sides of assignments in α, that (ii) $x_1, \ldots, x_m \in V_l$ are the logical variables that occur in U, ϕ, or Inv, and that (iii) $f_1, \ldots, f_n$ are fresh function symbols, we say that the update

$$v_1 := f_1(x_1, \ldots, x_m),\ \ldots,\ v_n := f_n(x_1, \ldots, x_m)$$

is a fresh anonymising update for α with respect to U, ϕ, Inv. Note, that we need to inject the logical variables $x_1, \ldots, x_m$ as arguments of the functions $f_1, \ldots, f_n$ for exactly the same reasons as in the standard Skolemisation rule (e.g., [4]).

Finally, *theory rules* are necessary to handle equality, integers, etc. in the calculus, we refer the reader to [10] (page 149) for more details. An example proof using the WhileDL calculus is shown below.

When inspecting the calculus rules, it can be observed that all rules but WHILE are local equivalence transformations: for all structures, program variable assignments and logical variable assignments, the conclusion of a rule holds iff all premisses hold. This property is important for us, because it implies that countermodels of an open goal are also countermodels of the initial conjecture (unless WHILE has been applied). In Sect. 4, we use counterexamples that were extracted from open proof goals to refine invariant candidates.

Incremental Closure of Proofs. In order to close a proof tree that contains free logical variables, we have to show that the variables can be given values (depending on the considered structure) such that all remaining goals are turned into obviously valid sequents. We apply the idea of *incremental closure* [4, 11] together with the arithmetic constraint language from [3, Sect. 4] (page 50) to check the existence of such values. The rules in Fig. 3 are responsible for introducing closure constraints for proof goals. If it is possible, in this way, to find compatible closure constraints for *all* proof goals (i.e., the conjunction of the constraints is valid), then it is sound to close the proof.

$$\frac{*}{\Gamma \vdash \mathit{true}, \Delta}\ \textsc{true-right} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, \mathit{true} \vdash \Delta}\ \textsc{true-left}$$

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}\ \wedge\textsc{-right} \qquad \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \wedge\textsc{-left}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \neg\textsc{-right} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \neg\textsc{-left}$$

$$\cdots$$

$$\frac{\Gamma \vdash \{U\}\phi, \Delta}{\Gamma \vdash \{U\}[\,]\phi, \Delta}\ \textsc{Skip} \qquad \frac{\Gamma \vdash \{U\}\{v := t\}[\ldots]\phi, \Delta}{\Gamma \vdash \{U\}[v = t\,;\ \ldots]\phi, \Delta}\ \textsc{Assign}$$

$$\frac{\Gamma \vdash \{U\}(\psi \rightarrow [\alpha_1\ ;\ \ldots]\phi), \Delta \qquad \Gamma \vdash \{U\}(\neg\psi \rightarrow [\alpha_2\ ;\ \ldots]\phi), \Delta}{\Gamma \vdash \{U\}[\texttt{if}\ (\psi)\ \alpha_1\ \texttt{else}\ \alpha_2\ ;\ \ldots]\phi, \Delta}\ \textsc{If}$$

$$\frac{\begin{array}{l}\Gamma \vdash \{U\}\mathit{Inv}, \Delta \\ \Gamma \vdash \{U\}\{A_1\}\,(\mathit{Inv} \wedge \psi \rightarrow [\alpha]\,\mathit{Inv}), \Delta \\ \Gamma \vdash \{U\}\{A_2\}\,(\mathit{Inv} \wedge \neg\psi \rightarrow [\ldots]\,\phi), \Delta\end{array}}{\Gamma \vdash \{U\}[\texttt{while}\ (\psi)\ \alpha\ ;\ \ldots]\phi, \Delta}\ \textsc{While}$$

(A_1, A_2 are fresh anonymising updates for α w.r.t. U, ϕ, Inv)

Fig. 1. Sequent calculus for WhileDL. In the last four rules, the update $\{U\}$ can also be empty and disappear.

$$\begin{array}{rcll}
\{v_1 := t_1, \ldots, v_k := t_k\}\ v_i & \rightarrow & t_i & \text{if } v_i \notin \{v_{i+1}, \ldots, v_k\} \\
\{v_1 := t_1, \ldots, v_k := t_k\}\ t & \rightarrow & t & \text{if } v_1, \ldots, v_k \text{ do not occur in } t \\
\{U\}\ f(t_1, \ldots, t_n) & \rightarrow & f(\{U\}\ t_1, \ldots, \{U\}\ t_n) & \\
\{U\}\ p(t_1, \ldots, t_n) & \rightarrow & p(\{U\}\ t_1, \ldots, \{U\}\ t_n) & \\
\{U\}\ \neg\phi & \rightarrow & \neg\{U\}\ \phi & \\
\{U\}\ (\phi \wedge \psi) & \rightarrow & \{U\}\ \phi \wedge \{U\}\ \psi & \\
\{U\}\ \{v_1 := t_1, \ldots, v_k := t_k\}\ \phi & \rightarrow & \{U,\ v_1 := \{U\}\ t_1,\ \ldots,\ v_k := \{U\}\ t_k\}\ \phi & \\
\{\ldots, v := s, \ldots, v := t, \ldots\}\ \phi & \rightarrow & \{\ldots, v := t, \ldots\}\ \phi &
\end{array}$$

Fig. 2. The main application rules for updates in WhileDL. Further rules to simplify updates can be formulated, but are not shown here [8, Chap. 3].

$$\frac{[\,s = t\,]}{\Gamma \vdash s = t, \Delta}\ =\textsc{-right} \qquad \frac{[\,s \le t\,]}{\Gamma \vdash s \le t, \Delta}\ \le\textsc{-right} \qquad \frac{[\,s \ge t\,]}{\Gamma \vdash s \ge t, \Delta}\ \ge\textsc{-right}$$

$$\frac{[\,s \ne t\,]}{\Gamma, s = t \vdash \Delta}\ =\textsc{-left} \qquad \frac{[\,s > t\,]}{\Gamma, s \le t \vdash \Delta}\ \le\textsc{-left} \qquad \frac{[\,s < t\,]}{\Gamma, s \ge t \vdash \Delta}\ \ge\textsc{-left}$$

Fig. 3. Closure rules for the WhileDL sequent calculus

Example. We illustrate the usage of the sequent calculus by proving the non-termination of the following program:

$$\text{LCM} \quad = \quad \begin{cases} a = a_0 \ ; \ b = b_0 \ ; \\ \texttt{while} \ (a \neq b) \ \{ \\ \quad\quad \texttt{if} \ (a > b) \ b = b + b_0 \ \texttt{else} \ a = a + a_0 \\ \} \end{cases}$$

In case of termination, the post-value of a and b is the least common multiple of the two integers a_0, b_0. The program fails, however, to handle negative inputs correctly: if the signs of a_0 and b_0 are different, for instance, the program does not terminate. To prove this formally, we instantiate (1) with LCM and construct a proof tree (Fig. 4).

The only step in the course of the proof that requires creativity is the choice of the formula *Inv* that is used as invariant when applying the rule WHILE (our technique for synthesising such formulae is described in the next section). In terms of the program execution, *Inv* has to describe a set of program states that (i) is entered when LCM reaches the while-loop, (ii) is not left during the execution of the loop, and (iii) does not contain any states in which the loop guard becomes false. We chose $a < b$ as invariant in this example, but similar proofs can be given for the invariants $a < 0 \wedge b > 0$ or $a \neq b$. In all cases, the technique of incremental closure has to be used to determine some initial state (i.e., values of the variables a_0, b_0) for which the chosen formula *Inv* actually *is* an invariant and the proof can be closed. The closing constraint in Fig. 4 is $[\, x_a < 1 \wedge x_a < x_b \,]$, which means that we have proven the non-termination for initial states (a_0, b_0) like $(0, 1), (0, 2), (-10, -5)$, etc.

4 Automatically Detecting Non-Termination

In our work, we developed an algorithm to identify non-terminating programs automatically. It has two components, an invariant generator and a theorem prover. The theorem prover is used to prove formulae that state the non-termination of a program. This done by construction of proof trees using the calculus rules and incremental closure, described in Sect. 3. The other component, the invariant generator, is used to provide and refine invariants for the theorem prover. It was used to construct the invariant $a < b$ from the previous section in a systematic way.

The idea of the algorithm is to construct a non-termination proof as described in the preceding section. The essential part of a non-termination proof is the invariant which is used in the application of the WHILE rule. Our algorithm tries to find this invariant by repeatedly constructing proof attempts. In each iteration a different invariant is used, starting with the formula *true*, representing that the prover has no knowledge about the invariant at start up. After each failed proof attempt, the incomplete proof tree is examined. The retrieved information from this examination is then used to refine the invariant. There are several ways of refinement of which one uses template variables for the invariants.

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{
    \dfrac{
     \dfrac{
      \dfrac{
       \dfrac{[\,x_a < x_b\,]}{x_a \geq x_b \;\vdash}\;{\geq}\text{-LEFT}
      }{\vdash\; x_a < x_b}\;(*)
     }{\vdash\; \{\,U_0, a := x_a, b := x_b\,\}\; a < b}\;\overset{*}{\to}
     \qquad \textit{Inv. Preservation} \qquad \textit{Inv. Usage}
    }{\vdash\; \{\,U_0, a := x_a, b := x_b\,\}\,[\mathtt{while}\ (a \neq b)\ \beta]\,\mathit{false}}\;\textsc{While}
   }{\vdash\; \{\,U_0, a := x_a, b := x_2\,\}\,[b = b_0\ ;\ \mathtt{while}\ (a \neq b)\ \beta]\,\mathit{false}}\;\textsc{Assign}, \overset{*}{\to}
  }{\vdash\; \{\,U_0, a := x_1, b := x_2\,\}\,\{\,a := a_0\,\}\,[b = b_0\ ;\ \mathtt{while}\ (a \neq b)\ \beta]\,\mathit{false}}\;\overset{*}{\to}
 }{\vdash\; \{\,a_0 := x_a, b_0 := x_b, a := x_1, b := x_2\,\}\,[a = a_0\ ;\ b = b_0\ ;\ \ldots]\,\mathit{false}}\;\textsc{Assign}
}{\vdash\; \{\,a_0 := x_a, b_0 := x_b, a := x_1, b := x_2\,\}\,[\textsc{Lcm}]\,\mathit{false}}
$$

$$
\dfrac{
 \dfrac{
  \dfrac{
   \overset{*}{\vdots}\;(*)
   \qquad
   \dfrac{
    \dfrac{
     \dfrac{
      \dfrac{
       \dfrac{
        \dfrac{[\,x_a < 1\,]}{f_a \leq f_b - 1, f_a \geq f_b - x_a, x_a \geq 1 \;\vdash}\;{\geq}\text{-LEFT}
       }{f_a \leq f_b - 1 \;\vdash\; f_a + x_a < f_b}\;(*)
      }{f_a \leq f_b - 1 \;\vdash\; \{\,U_0, a := f_a + x_a, b := f_b\,\}\,[\,]\,a < b}\;\textsc{Skip}, \overset{*}{\to}
     }{f_a \leq f_b - 1 \;\vdash\; \{\,U_0, a := f_a, b := f_b\,\}\,[a = a + a_0\,]\,a < b}\;\textsc{Assign}, \overset{*}{\to}
    }{f_a \leq f_b - 1 \;\vdash\; f_a \not> f_b \to \{\,U_0, a := f_a, b := f_b\,\}\,[a = a + a_0\,]\,a < b}\;(*)
   }{f_a \leq f_b - 1 \;\vdash\; \{\,U_0, a := f_a, b := f_b\,\}\,(a \not> b \to [a = a + a_0\,]\,a < b)}\;\overset{*}{\to}
  }{f_a \leq f_b - 1 \;\vdash\; \{\,U_0, a := f_a, b := f_b\,\}\,[\mathtt{if}\ (a > b)\ \ldots\ \mathtt{else}\ \ldots]\,a < b}\;\textsc{If}
 }{\vdash\; f_a < f_b \wedge f_a \neq f_b \to \{\,U_0, a := f_a, b := f_b\,\}\,[\beta]\,a < b}\;(*)
}{
 \dfrac{\vdash\; \{\,U_0, a := x_a, b := x_b\,\}\,\{\,a := f_a, b := f_b\,\}\,(a < b \wedge a \neq b \to [\beta]\,a < b)}{\textit{Inv. Preservation}}
}\;\overset{*}{\to}
$$

$$
\dfrac{
 \dfrac{*}{\vdash\; g_a < g_b \wedge g_a = g_b \to \{\,U_0, a := g_a, b := g_b\,\}\,[\,]\,\mathit{false}}\;(*)
}{
 \dfrac{\vdash\; \{\,U_0, a := x_a, b := x_b\,\}\,\{\,a := g_a, b := g_b\,\}\,(a < b \wedge a = b \to [\,]\,\mathit{false})}{\textit{Inv. Usage}}
}\;\overset{*}{\to}
$$

Fig. 4. Proof for the (potential) non-termination of the program Lcm using the invariant $a < b$. The proof can be closed with the constraint $[\,x_a < 1 \wedge x_a < x_b\,]$, which describes a set of initial states that causes Lcm to diverge. We write β for the body of the while-loop, f_a, f_b, g_a, g_b as abbreviation for the Skolem terms $f_a(x_a, x_b)$, $f_b(x_a, x_b)$, $g_a(x_a, x_b)$, $g_b(x_a, x_b)$, and U_0 as abbreviation for the update $a_0 := x_a, b_0 := x_b$. Rewriting steps to apply updates are denoted by $\overset{*}{\to}$, whereas $(*)$ means that rules for propositional and arithmetic reasoning are applied which are not shown in detail.

It.	cur. Inv.	Open goals	Queue after step 5 of algorithm
1	*true*	$a = b \;\vdash$	$b > a$, $b < a$, $b < U_b$, $a < U_a$, $b > L_b$, $a > L_a$, $a \neq b$
2	$b > a$	none	$b < a$, $b < U_b$, $\ldots$

Fig. 5. Application of the algorithm on Lcm. Technically, a and b in the open goals are Skolem terms like $f_a(x_a, x_b)$ in Fig. 4, which have to be translated back to obtain invariants in terms of the program variables. In iteration 2, the non-termination proof can be closed with the constraint $[\,x_a < x_b \wedge x_a < 1\,]$. The result expresses that Lcm does not terminate if the initial value of a_0 is less than that of b_0 and not positive.

A positive result of the algorithm is a successful non-termination proof of the program together with a description of a set of input values for which the loop of the program runs forever.

Note on Nested Loops. The algorithm as it is described here is only applicable to single, unnested loops. As it is always possible to transform nested loops into unnested ones, this is no real restriction. Besides, in [5] we describe how our algorithm can be adapted so that it directly works on nested loops.

Outline of the Algorithm. Let α be the program whose termination is in question. The input of the algorithm is α's source code, which is inserted into a WhileDL formula ϕ (formula (1) in Sect. 3) that states that there are inputs for which α does not terminate.

Initialisation

1. The formula ϕ is handed over to the theorem prover. The proof procedure is invoked and constructs a proof tree in which the program is symbolically executed until the execution reaches the loop.

Iteration

2. The proof procedure applies the invariant rule While (Fig. 1). The invariant Inv_{cur} which is used in the invariant rule's application is chosen from a queue of invariants. Initially there is only $Inv_{cur} \equiv true$ in the queue.
3. The proof procedure keeps on constructing the proof as far as possible without human interaction.
4. If the proof procedure can close the proof, the algorithm terminates with the result that the program does not terminate. If the proof cannot be closed, the open goals of the proof are extracted and handed over to the invariant generator.
5. The invariant generator inspects the formulae of the open goals. The obtained information is used to refine the invariant candidate to create one or more new candidates, which are then added to the queue.

The algorithm repeats step 2 to 5 iteratively, each time using one of the invariant candidates from the invariant queue. The iterations are carried out until one of these events occurs: the proof can be closed with the help of the invariant candidate, the algorithm runs out of new invariant candidates or a maximum number of iterations is reached. In case of a successful termination of the algorithm, it outputs the invariant used for the final proof, together with the (consistent) closing constraint.

There are three parts of step 5 of the algorithm that we like to describe in more detail. The first is the actual creation of the invariants.

Invariant Creation. There are different methods to create new invariants from the open goals of failed proofs. Assume that we obtained the open goal

$$\phi_1, \ldots, \phi_n \vdash \psi_1, \ldots \psi_m$$

where ϕ_i and ψ_i are WhileDL-formulae. Given such an open goal, the invariant generator creates invariant fragments ρ which are conjunctively added to the invariant Inv_{cur} which was used in the current iteration to obtain a new invariant $Inv_{new} = Inv_{cur} \wedge \rho$. The invariant fragments are created by the following operations:

- Add. A formula ψ_i in the succedent states a situation in which there is a problem with the non-termination proof when ψ_i does not hold. Most often that means that in this situation the loop actually terminates. We exclude this situation by setting $\rho = \psi_i$.
- NegAdd. A formula ϕ_i in the antecedent means that there is a problem with the non-termination in the situation where ϕ_i holds. Here, the same idea applies as for formulae in the succedent, but in this case we have to negate it before we add it to the old invariant, which means $\rho = \neg\phi_i$.
- Ineq. In case a formula ϕ_i of the antecedent is of form $\phi_i \equiv a = b$, we do not only add the negation as in negAndAdd, but an inequality. That means from $a = b$ we obtain two fragments $\rho_1 \equiv a \geq b$ and $\rho_2 \equiv a \leq b$, yielding two different new invariants.
- IneqVar. Often it is useful to express that *there are* upper or lower bounds for an expression rather than specifically setting one like in Ineq. This is done through the introduction of free logical variables. Those variables stand for particular but not yet specified values. For each term in the open goal, we provide two new variables U and L, one for the upper and one for the lower bound. Thus, for each term t_k occurring in one ϕ_i, we obtain two fragments $\rho_k^u \equiv t_k \leq U_k$ and $\rho_k^l \equiv t_k \geq L_k$. The values for the new variables are estimated by the constraint solver of the proof procedure.

The latter two creation methods are of course only applicable if a, b and t_k are expressions of an ordered type, in our case integers.

Invariant Filtering. In the process of invariant creation, sometimes invariant candidates are created that are not helpful in the search of a non-termination invariant. This is due to the fact that these methods are applied "blindly" without actually examining the old invariant candidate. Therefore, after the creation of invariants in step 5 of the algorithm, we filter out those candidates which are obviously useless:

- *Inconsistent Invariant.* A newly created invariant candidate can be equivalent to the formula *false*. Because the first property of non-termination invariants is that the invariant must hold before the loop execution, it is dismissed.

– *Equivalence to Previous Invariants.* A new invariant candidate can be equivalent to a candidate that was already created and/or used in an earlier iteration. Dismissal of these candidates avoid unnecessary calculations and thus save resources.
– *Impossible Closure of the Init-branch.* The application of the invariant rule makes the proof branch into three branches. The first branch proves that the invariant holds when the loop is reached in the execution of the program. In the refinement process, invariant candidates might be created that do not hold in the beginning of the loop, even if they are satisfiable in general. Once we have created an invariant candidate which prevents the first branch from closing, it does not make sense to refine any further: refinement would only strengthen the candidate even more.[3]
– *Complexity.* For performance reasons, we set a limit on the complexity of formulae to keep the runtime at a reasonable level.

Invariant Scoring. In each iteration of the algorithm, when the invariant candidates are created and filtered in step 5 still a lot of invariants can remain. In order to traverse the search space of invariants in a reasonable way, we have to queue invariants according to their probable usefulness for non-termination proofs.

We estimate this usefulness by several criteria and express it in a score, which is a real number between 0 and 1. The lower the score is, the more the invariant is preferred in the queue. The score is calculated as a weighted average of scores for each of the following criteria.

– *Complexity.* In order to find the most general description of a set of critical inputs, we prefer simple invariants to complex ones. The complexity is measured in both the term depth and the number of operators of the invariant.
– *Existence of Free Variables.* The creation method IneqVar is a strong tool (and sometimes the only effective one) to find the desired invariant. The problem with free variables is that in cases where they do *not* lead to a closed proof, they tend to lead to even bigger open proofs. It is reasonable to prefer invariant candidates that do not contain free variables to those who do in order to keep the number of newly created candidates as low as possible.
– *Multiple Occurrence of Formulae.* In an open proof, sometimes the same formulae occur in several open goals. We prefer invariant candidates made from those formulae to others, because if the candidate makes the algorithm close branches, it will close several branches in the same proof.
– *Reoccurring Formulae.* Formulae which occurred in open proofs in several iterations of the algorithm might be suitable candidates for the next invariant, because they hint to situations where the non-termination proof repeatedly failed.

[3] The filtering of inconsistent invariants is subsumed in this filter. We kept it in the list of filters, because checking for inconsistency is easier than for closure of the initial branch. So, for performance reasons it is useful to first check only for consistency before examining the closability of the first branch.

– *Proof Size.* We presume that the smaller an open proof is (measured in the number of open goals) the closer it is to being closed. Therefore we prefer formulae which come from small open proofs to those from big open proofs.

Experiments have shown that the choice and weighting of the criteria is extremely important for the search in the space of invariants. In our work, we ran several experiments to test the impact of different heuristics, the results of which are given in Sect. 5 and [5].

Examples. We apply our algorithm to the example programs Fib and Lcm, of which the latter one was introduced in Sect. 3 already. For the sake of simplicity, we assume that for scoring of the invariants only the criteria of complexity is applied.

Example Lcm. Fig. 5 shows how the algorithm works on Lcm. In this case all presented creation methods are used.

Example Fib. Given a Fibonacci number n as input, Fib calculates how many calculation steps are necessary in the series of Fibonacci numbers to reach n. The result is stored in variable c. In case n is not a Fibonacci number, the program does not terminate.

$$\textsc{Fib} \quad = \quad \begin{cases} i = 0\ ;\ j = 1\ ;\ t = 0\ ;\ c = 2\ ; \\ \texttt{while}\ (j \neq n)\ \{ \\ \qquad t = j + i\ ;\ i = j\ ;\ j = t\ ;\ c = c + 1 \\ \} \end{cases}$$

In contrast to Lcm, the algorithm needs several refinement steps (Fig. 6) to prove the non-termination of Fib. The input variable n is associated with the free logical variable x_n. This time, we used only the creation methods Add, NegAdd, Ineq, together with the complexity scoring criterion. We abstained from showing the creation method IneqVar, because it increases the number of necessary iterations too much to be shown here. However, we did run the same experiment with IneqVar and will present the results in the following section.

Properties of the Algorithm. We would like to have a closer look at the properties of the algorithm which we presented here.

– *Soundness.* The algorithm is sound for non-termination: it will never identify a terminating program as non-terminating. This is an immediate consequence of the soundness of the calculus from Sect. 3, because non-termination is only reported if it was possible to construct a proof for it. Applied to a terminating program, the algorithm will fail to find such a proof and will output that it was not able to prove non-termination.

- *Incompleteness.* Unfortunately, but expectedly, both our calculus and the algorithm are not complete for non-termination: there are programs that do not terminate for some inputs, but there is no proof of this fact in the calculus from Sect. 3. This is implied by the soundness, because the set of programs that do not terminate for some inputs is not recursively enumerable.[4] Because the algorithm is based on heuristics, it might also fail to find existing non-termination proofs for a program, of course.
- *Automation.* The algorithm works fully automatic, in the sense that no manual "human" actions are necessary to obtain the results.
- *Determinism.* The algorithm is deterministic, because for the same input it always produces the same results. The indeterministic calculus which forms the base of the prover is made deterministic by choice of heuristics and prioritisation.
- *Termination.* Our algorithm itself always terminates. This is ensured by setting an upper limit for the number of iterations, and by limiting the size of proofs in the calculus from Sect. 3 that are constructed. Of course these limits have to be chosen carefully, because the lower they are the fewer non-terminating programs can be identified.

5 Experiments

To evaluate our method, we implemented the algorithm presented in Sect. 4 in the Java programming language: we wrote an invariant generator as a standalone tool, in which the KeY theorem prover [8] was embedded to take care of the proofs in dynamic logic. Since there was no publicly available standardised example set of non-terminating programs, we also built up such a set to estimate the quality of our approach and to test different heuristics.

Example Set. Our example set consists of 55 programs, of which 53 are known to be non-terminating for all or some input values, one whose termination behaviour is not fully known, and one which is terminating. All programs are written in a fragment of Java that captures the functionality of the While language from Sect. 2. The programs operate on between one and five variables and consist of up to 25 lines of code each. The selection of programs was made with the goal to cover both typical programming errors and particularly tricky (non-)termination behaviour.

Results of the Experiments. We tested different settings for creation and scoring of invariants in several experiments [5]. Our software could solve 41 of the 55 examples automatically, but not more than 37 with one setting. This fact shows how sensitive the algorithm's heuristics are.

Some of the experiments were used to estimate the usefulness of the different creation methods of Sect. 4, in particular the method IneqVar. When including

[4] Note that the set of programs that terminate for *all* possible inputs is not recursively enumerable either.

It.	cur. Inv.	Open goals
1	$Inv_1 \equiv true$	$j = x \vdash$
2	$Inv_2 \equiv j > x$	$x \geq 1 \vdash\ ,\ j \leq x - i, i \leq -1, j \geq 1 + x \vdash$
3	$Inv_3 \equiv j < x$	$x \leq 1 \vdash\ ,\ i \geq 1, j \geq x - i, j \leq -x \vdash$
4	$Inv_4 \equiv j \neq x$	$x = 1 \vdash\ ,\ j = x - i, i = 0 \vdash$
5	$Inv_5 \equiv j > x \wedge x < 1$	$x \geq 1 \vdash\ ,\ j \leq x - i, x \leq 0, i \leq -1, j \geq 1 + x \vdash$
6	$Inv_6 \equiv j > x \wedge x > -1$	none

The next invariants to be tried:

$Inv_7 \equiv j < x \wedge x > 1$	$Inv_{14} \equiv j \neq x \wedge j > x - i$
$Inv_8 \equiv j < x \wedge i < 1$	$Inv_{15} \equiv j > x \wedge x < 1 \wedge x > 0$
$Inv_9 \equiv j \neq x \wedge i = 0$	$Inv_{16} \equiv j > x \wedge j > x - i$
$Inv_{10} \equiv j \neq x \wedge x > 1$	$Inv_{17} \equiv j < x \wedge j < x - i$
$Inv_{11} \equiv j \neq x \wedge x < 1$	$Inv_{18} \equiv j \neq x \wedge j \neq x - i$
$Inv_{12} \equiv j \neq x \wedge x \neq 1$	$Inv_{19} \equiv j > x \wedge x < 1 \wedge j > x - i$
$Inv_{13} \equiv j > x \wedge x < 1 \wedge i > -1$	

Fig. 6. Application of the algorithm on example FIB. Again, technically, i and j in the open goals are Skolem terms like $f_a(x_a, x_b)$ in Fig. 4. In iteration no. 6, the non-termination proof can be closed with the constraint $[\,x_n < 1 \wedge -2 < x_n\,]$ for the free variables. This result expresses that for n being 0 or -1, FIB does not terminate. The following invariants were dismissed by the filters because of inconsistency: $j > x_n \wedge j < 1 + x_n$, $j < x_n \wedge j > x_n - 1$, and $j > x_n \wedge x_n < 1 \wedge j < 1 + x_n$.

this creation method, and thus invariant templates with free logical variables, about 20% more problems could be solved than without the method. This shows that free variables are a strong tool to conduct non-termination proofs. Unfortunately, proof attempts that use invariants with free variables also tend to be considerably larger and more complex than those without, and more iterations are needed in the algorithm. This means that the algorithm is slowed down by free variables in those cases in which they are not strictly necessary to find proofs. The average number of iterations in successful cases (in which a suitable invariant was found) lay between 1.5 and 3.5 depending on the heuristics.

The example LCM of Sect. 4 was solved in all experiments and needed between 2 and 8 iterations. The example FIB was solved by some of the experiments with an iteration number between 6 and 39. The best run is illustrated in Fig. 6. The number of iterations increases if the creation method INEQVAR is used (depending on the heuristics). An invariant which was found in this case is $j > L_j \wedge i > L_i$, where the proof was closed with the (simplified) constraint $[\,L_j = -1 \wedge L_i = -1 \wedge x < 0\,]$. Invariant and constraint describe the situation where the input value n is negative and the variables j and i are non-negative.

The example set and the implementation of the software is publicly available at `http://www.key-project.org/nonTermination/`.

6 Related Work

Although the development of *termination checkers* is a flourishing research subject, we only know of two methods (and implementations) that are directly comparable to the *non-termination* analysis presented in this paper:

The more similar approach is [12], which uses concolic program execution to search for lassos (loops) in a program, and constraint solving for proving the feasibility of lassos. The latter part is similar to the invariant generation method shown in the present paper, but it does not make use of counterexamples to refine invariant candidates. Because we use purely symbolic reasoning to determine critical initial program states, it can also be expected that our approach is able to derive more general descriptions of such input states than [12], at the cost of being less scalable.

Secondly, the AProVE system [13] is able to prove both the termination and non-termination of term rewrite systems [14] and is in principle also applicable to imperative programs: such programs can be analysed after a suitable translation to rewrite systems [2]. So far, existing translations are incomplete, however, which means that the resulting rewrite system might be non-terminating even if the original program is terminating.

Construction of invariants using invariant templates and constraint solving is an approach that is employed in many contexts, e.g., [15, 16]. The principle is usually not embedded in a program logic as it is done in the present paper.

The iterative refinement of invariants described in this paper has some similarities to iterative backwards-propagation of assertions, which is described in [17] but can, in some form or another, be found in many static program analysis techniques.

7 Conclusion and Future Work

We have introduced a novel approach to automated detection of non-termination defects in software programs. The approach is built on the basis of a sequent calculus for dynamic logic and works by generating invariants that prove the unreachability of terminating states. In experiments, the majority of our example programs could automatically be proven non-terminating. Furthermore, when experimenting with more complex non-terminating Java programs [5], we found that also here it is often possible to find small and simple invariants that witness non-termination. The intuitive explanation for this is that (i) the usage of the invariant rule WHILE (with anonymising updates) allows to ignore those parts of the program state that are not changed in the loop, and that (ii) the precise character of state changes caused by a loop can be ignored in the invariant as well, as long as non-termination is preserved. Although further investigations concerning such programs are necessary, this indicates that our method is also applicable to programs that operate on heap data structures.

When moving from the while-language to actual Java-like programs, one modification of the algorithm that appears helpful is to automatically add heapwellformedness conditions to the invariant candidates. Partly, this is a consequence of using dynamic logic for Java [8, Sect. 3], in which properties like "attributes of allocated objects only point to allocated objects" are non-trivial and can be difficult to synthesise for the invariant generator. Another aspect that becomes more central with Java programs is the detection of the variables and heap locations that a loop can assign to. It might be useful to determine also these locations incrementally and simultaneously with the loop invariant, based on failed proof attempts.

As a prerequisite for more extensive experiments, we want to develop an implementation of our non-termination checker that is more tightly integrated with the program verification tool used. This way, we expect to achieve a significantly higher performance. On the more theoretic level, we are in the process of investigating the usage of closure constraints (Sect. 3) more systematically in order to define fragments of first-order logic with integer arithmetic for which the calculus is complete, and in order to further develop the approach.

Acknowledgements

We like to thank the following people for constructive feedback and support: Richard Bubel, Prof. Reiner Hähnle, Mattias Ulbrich, Benjamin Weiss and all others of the KeY-group. Besides we like to thank Prof. Giesl for supporting the diploma thesis which was the base for this paper.

References

1. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Proceedings, 18th International Conference on Computer-Aided Verification, Seattle, USA. Volume 4144 of LNCS, Springer (2006) 415–418
2. Sondermann, M.: Automatische Terminierungsanalyse von imperativen Programmen. Master's thesis, RWTH Aachen University, Aachen, Germany (2006)
3. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In Gurevich, Y., Meyer, B., eds.: Proceedings, First International Conference on Tests and Proofs, Zurich, Switzerland. Volume 4454 of LNCS, Springer (2007) 41–60
4. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
5. Velroyen, H.: Automatic non-termination analysis of imperative programs. Master's thesis, Chalmers University of Technology, Aachen Technical University, Göteborg, Sweden and Aachen, Germany (2007)
6. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge, Massachusetts (1993)
7. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
8. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2007)
9. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In Hermann, M., Voronkov, A., eds.: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia. Volume 4246 of LNCS, Springer (2006) 422–436

10. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In Beckert, B., ed.: Proceedings, Fourth International Verification Workshop, Bremen, Germany. Volume 259 of CEUR (`http://ceur-ws.org/`) (2007)
11. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, First International Joint Conference on Automated Reasoning, Siena, Italy. Volume 2083 of LNCS, Springer (2001) 545–560
12. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In Necula, G.C., Wadler, P., eds.: ACM Symposium on Principles of Programming Languages, San Francisco, USA, ACM Press (2008) 147–158
13. Giesl, J., Schneider-Kamp, P., Thiemann, R.: Aprove 1.2: Automatic termination proofs in the dependency pair framework. In Furbach, U., Shankar, N., eds.: Proceedings, Third International Joint Conference on Automated Reasoning, Seattle, USA. Volume 4130 of LNCS, Springer (2006) 281–286
14. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In Gramlich, B., ed.: Proceedings, 5th International Workshop on Frontiers of Combining Systems, Vienna, Austria. Volume 3717 of LNCS, Springer (2005) 216–231
15. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In Baader, F., Baumgartner, P., Nieuwenhuis, R., Voronkov, A., eds.: Deduction and Applications. Number 05431 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany (2006)
16. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Proceedings, 15th International Conference on Computer-Aided Verification, Boulder, USA. Volume 2725 of LNCS, Springer (2003) 420–432
17. Bjørner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theoretical Computer Science **173** (1997) 49–87

Paper 3

Verifying Object-Oriented Programs with KeY: A Tutorial

Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H. Schmitt

Post-proceedings of the 5th International Symposium on Formal Methods for Components and Objects
Amsterdam, The Netherlands, 2006
Volume 4709 of LNCS, Springer (2007) 70–101

The paper has been updated to cover the upcoming version 1.4 of KeY.

Verifying Object-Oriented Programs with KeY: A Tutorial

Wolfgang Ahrendt[1], Bernhard Beckert[2], Reiner Hähnle[1], Philipp Rümmer[1], and Peter H. Schmitt[3]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
`ahrendt|reiner|philipp@chalmers.se`
[2] Department of Computer Science,
University of Koblenz-Landau
`beckert@uni-koblenz.de`
[3] Department of Theoretical Computer Science,
University of Karlsruhe
`pschmitt@ira.uka.de`

Abstract. This paper is a tutorial on performing formal specification and semi-automatic verification of Java programs with the formal software development tool KeY. This tutorial aims to fill the gap between elementary introductions using toy examples and state-of-art case studies by going through a self-contained, yet non-trivial, example. It is hoped that this contributes to explain the problems encountered in verification of imperative, object-oriented programs to a readership outside the limited community of active researchers.

1 Introduction

The KeY system is the main software product of the KeY project, a joint effort between the University of Karlsruhe, Chalmers University of Technology in Göteborg, and the University of Koblenz. The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.

This paper is a tutorial on performing formal specification and semi-automatic verification of Java programs with KeY. There is already a tutorial introduction to the KeY prover that is set at the beginner's level and presupposes no knowledge of specification languages, program logic, or theorem proving [1, Chapt. 10]. At the other end of the spectrum are descriptions of rather advanced case studies [1, Chapt. 14 and 15] that are far from being self-contained. The present tutorial intends to fill the gap between first steps using toy examples and state-of-art case studies by going through a self-contained, yet non-trivial, example. We found few precisely documented and explained, yet realistic, case studies even for other verification systems. Therefore, we believe that this tutorial is of interest in its own right, not only for those who want to know about KeY.

We hope that it can contribute to explain the problems encountered in verification of imperative, object-oriented programs to a readership outside the limited community of active researchers.

We assume that the reader is familiar with the Java programming language, with first-order logic and has some experience in formal specification and verification of software, presumably using different approaches than KeY. Specifications in the Java Modeling Language (JML) [2] and expressions in KeY's program logic Java Card DL [1, Chapt. 3] are explained as far as needed.

In this tutorial we demonstrate in detail how to specify and verify a Java application that uses most object-oriented and imperative features of the Java language. The presentation is such that the reader can trace and understand almost all aspects. To this end, we provided the complete source code and specifications at `www.key-project.org/fmco06`. We strongly encourage reading this paper next to a computer with a running KeY system. The descriptions of this paper refer to the upcoming version 1.4 of KeY, which is available under GPL and can be freely downloaded from `www.key-project.org`. Information on how to install the KeY tool can also be found on that web site.

The tutorial is organised as follows: in Section 2 we provide some background on the architecture and technologies employed in the KeY system. In Section 3 we describe the case study that is used throughout the remaining paper. It is impossible to discuss all verification tasks arising from the case study. Therefore, in Section 4, we walk through a typical proof obligation (inserting an element into a datastructure) in detail including the source code, the formal specification of a functional property in JML, and, finally, the verification proof. In Section 5 we repeat this process with a more difficult proof obligation. This time around, we abstract away from most features learned in the previous section in favour of discussing some advanced topics, in particular complex specifications written in Java Card DL, handling of complex loops, and proof modularisation with method contracts. We conclude with a brief discussion.

2 The KeY Approach

The KeY Program Verification System. KeY supports several languages for specifying properties of object-oriented models. Many people working with UML and MDA have familiarity with the specification language OCL (Object Constraint Language), as part of UML 2.0. KeY can also translate OCL expressions to natural language (English and German). Another specification language supported by KeY, which enjoys popularity among Java developers and which we use in this paper, is the Java Modeling Language (JML). Optional plugins of KeY into the popular Eclipse IDE and the Borland Together CASE tool suite are available with the intention to lower initial adoption cost for users with no or little training in formal methods.

The target language for verification in KeY is Java Card 2.2.1. KeY is the only publicly available verification tool that supports the full Java Card standard including the persistent/transient memory model and atomic transactions. Rich specifications of the Java Card API are available both in OCL and JML. Java 1.4 programs that respect the limitations of Java Card (no floats, no concurrency, no dynamic class loading) can be verified as well.

The Eclipse and Together KeY plugins allow to select Java classes or methods that are annotated with formal specifications and both plugins offer to prove a number of correctness judgements such as behavioural subtyping, partial and total correctness, invariant preservation, or frame properties. In addition to the JML/OCL-based interfaces one may supply proof obligations directly on the level of Java Card DL. For this, a stand-alone version of the KeY prover not relying on Eclipse or Together is available.

The program logic Java Card DL is axiomatised in a *sequent calculus*. Those calculus rules that axiomatise program formulae define a symbolic execution engine for Java Card and so directly reflect the operational semantics. The calculus is written in a small domain-specific so-called *taclet* language that was designed for concise description of rules. Taclets specify not merely the logical content of a rule, but also the context and pragmatics of its application. They can be efficiently compiled not only into the rule engine, but also into the automation heuristics and into the GUI. Depending on the configuration, the axiomatisation of Java Card in the KeY prover uses 1000–1300 taclets.

The KeY system is not merely a verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved.

At the core of the KeY system is the deductive verification component, which also can be used as a stand-alone prover. It employs a free-variable sequent calculus for first-order Dynamic Logic for Java. The calculus is proof-confluent, i.e., no backtracking is necessary during proof search.

While we constantly strive to increase the degree of automation, user interaction remains indispensable in deductive program verification. The main design goal of the KeY prover is thus a seamless integration of automated and interactive proving. Efficiency must be measured in terms of user plus prover, not just prover alone. Therefore, a good user interface for presentation of proof states and rule application, a high level of automation, extensibility of the rule base, and a calculus without backtracking are all important features.

Syntax and Semantics of the KeY Logic. The foundation of the KeY logic is a typed first-order predicate logic with subtyping. This foundation is extended with parameterised modal operators $\langle p \rangle$ and $[p]$, where p can be any sequence of legal Java Card statements. The resulting multi-modal program logic is called Java Card Dynamic Logic or, for short, Java Card DL [1, Chapt. 3].

As is typical for Dynamic Logic, Java Card DL integrates programs and formulae within a single language. The modal operators refer to the final state of program p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state. For example, "when started in a state where `x` is zero, `x++;` terminates in a state where `x` is one" can be expressed as $\mathtt{x} \doteq 0 \rightarrow \langle \mathtt{x++;} \rangle (\mathtt{x} \doteq 1)$. The states used to interpret formulae are first-order structures sharing a common universe.

The type system of the KeY logic is designed to match the Java type system but can be used for other purposes as well. The logic includes *type casts* (changing the static type of a term) and *type predicates* (checking the dynamic type of a term) in order to reason about inheritance and polymorphism in Java programs [1, Chapt. 2]. The type hierarchy contains the types such as *boolean*, the root reference type Object, and the type Null, which is a subtype of all reference types. It contains a set of user-defined types, which are usually used to represent the interfaces and classes of a given Java Card program. Finally, it contains several integer types, including both the range-limited types of Java and the infinite integer type $\mathbb{Z}$.

Besides built-in symbols (such as type-cast functions, equality, and operations on integers), user-defined functions and predicates can be added to the signature. They can be either *rigid* or *non-rigid*. Intuitively, rigid symbols have the same meaning in all program states (e.g., the addition on integers), whereas the meaning of non-rigid symbols may differ from state to state.

Moreover, there is another kind of modal operators called *updates*. They can be seen as a language for describing program transitions. There are simple function updates corresponding to assignments in an imperative programming language, which in turn can be composed sequentially and used to form parallel or quantified updates. Updates play a central role in KeY: the verification calculus transforms Java Card programs into updates. KeY contains a powerful and efficient mechanism for simplifying updates and applying them to formulae.

Rule Formalisation and Application. The KeY system has an automated-proof-search mode and an interactive mode. The user can easily switch modes during the construction of a proof.

For interactive rule application, the KeY prover has an easy to use graphical user interface that is built around the idea of direct manipulation. To apply a rule, the user first selects a *focus of application* by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules applicable at this focus. This choice remains manageable even for very large rule bases. Rule schema variable instantiations are mostly inferred by matching. A simpler way to apply rules and give instantiations is by drag and drop. If the user drags an equation onto a term the system will try to rewrite the term with the equation. If the user drags a term onto a quantifier the system will try to instantiate the quantifier with this term.

The interaction style is closely related to the way rules are formalised in the KeY prover. There are no hard-coded rules; all rules are defined in the "taclet language" instead. Besides the conventional declarative semantics, taclets have a clear operational semantics, as the following example shows—a "modus ponens" rule in textbook notation (left) and as a taclet (right):

$$\frac{\phi, \psi, \Gamma \Longrightarrow \Delta}{\phi, \phi \rightarrow \psi, \Gamma \Longrightarrow \Delta}$$

```
\find (p -> q ==>)          // implication in antecedent
\assumes (p ==>)            // side condition
\replacewith(q ==>)         // action on focus
\heuristics(simplify)       // strategy information
```

The `find` clause specifies the potential application focus. The taclet will be offered to the user on selecting a matching focus and if a formula mentioned in the `assumes` clause is present in the sequent. The action clauses `replacewith` and `add` allow modifying (or deleting) the formula in focus, as well as adding additional formulae (not present here). The `heuristics` clause records information for the parameterised automated proof search strategy.

The taclet language is quickly mastered and makes the rule base easy to maintain and extend. Taclets can be proven correct against a set of base taclets [3]. A full account of the taclet language is given in [1, Chapt. 4 and Appendix B.3.3].

Applications. Among the major achievements using KeY in the field of program verification so far are the treatment of the Demoney case study, an electronic purse application provided by Trusted Logic S.A., and the verification of a Java implementation of the Schorr-Waite graph marking algorithm. This algorithm, originally developed for garbage collectors, has recently become a popular benchmark for program verification tools. Chapters 14 and 15 of the KeY book [1] are devoted to a detailed description of these case studies. A case study [4] performed within the HIJA project has verified the lateral module of the flight management system, a part of the on-board control software from Thales Avionics.

Lately we have applied the KeY system also on topics in security analysis [5], and in the area of model-based test case generation [6, 7] where, in particular, the prover is used to compute path conditions and to identify infeasible paths.

The flexibility of KeY w.r.t. the used logic and calculus manifests itself in the fact that the prover has been chosen as a reasoning engine for a variety of other purposes. These include the mechanisation of a logic for Abstract State Machines [8] and the implementation of a calculus for simplifying OCL constraints [9].

KeY is also very useful for teaching logic, deduction, and formal methods. Its graphical user interface makes KeY easy to use for students. They can step through proofs with different degrees of automation (using the full verification calculus or just the first-order core rules). The authors have been successfully teaching courses for several years using the KeY system. An overview and course material is available at `www.key-project.org/teaching`.

Related Tools. There exist a number of other verification systems for object-oriented programs. The KIV[4] tool [10] is closest to ours in that it is also interactive and also based on Dynamic Logic. Most other systems are based on

[4] `www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/`

a verification condition generator (VCG) architecture and separate the translation of programs into logic from the actual proof process. A very popular tool of this kind is ESC/Java2[5] (Extended Static Checker for Java2) [11], which uses the Simplify theorem prover [12] and attempts to find run-time errors in JML-annotated Java programs. ESC/Java2 compromises on completeness and even soundness for the sake of ease of use and scalability. Further systems are JACK [13], Krakatoa [14], LOOP [15], which can also generate verification conditions in higher order logic that may then be proved using interactive theorem provers like PVS, Coq, Isabel, etc. Like KeY, JACK, Krakatoa, and LOOP support JML specifications. With JACK, we moreover share the focus on smart card applications.

3 Verification Case Study: A Calendar Using Interval Trees

In this tutorial, we use a small Java calendar application to illustrate how specifications are written and programs are verified with the KeY system. The application provides typical functionality like creating new calendars, adding or removing appointments, notification services that inform about changes to a particular appointment or a calendar, and views for displaying a time period (like a particular day or month) or for more advanced lookup capabilities.

The class structure of the calendar application is shown in Fig. 1 and 2. It consists of two main packages: a datastructure layer `intervals` that provides classes for working with (multisets of) intervals, and a domain layer `calendar` that defines the actual logic of a calendar. Intervals (interface `Interval`) are the basic entities that our calendars are built upon. In an abstract sense, each entry or appointment in a calendar is primarily an interval spanned by its start and its end point in time. A calendar is a multiset of such intervals. For reasons of simplicity, we represent discrete points of time as integers, similarly to the time representation in Unix (the actual unit and offset are irrelevant here). Further, we use the *observer design pattern* (package `observerPattern`) for being able to observe all modifications that occur in a calendar entry.

Interval Datastructures. The most important lookup functionality that our calendar provides, is the ability to retrieve all entries that overlap a certain query time interval (i.e., have a point of time in common with the query interval). Such queries are used, for instance, when displaying all appointments for a particular day. We consequently store intervals in an *interval tree* datastructure [16] (class `IntervalTree` in Fig. 2), which allows to retrieve overlapping entries with logarithmic complexity in the size of the calendar. An interval tree is a binary tree, in which each node (class `IntervalTreeNode`) stores (a) the multiset of intervals that include a certain point (the `cutPoint`) and (b) pointers to the subtrees that handle the intervals strictly smaller (association `left`) resp. strictly bigger

[5] `http://secure.ucd.ie/products/opensource/ESCJava2/`

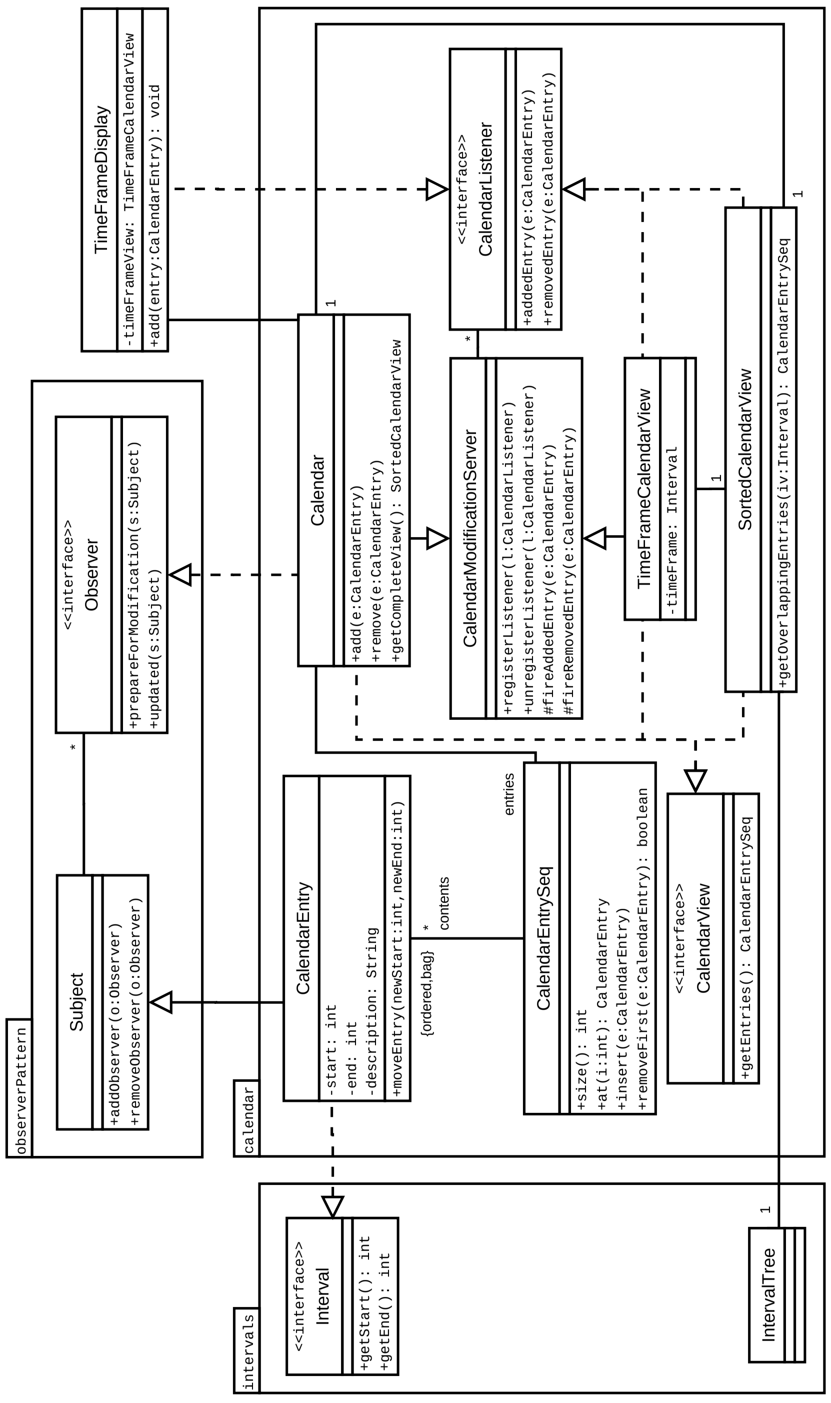

observerPattern
Subject
+addObserver(o:Observer)
+removeObserver(o:Observer)
*
<<interface>>
Observer
+prepareForModification(s:Subject)
+updated(s:Subject)
TimeFrameDisplay
-timeFrameView: TimeFrameCalendarView
+add(entry:CalendarEntry): void
intervals
<<interface>>
Interval
+getStart(): int
+getEnd(): int
IntervalTree
1
calendar
CalendarEntry
-start: int
-end: int
-description: String
+moveEntry(newStart:int,newEnd:int)
{ordered,bag}
*
contents
entries
CalendarEntrySeq
+size(): int
+at(i:int): CalendarEntry
+insert(e:CalendarEntry)
+removeFirst(e:CalendarEntry): boolean
<<interface>>
CalendarView
+getEntries(): CalendarEntrySeq
Calendar
+add(e:CalendarEntry)
+remove(e:CalendarEntry)
+getCompleteView(): SortedCalendarView
1
CalendarModificationServer
+registerListener(l:CalendarListener)
+unregisterListener(l:CalendarListener)
#fireAddedEntry(e:CalendarEntry)
#fireRemovedEntry(e:CalendarEntry)
*
<<interface>>
CalendarListener
+addedEntry(e:CalendarEntry)
+removedEntry(e:CalendarEntry)
TimeFrameCalendarView
-timeFrame: Interval
1
SortedCalendarView
+getOverlappingEntries(iv:Interval): CalendarEntrySeq
1

Fig. 1. The packages `observerPattern` and `calendar` of the calendar case study

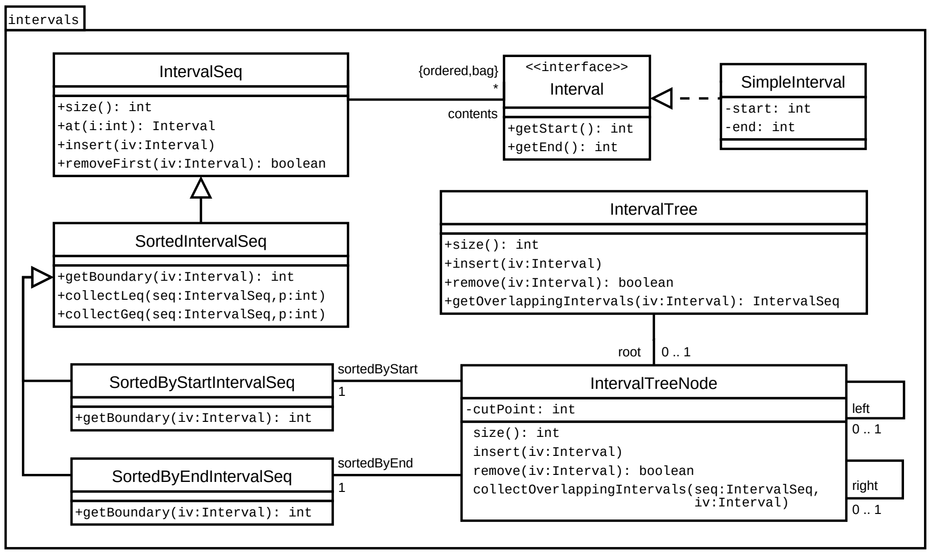


Fig. 2. The package `intervals` of the calendar case study

(association `right`) than the cut point. The intervals belonging to a particular node have to be stored both sorted by the start and by the end point, which is further discussed in Sect. 4.

Package Calendar. The two primary classes that implement a calendar are `CalendarEntry` for single appointments (an implementation of the interface `Interval`) and `Calendar` for whole calendars. The basic `Calendar` provides the interface `CalendarView` for accessing all entries that are part of the calendar in an unspecified order. A more advanced lookup interface, `SortedCalendarView`, can be accessed through the method `getCompleteView` of `Calendar`. It allows to retrieve all entries that overlap with a given interval. This interface is realised using the interval trees from package `intervals`.

A further view on calendars is `TimeFrameCalendarView`, which pre-selects all appointments within a given period of time, and which is based on the class `SortedCalendarView`. Both `Calendar` and `TimeFrameCalendarView` also provide a notification service (`CalendarModificationServer`) that informs about newly added and removed entries. We illustrate the usage of this service (and of `TimeFrameCalendarView`) in class `TimeFrameDisplay`, which is further discussed and verified in Sect. 5.

4 First Walk-through: Verifying Insertion into Interval Sequences

In this section, we zoom into a small part of the scenario described above, namely the `insert()` method belonging to the class `IntervalSeq` and its subclasses. In the context of that method, we demonstrate the different basic stages of formal software development with the KeY system. We discuss the *formal specification* of the `insert()` method, the generation of corresponding *proof obligations* in the used program logic, and the *formal verification* with the KeY prover. Along with demonstrating the basic work-flow, we introduce the used formalisms on the way, when they appear, but just to the extent which allows to follow the example. These formalisms are: the specification language JML (Java Modeling Language) [2], the program logic Java Card DL, and the corresponding calculus.[6]

As described in Section 3, the basic data structure of the case study scenario is a tree, the nodes of which are instances of the class `IntervalTreeNode`. Each such node contains one integer number (representing a point in time, the "cut point" of the node), and two interval sequences, both containing the same intervals (all of which contain the cut point of the node). The difference between the two sequences is that the contained intervals are sorted differently, once by their start, and once by their end.

Correspondingly, the two sequences contained in each node are instances of the classes `SortedByStartIntervalSeq` and `SortedByEndIntervalSeq`, respectively. Both are subclasses of `SortedIntervalSeq`, which in turn is a subclass of `IntervalSeq`. One of the basic methods provided by (instances of) these classes is `insert(Interval iv)`. In this section, we discuss, as an example, the implementation and specification of that method, as well as the verification of corresponding proof obligations.

The specification of the `insert()` method of the class `SortedIntervalSeq` also involves the superclass, `IntervalSeq`, because parts of the specification are *inherited* from there. Later, in the verification we will also be concerned with the two subclasses of `SortedIntervalSeq`, which provide different implementations of a method called by `insert()`, namely `getBoundary()`.

4.1 Formal Specification and Implementation

Within the Class `IntervalSeq`. This class is the topmost one in this small hierarchy, instances of which represent a sequence of intervals. Internally, the sequence is realised via an array `contents` of type `Interval[]`. This array can be longer than the actual `size` of the interval sequence. Thereby, we avoid having to allocate a new array at each and every increase of the sequence's `size`. Instead, `size` points to the index up to which we consider `contents` be filled with "real" intervals; only if `size` exceeds `contents.length`, a new array is allocated, into which the old one is copied. This case distinction is encapsulated in the method `incSize()`, to be called by `insert()`.

— Java (1.1) ———————————————————————

```java
protected void incSize() {
  ++size;
```

[6] All these are described in more detail in the KeY book [1]: JML in Section 5.3, Java Card DL and the calculus in Chapter 3.

```
  if ( size > contents.length ) {
    final Interval[] oldAr = contents;
    contents = new Interval[contents.length * 2];
    int i = 0;
    while (i < oldAr.length) {contents[i] = oldAr[i]; ++i;}
  }
}
```

——— Java ——

We turn to the actual `insert()` method now. The class `IntervalSeq` is ignorant of sorting, so all we require from `insert(iv)` is that `iv` is indeed inserted, *wherever*, in the sequence. To the very least, this means that, in a post state, `iv` is stored at *any* of the indices of `contents`. Using mathematical standard notation, we can write this as

$$\exists i.\ 0 \leq i \wedge i < \texttt{size} \wedge \texttt{contents}[i] = \texttt{iv}$$

Note that, already in this mathematical notation, we are mixing in elements from the programming language level, namely the instance field names, and the array access operator "`[ ]`". Now, the specification language JML takes this several steps further, using Java(like) syntax wherever possible: `<=` for $\leq$, `&&` for $\wedge$, `==` for $=$, `!=` for $\neq$, and so on. Special keywords are provided for concepts not covered by Java, like `\exists` for $\exists$. Altogether, the above formula is expressed in JML as:

```
  \exists int i; 0 <= i && i < size; contents[i] == iv
```

As we can see, quantified formulae in JML have three parts, separated by "`;`". The first declares the type of the quantified variable, the second is intended to further restrict the range of the variable, while the third states the "main" property, intuitively speaking. Logically, however, the second and the third part of a JML "`\exists`"-formula are connected via "and" ($\wedge$).

The above formula is a *postcondition*, as it constrains the admissible states after execution of `insert()`. A sensible *precondition* would be that the interval to be inserted is defined: `iv != null`. Because assumptions like this are so common, however, they are implicitly assumed in the latest versions of the JML standard [2, Sect. 2.8] and do not have to be added by hand. This behaviour is called "non-null by default." Vice versa, a reference for which `null` is a legal value has to be declared as `nullable`:

—— Java + JML ———

```
static boolean remove(IntervalTreeNode /*@ nullable @*/ node,
                      Interval iv) {
  if ( node == null ) return false;
  ...
```

——— Java + JML ——

In general, JML specifications are written into Java source code files, in form of Java comments starting with the symbol "`@`". The location of the comment depends on the kind of the specification: while tags like `nullable` are attached to variable declarations, JML method specifications (including pre/post-conditions) precede the method they specify. In our example, `IntervalSeq.java` would contain the following lines:

—— Java + JML (1.2) ——

```
/*@ public normal_behavior
  @  ensures (\exists int i; 0 <= i && i < size;
  @                 contents[i] == iv);
  @*/
public void insert(Interval iv) {
    ...
```

—— Java + JML ——

This is an example for a *method contract* in JML. For the purpose of our example, this contract is, however, still very weak. It does, for instance, not specify how the values of `size` before and after execution of `insert()` relate to each other. For such a purpose, JML offers the "`\old`" construct, which is used in a postcondition to refer back to the pre-state. With that, we can state `size == \old(size) + 1`. Further, the contract does not yet tell whether all (or, in fact, any) of the intervals previously contained in `contents` remain therein, not to speak of the indices under which they appear. What we need to say is (a) that, up to the index `i` where `iv` is inserted, the elements of `contents` are left untouched, and (b) that all other elements are shifted by one index. Both can be expressed using the universal quantifier in JML, "`\forall`", which is quite analogous to the "`\exists`" operator. Using that, (b) would translate to:

```
\forall int k; i < k && k < size;
               contents[k] == \old(contents[k-1])
```

Note that, in case of "`\forall`", the second "`;`" logically is an implication, not a conjunction as was the case for "`\exists`". In the above formula, `i` refers to the index of insertion, which we have existentially quantified over earlier, meaning we get a nested quantification here.

Together with an appropriate `assignable` clause to be explained below, we now arrive at the following JML specification of `insert()`:

—— Java + JML (1.3) ——

```
/*@ public normal_behavior
  @  ensures size == \old(size) + 1;
  @  ensures (\exists int i; 0 <= i && i < size;
  @                contents[i] == iv
  @                && (\forall int j; 0 <= j && j < i;
  @                       contents[j] == \old(contents[j]))
  @               && (\forall int k; i < k && k < size;
  @                      contents[k] == \old(contents[k-1])));
  @  assignable contents, contents[*], size;
  @*/
public void insert(Interval iv) {
    ...
```
——— Java + JML ———

The `assignable` clause, in this example, says that the `insert()` is allowed to change the value of `contents`, the value of the element locations of `contents`, and of `size`, *but nothing else.* The purpose of the `assignable` clauses is not so much the verification of the method `insert` (in this case), but rather to keep feasible the verification of other methods calling `insert()`.

Within the Abstract Class `SortedIntervalSeq`. This class `extends` the class `IntervalSeq`, augmenting it with the notion of *sortedness.* In particular, this class' implementation of `insert()` must respect the sorting. To specify this requirement in JML, one could be tempted to add sortedness to both, the pre- and the postcondition of `insert()`. However, such invariant properties should rather be placed in JML *class invariants*, which like method contracts are added as comments to the source code.

The following lines are put *anywhere* within the class `SortedIntervalSeq`:

——— JML (1.4) ———
```
/*@ public invariant
  @   (\forall int i; 0 <= i && i < size - 1;
  @       getBoundary(contents[i]) <= getBoundary(contents[i+1]));
  @*/
```
——— JML ———

The actual sorting criterion, `getBoundary()`, is left to subclasses of this class, by making it an `abstract` method.

——— Java + JML (1.5) ———
```
protected /*@ pure @*/ abstract int getBoundary(Interval iv);
```
——— Java + JML ———

The phrase "`/*@ pure @*/`" is another piece of JML specification, stating that all implementations of this method terminate (on all inputs), and are free of side effects. Without that, we would not be allowed to use `getBoundary()` in the invariant above, nor in any other JML formula.

Finally, we give the `SortedIntervalSeq` implementation of `insert()` (overriding some non-sorted implementation from `IntervalSeq`):

——— Java (1.6) ———
```
public void insert(Interval iv) {
    int i = size;
    incSize ();
    final int ivBoundary = getBoundary( iv );
    while ( i > 0 && ivBoundary < getBoundary( contents[i-1] ) ) {
        contents[i] = contents[i - 1];
        --i;
    }
    contents[i] = iv;
}
```
——— Java ——

Within the `SortedByStart...` and `SortedByEnd...` Classes. These two classes extend `SortedIntervalSeq` by defining the sorting criteria to be the “start” resp. “end” of the interval. Within `SortedByStartIntervalSeq`, we have:

—— Java + JML (1.7) ———

```
protected /*@ pure @*/ int getBoundary(Interval iv) {
    return iv.getStart ();
}
```
——— Java + JML ——

and within `SortedByEndIntervalSeq`, we have

—— Java + JML (1.8) ———

```
protected /*@ pure @*/ int getBoundary(Interval iv) {
    return iv.getEnd ();
}
```
——— Java + JML ——

4.2 Dynamic Logic and Proof Obligations

After having completed the specification as described in the previous section we start `bin/runProver` (in your KeY installation directory) as a first step towards verification. The graphical user interface of the KeY prover will pop up. To load files with Java source code and JML specifications, we select File → Load ... (or 🟩 in the tool bar). For the purposes of this introduction we navigate to where the `calender-sources`[7] are stored locally, select that very directory (not any of the sub-directories), and push the open button. After an instant the proof obligation browser will appear on the screen. In the left of the two window panes, the Classes and Operations pane, we expand the folder corresponding to the package `intervals`, then the folder for the class `IntervalSeq`, and finally we select method `insert(Interval iv)`. Now also the Proof Obligations pane shows some entries, of which we choose EnsuresPost. Clicking on Start Proof takes

[7] The sources can be downloaded from `www.key-project.org/fmco06`.

KeY

```
  inReachableState
& \forall intervals.IntervalSeq i_0; (i_0.<created>=TRUE & !i_0=null
                                -> !i_0.contents = null)
& \forall intervals.IntervalSeq i_0; (i_0.<created>=TRUE & !i_0=null
                                -> i_0.contents.length >= i_0.size)
& \forall intervals.IntervalSeq i_0; (i_0.<created>=TRUE & !i_0=null
                                -> i_0.contents.length >= (jint)(1))
...
& (self.<created> = TRUE & !self = null)
& (iv.<created> = TRUE | iv = null)
& !iv = null
->
 {_iv:=iv ||
  \for intervals.IntervalSeq x; contentsAtPre_0(x):=x.contents ||
  \for (int x1; intervals.Interval[] x0) getAtPre_0(x0,x1):=x0[x1] ||
  \for intervals.IntervalSeq x; sizeAtPre_0(x):=x.size}
  \<{
      exc=null;try {
        self.insert(_iv)@intervals.IntervalSeq;
      }  catch (java.lang.Throwable e) {
        exc=e;
      }
    }\> (  self.size = (jint)(javaAddInt(sizeAtPre_0(self),(jint)(1)))
         & \exists jint i; ...
         & exc = null)
```

KeY

Fig. 3. Proof obligation for the `insert` method in class `IntervalSeq`

us to a second dialogue in which the contract to be verified can be chosen (only one is available for the method `insert`), along with the object invariants that are assumed by the method implementation. For the time being, the pre-selected contract and invariants are just fine and we proceed by clicking on Ok, which brings us back to the KeY prover interface.

Now, the Tasks pane records the tasks we have loaded (currently one) and the main window Current Goal shows the proof obligation. It looks quite daunting and we use the rest of this section to explain what you see there. The construction of the actual proof is covered in the next section. Ignoring the leading `==>`, the proof obligation is of the form shown in Fig. 3.

Java Card DL. Fig. 3 shows a formula of Dynamic Logic (DL), more precisely Java Card DL, see Section 2. The reader might recognise typical features of first-order logic: the propositional connectives (e.g., `->` and `&`), predicates (e.g., `inReachableState`, a predicate of arity 0), equality, constant symbols (e.g., `self`), unary function symbols (e.g., `size`), and quantifiers (e.g.,

`\exists jint i;`). The function symbol `size` is the logical counterpart of the attribute of the same name. Note also that Java Card DL uses dot-notation for function application, for example, `self.size` instead of `size(self)` on line 23. What makes Java Card DL a proper extension of first-order logic are modal operators. In the above example the diamond operator

```
\<{... self.insert(_iv)@intervals.IntervalSeq; ...}\>
```

occurs on line 17 (note that in KeY the modal operators `<>` and `[]` are written with leading backslashes). In general, if *prog* is any sequence of legal Java Card statements and F is a Java Card DL formula, then `\<`*prog*`\>`F is a Java Card DL formula too. As already explained in Section 2, the formula `\<`*prog*`\>`F is true in a state s_1 if there is a state s_2 such that `prog` terminates in s_2 when started in s_1 and `F` is true in s_2. The box operator `\[...\]` has the same semantics except that it does not require termination.

In theoretical treatments of Dynamic Logic there is only one kind of variable. In Java Card DL we find it more convenient to separate logical variables (e.g., `i` in the above example), from program variables (e.g. `self`). Program variables are considered as (non-rigid) constant symbols in Java Card DL and may thus not be quantified over. Logical variables on the other hand are not allowed to occur within modal operators, because they cannot occur as part of Java programs.

Exceptions. The abrupt termination of a Java statement due to the occurrence of an exception (e.g., because a reference with value `null` was dereferenced) is in Java Card DL considered as non-termination. This implies that `\<`*prog*`\>`F is false if *prog* raises an exception, while the corresponding box-formula is true, which is often the intended meaning when writing DL formulae. If a more fine-grained specification of the termination behaviour of a program *prog* is required (e.g., to allow only certain kinds of exceptions), *prog* can be enclosed in a **`try ... catch`** statement, as it is the case in Fig. 3. The resulting program as a whole will never raise an exception, but by asserting properties involving the variable `exc` in the post-condition F it is possible to observe how the program terminated.

State Updates. We are certainly not able to touch on all central points of Java Card DL in this quick introduction, but there is one item we cannot drop, namely *updates.* The expression in line 13–16 in Fig. 3 is an example of an update, which consists of a sequence of assignments like `_iv:=iv`. The left-hand side of such an assignment (a *function update*) has to be a non-rigid term like a program variable, as `_iv` in this example, or an array or field access. The right-hand side can be an arbitrary Java Card DL term, which of course must be compatible with the type of the left-hand expression. Constructs like `i:=j++` where the right-hand side would have side-effects are *not* allowed in updates. If *lhs*`:=`*rhs* is a function update and F is a formula, then `{`*lhs*`:=`*rhs*`}`F is a Java Card DL formula. The formula `{`*lhs*`:=`*rhs*`}`F is true in state s_1 if F is true in state s_2 where s_2 is obtained from s_1 by *performing* the update. For example, the state s_2 obtained from s_1 by performing the update `_iv:=iv` (only) differs in the value of `_iv`, which is in s_2 the value that `iv` has in s_1.

The assignments in line 13–16 are combined using the *parallel composition* operator "`||`" that carries out the individual assignments simultaneously: none of the assignments can observe the effects of the other assignments. Swapping two variables, for instance, can be performed using the update `x:=y || y:=x`. A second update connective that occurs in line 14–16 is the *quantification* operator `\for`, which carries out an assignment simultaneously for all values of one or multiple bound variables. Quantification is in Fig. 3 used to store the pre-state values of the fields `contents` and `size` as well as the contents of `Interval`-arrays, which can then be referred to in the post-state (as in line 23). The introduction of these updates is triggered by the usage of the `\old` construct in a specification, like in the following JML `ensures` clause:

```
—— JML ——————————————————————————————————————
  @  ensures size == \old(size) + 1;
—————————————————————————————————————— JML ——
```

One difference between updates and Java assignment statements is that logical variables such as `i` may occur on the right-hand side of updates. In Java Card DL it is not possible to quantify over program variables. This is made up for by the possibility of quantifying over logical variables, whose values can then be assigned to program variables by an update. Finally, the most important role of updates is that of *delayed substitutions*. During symbolic execution (performed by the prover using the Java Card DL calculus) the effects of a program are removed from the modality `\<...\>` and turned into updates, where they are *simplified* and *parallelised*. Only when the modality has been eliminated, updates are substituted into the post-state. For a more thorough discussion, we refer to the KeY book [1, Chapt. 3] and to [17] (page 115).

Kripke Semantics. A state $s \in S$ contains all information necessary to describe the complete snapshot of a computation: the existing instances of all types, the values of instance fields and local program variables etc. Modal logic expressions are not evaluated relative to one state model but relative to a collection of those, called a *Kripke Structure*. There are *rigid* symbols that evaluate to the same meaning in all states of a *Kripke Structure*. The type `int` (see e.g., line 15 in Fig. 3) in all states evaluates to the (infinite) set of integers, also addition `+` on `int` are always evaluated as the usual mathematical addition. Logical variables also count among the rigid symbols, no program may change their value. On the other hand there are *non-rigid* symbols like `self`, `_iv`, `contents`, or `at(i)`.

Proof Obligations. We have to add more details on Java Card DL as we go along but we are now well prepared to talk about proof obligations. We are still looking at Fig. 3 containing the proof obligation in the Current Goal pane that was generated by selecting the `normal_behavior` specification case. Line 11 contains the (implicit) pre-condition that `iv` is a valid reference, while the conjunction of all JML invariants for class `IntervalSeq` appears in line 2–8. Since in the JML semantics `normal_behavior` includes the termination requirement, the diamond modality is used. Starting with line 23, the first line within the scope of the modal operator, follows the conjunction of the `ensures` clauses in the JML method specification.

Looking again at Fig. 3, we notice in line 10 additional restrictions on the implicitly universally quantified parameter `iv`. To understand what we see here, it is necessary to explain how Java Card DL handles object creation. Java Card DL adopts what is called the *constant domain assumption* in modal logic theory. According to this assumption all states share the same objects for all occurring types. In addition there is an implicit field `<created>` that is defined in the class `java.lang.Object` (to emphasise that this is not a normal field, it is set within angled brackets). Initially we have `o.<created> = FALSE` for all objects `o`. If a `new` operation is performed we look for the next object `o` to be created and change the value of `o.<created>` from `FALSE` to `TRUE`, which now is nothing more than any other function update.

A symbol that remained unexplained so far is the function `javaAddInt` in line 23, which is puzzling considering that also an ordinary `+` is available in Java Card DL. The reason for not using `+` directly is that KeY allows different integer semantics both for Java and for JML, and depending on the semantics a JML-`+` can be interpreted as the mathematical `+` or as the addition in modular 32-bit arithmetic [1, Chapt. 12]. The symbol `javaAddInt` always refers to the addition of the active integer semantics; similar function symbols exist for the other arithmetic operations. The mode to be used can be chosen under Options → Default Taclet Options ....

Proper Java States. It still remains to comment on the precondition that we skipped on first reading, `inReachableState`. In KeY a method contract is proved by showing that the method terminates in a state satisfying the postcondition when started in any state s_1 satisfying the preconditions and the invariants. This may also include states s_1 that cannot be reached from the `main` method. But, usually the preconditions and invariants narrow down this possibility and in the end it does not hurt much to prove a bit more than is needed. But, there is another problem here: the implicit fields. A state with object `o` and field `a` such that `o.<created> = TRUE`, `o.a != null`, and `o.a.<created> = FALSE` is not possible in Java, but could be produced via updates. It is the precondition `inReachableState` that excludes this kind of anomalies.

Capturing JML Specifications in Java Card DL. Let us go back to the proof obligation browser and select RespectsModifies for the `insert` method. When proving, e.g., the `normal_behavior` clause of a method contract, we also take advantage of the JML `assignable` clause. The current proof obligation now checks if the `assignable` clauses are indeed correct: a call to the `insert` method only assigns to those fields of the called object that are mentioned in the `assignable` clause, all other fields remain unchanged.

Now, let us select the last proof obligation in the proof obligation browser, which is named PreservesInv. Its purpose is to make sure that, for any state s_1 that satisfies all invariants of the `IntervalSeq` class and the preconditions of `insert(iv)`, the invariants are again true in the end state s_2 of this method. Note that here the modal box-operator is used. Termination of the method was already part of its method contract, so we need not prove it again here. The proof obligation requires the invariants to also hold when the methods terminates via an exception. This is the reason why `insert(iv)` is enclosed in a `try-catch` block.

4.3 Verification

In this section, we demonstrate how the KeY prover is used to verify a proof obligation resulting from our example. It is important to note, however, that a systematic introduction into the usage of the prover is beyond the scope of this paper. Such an introduction can be found in Chapter 10 of the KeY book [1]. On the other hand, the examples in that chapter are of toy size as compared to the more realistic proof obligations we consider in this paper.

This section is meant to be read with the KeY prover up and running, to perform the described steps with the system right away. The exposition aims at giving an *impression* only, on how verification of more realistic examples is performed, while we cannot explain in detail *why* we are doing what we are doing. Again, please refer to [1, Chapt. 10] instead.

We will now verify that the implementation of the method `insert()` in class `SortedIntervalSeq` (not in `IntervalSeq`) respects the contract that it inherits from `IntervalSeq`. Before starting the proof, we remind ourselves of the code we are going to verify: the implementation of `insert()` was given in listing (1.6) in Sect. 4.1, and it calls the inherited method `incSize()`, see listing (1.1). Both these methods contain one `while` loop, which we advise the reader to look at, as we have to recognise them at some point during the verification.

We first let KeY generate the corresponding proof obligation, by following the same steps as described at the beginning of Sect. 4.2 (from File → Load ... onwards), but with the difference that we select `SortedIntervalSeq` instead of `IntervalSeq` the Classes and Operations pane. We choose again the method `insert(Interval iv)` and the specification case EnsuresPost.

In the next dialogue, the Contract Configurator, we need to add further invariants to be assumed for the verification: by default, only the invariants of the class `SortedIntervalSeq` are included, which are not sufficient as important properties are also asserted in the superclass `IntervalSeq`. To add these invariants, change to the Assumed Invariants tab, click on the class `IntervalSeq` in the Classes pane, and then select all of the offered invariants in the Invariants pane (to select multiple invariants, use the left mouse key together with the Control or the Shift key).

Afterwards, the Current Goal pane contains a proof obligation that is very similar to the one discussed in Sect. 4.2, just that now the (translated) class invariant of `SortedIntervalSeq`, see listing (1.4), serves as an additional assumption.

This now is a good time to comment on the the leading "`==>`" symbol in the Current Goal pane. As described in Sect. 2, the KeY prover builds proofs based on a *sequent calculus.* Sequents are of the form $\phi_1, \ldots, \phi_n \Longrightarrow \phi'_1, \ldots, \phi'_m$, where $\phi_1, \ldots, \phi_n$ and $\phi'_1, \ldots, \phi'_m$ are two (possibly empty) comma-separated lists of formulae, separated by the sequent arrow $\Longrightarrow$ (that is written as "`==>`" in the KeY system). The intuitive meaning of a sequent is: if we assume all formulae $\phi_1, \ldots, \phi_n$ to hold, then *at least one* of the formulae $\phi'_1, \ldots, \phi'_m$ holds. We refer to "$\phi_1, \ldots, \phi_n$" and "$\phi'_1, \ldots, \phi'_m$" as the "left-hand side" (or "antecedent") and "right-hand side" (or "succedent") of the sequent, respectively.

The particular sequent we see now in the Current Goal pane has only one formula on the right-hand side, and no formulae on the left-hand side, which is the typical shape for generated proof obligations, prior to application of any calculus rule. It is the purpose of the sequent calculus to, step by step, take such formulae apart, while collecting assumptions on the left-hand side, and alternatives on the right-hand side, until the sheer shape of a sequent makes it trivially true. Meanwhile, certain rules make the proof branch.

We prove this goal with the highest possible degree of automation. However, we first apply one rule interactively, just to show how that is done. In general, interactive rule application is supported by the system offering only those rules which are applicable to the highlighted formula, resp. term (or, more precisely, to its top-level operator). If we now click on the leading "`->`" of the right-hand side formula, a context menu for rule selection appears. It offers several rules applicable to "`->`", among them impRight, which in textbook notation looks like this:

$$\mathsf{impRight}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \mathrel{-\!\!>} \psi, \Delta}$$

A tool-tip shows the corresponding taclet. Clicking on impRight will apply the rule in our proof, and the Current Goal pane displays the new goal. Moreover, the Proof tab in the lower left corner displays the structure of the (unfinished) proof. The nodes are labelled either by the name of the rule which was applied to that node, or by "OPEN GOAL" in case of a goal. (In case of several goals, the one currently in focus is highlighted in blue.) We can see that impRight has been applied *interactively* (indicated by a hand symbol).

The proof we are constructing will be several thousand steps big, so we better switch to automated proof construction now. For that, we select the Proof Search Strategy tab in the lower left corner, and configure the proof strategy as follows:

– Max. rule applications: 5000 (or just any big number)
– Java DL (the strategy for proving in Java Card DL)
– Logical splitting: Normal (do not delay proof splitting)
– Loop treatment: None (we want symbolic execution to stop in front of loops)
– Method treatment: Expand (methods are inlined during symbolic execution)
– Query treatment: Expand (queries in specifications are inlined)

– Arithmetic treatment: Basic (simple automatic handling of linear arithmetic)
– Quantifier treatment: No Splits
  (use heuristics for automatic quantifier handling, but do not perform instantiations that might cause proof splitting)
– User-specific taclets: all off
  (we do not make use of user-defined proof rules in this paper)

We run the strategy by clicking the ▶ button (either in the Proof Search Strategy tab or in the tool bar). The strategy will stop after about 1000 rule applications once the symbolic execution arrives at loops in the program (due to Loop treatment: None). We open the Goals tab, where we can see that there are currently five goals left to be proven.

We can view each of these goals in the Current Goal pane, by selecting one after the other in the Goals tab. In four of the five goals, the modality (preceded by a parallel update) starts with:

—— KeY (1.9) ——
```
\<{method-frame(...): {
      while ( i>0 && ivBoundary<getBoundary(contents[i-1]) ) {
        ...
```
—— KeY ——

In those four proof branches, symbolic execution is just about to "enter" the while loop in the method `insert()`. In the remaining fifth branch, the same holds for the while loop in the method `incSize()`. For the loop in `insert()`, we get four cases due to the two existing implementations of the interface `Interval` and the two concrete subclasses of the abstract class `SortedIntervalSeq`. All four cases can be handled in the same way and by performing the same interactions, to be described in the following.

In each case, we first have to process the while loop at the beginning of the modality. It is well known that loops cannot be handled in a similarly automated fashion as most other constructs.

Loop Invariants. Generally, for programs containing loops we have to choose a suitable *loop invariant*[8] (a formula) in order to prove that the loop has the desired effect and a *loop variant* (an integer term) for proving that the loop terminates. We also have to specify the *assignable memory locations* that can be altered during execution of the loop. All this information can be entered as part of an interactive proof step in KeY. However, the prover also supports the JML feature of annotating loops with invariants, variants, and assignable locations.

Invariants typically express that the loop counter is in a valid range, and give a closed description of the effect of the first n iterations. For the loop in the method `insert()`, it is necessary to state in the invariant that:

– the loop counter `i` never leaves the interval $[0, \texttt{size})$,

[8] As an alternative to using invariants, KeY offers induction, see [1, Chapt. 11].

– the interval is not inserted too far left in the array, and
– the original contents of the sequence are properly shifted to the right.

The last component of the invariant is very similar to the post-condition of the whole `insert()` method (see listing (1.3)): it has to be stated that all array components that have already been visited by the loop are shifted to the right, whereas a prefix of the array remains unchanged. This can be achieved using the JML `\old` operator, which in a loop invariant (like in a post-condition) refers to the pre-state of the enclosing method.

Stating the termination of the loop is simple, because the variable `i` is always non-negative and decreased in each iteration. Further, we specify that the only modifiable memory locations are the loop counter and the elements of the array `contents`. Altogether, this yields the following specification:

—— Java + JML ——
```
/*@ loop_invariant 0 <= i && i < size &&
  @    (i+1 < size ==>
  @            ivBoundary < getBoundary( contents[i+1] )) &&
  @    (\forall int k; 0 <= k && k < i;
  @            contents[k] == \old(contents[k])) &&
  @    (\forall int k; i < k && k < size;
  @            contents[k] == \old(contents[k-1]));
  @ decreases i;
  @ assignable contents[*], i;
  @*/
while ( i > 0 && ivBoundary < getBoundary( contents[i-1] ) ) {
     contents[i] = contents[i - 1];
     --i;
}
```
—— Java + JML ——

Verification Using Invariants. We continue our proof on one of the four similar goals, all of which containing the modality of the form (1.9), such that processing the loop in `insert()` is the next step. Because all these four goals can be handled in the same way, we can pick an arbitrary one of them, by selecting it in the Goals tab. Before proceeding, we switch to the Proof tab, to better see the effect of the upcoming proof step.

Before we apply the actual invariant rule, we perform one further interactive proof step that will simplify the rest of the proof. The sequent contains a quantified formula stating that the elements of the `contents` array are not `null`:

—— KeY ——
```
\forall intervals.IntervalSeq i_0; \forall jint i;
  (i <= -1 | i_0 = null | !i_0.<created> = TRUE | i_0.size <= i
                                         | !i_0.contents[i] = null)
```
—— KeY ——

We will frequently need instances of this invariant, but in some cases the heuristics built into KeY are not able to derive these instances automatically. KeY can be helped in such case by manually instantiating the formula with the required terms, which at this point is the variable `self` denoting the object of the class `SortedIntervalSeq` at hand. To perform this instantiation, use the mouse to drag any occurrence of `self` in the sequent to the quantifier `\forall intervals.IntervalSeq i` and choose the rule allLeft in the appearing menu.[9]

We then apply an invariant rule which automatically extracts the JML annotation of our loop from the source code. For that, we click on any of the "`:=`" symbols in the parallel update preceding the modality `\<...\>`, and select loopInvariant (with variant) from the rules offered. Depending on the settings, a Choose Taclet Instantiation window can pop up, where we just press Apply.

Afterwards, the Proof tab tells us (possibly after scrolling down a bit) that the application of this invariant rule has resulted in four proof branches:

– Invariant Initially Valid: It has to be shown that the chosen invariant holds when entering the loop.
– Body Preserves Invariant: Under the assumption that the invariant and the loop condition hold, after one loop iteration the invariant still has to be true.
– Termination: Under the assumption that the invariant and the loop condition hold, the chosen variant has to be decreased by the loop body, but has to stay non-negative.
– Use Case: The remaining program has to be verified now using the fact that after the loop terminates, the invariant is true and the loop condition is false.

The four cases can be proven as follows. Generally, for a complex proof like this, it is best to handle the proof goals one by one and to start the automatic application of rules only locally for a particular branch. This is done by clicking on a sequent arrow `==>` and choosing Apply rules automatically here, or by shift-clicking on a sequent arrow, or by right-clicking on a node in the proof tree display and selecting Apply Strategy from the context menu. (Clicking on ▶, in contrast, will apply rules to *all* remaining proof goals, which is too coarse-grained if different search strategy settings have to be used for different parts of the proof.)

Also, please note that a proof branch beginning with a green folder symbol is closed. Therefore, this symbol is a success criteria in each of the following four cases. Moreover, branches in the Proof tab can be expanded/collapsed by clicking on ⊞/⊟. To keep a better overview, we advise the reader to collapse the branches of the following four cases once they are closed.

Invariant Initially Valid. The proof obligation can easily be handled automatically by KeY and requires about 100 rule applications.

[9] In case the option Options → DnD Direction Sensitive is enabled, KeY will perform the instantiation without showing a menu.

Body Preserves Invariant. This is the goal that requires the biggest (sub-)proof with about 11000 rule applications. In the Proof Search Strategy pane, choose a maximum number of rule applications of 20000 and run the prover in auto-mode on the goal as described above. This will, after a while, close the "Body Preserves Invariant" branch.

Termination. Proceed as for the case Body Preserves Invariant (possibly after expanding the branch and selecting its OPEN GOAL). This case will be closed after about 6000 steps.

Use Case. Proceed similarly as for the case Body Preserves Invariant. At first, calling the automated strategy will perform about 5000 rule applications. In contrast to the other three cases, for this last branch it is also necessary to manually provide witnesses for certain existentially quantified formulae (in the succedent) that can neither be found by KeY, nor by the external prover Simplify [12], automatically. These formulae correspond to the post-condition of the method `insert()`, where a point has to be "guessed" at which a further element has been added to the sequence. The form of the formulae is:

```
\exists jint i; \forall jint j;  F
```

Fortunately, for this problem, it is easy to read off the witness `i` that allows to prove the formulae: the body F always contains equations of the form `i = `t, where t is the desired witness. To perform the instantiation of the formula, drag the term t to the quantifier `\exists jint i` and choose the rule exRightHide in the appearing menu. After this instantiation step, locally call the automated strategy. In this way, handle all branches with formulae of the above form, until the "Use Case" has a green folder at its beginning, meaning this case is closed.

5 Second Walk-through: Specifying and Verifying Timeframe Displays

In this section, we practise specification and verification a second time, now with higher speed, coarser granularity, and with more focus on the direct usage of dynamic logic (without JML). The example is the method `add()` of the class `TimeFrameDisplay`.

5.1 Formal Specification and Implementation

Within the Class `TimeFrameDisplay`. The class `TimeFrameDisplay` is a concrete application of the calendar view `TimeFrameCalendarView`, and could be (the skeleton of) a dialogue displaying a certain time period in a calendar. On the following pages, we demonstrate how we can give a more behavioural specification for some aspects of such a dialogue. The investigated method is `TimeFrameDisplay::add`, which simply delegates the addition of a new entry to the underlying `Calendar` object:

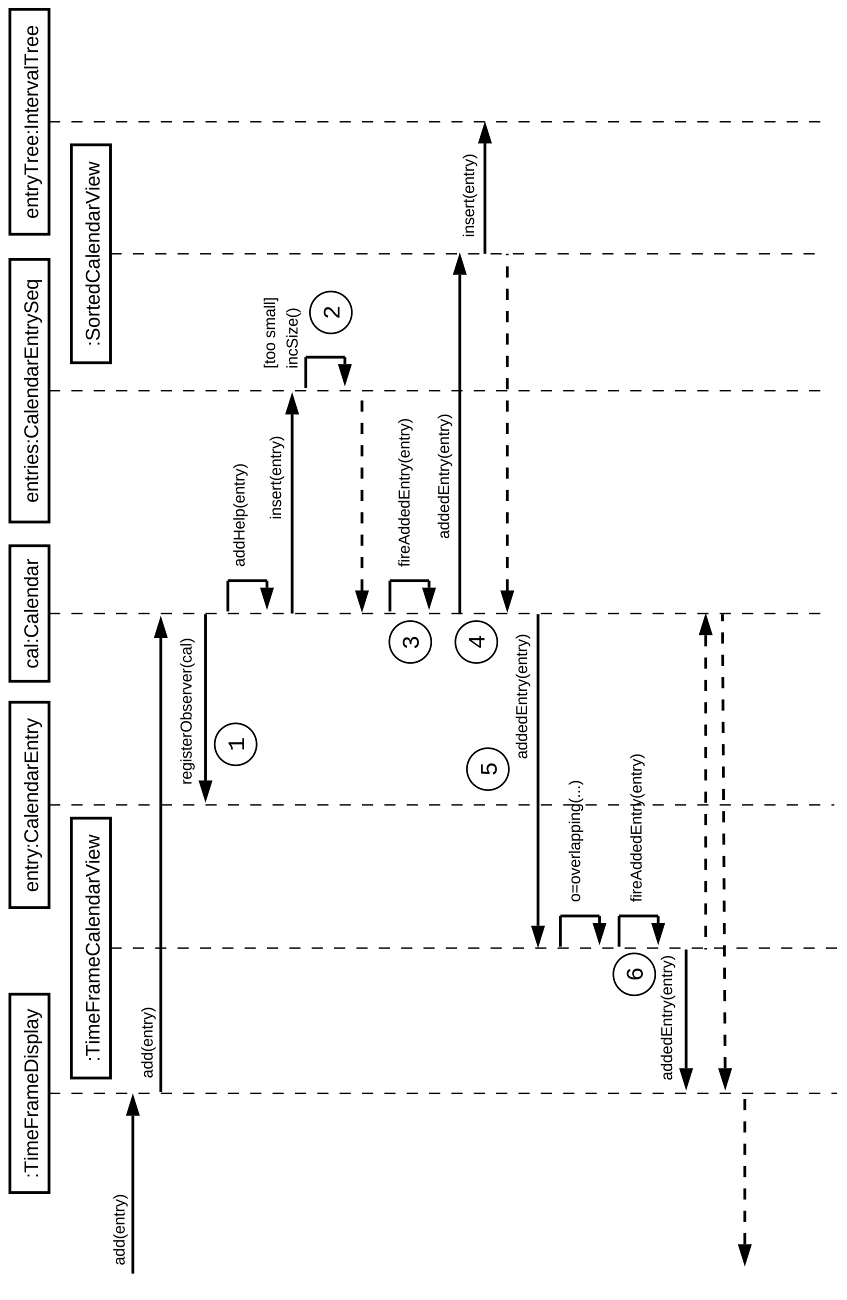


Fig. 4. UML sequence diagram showing the effect of calling `TimeFrameDisplay::add`

Java + JML (1.10)

```
public class TimeFrameDisplay implements CalendarListener {
  ...
  /*@ public normal_behavior
    @  requires entry != null;
    @  requires overlapping ( timeFrame, entry );
    @  ensures lastEntryAdded == entry;
    @*/
  public void add(CalendarEntry entry) {
    cal.add ( entry );
  }
  ...
  private CalendarEntry lastEntryAdded = null;
  public void addedEntry(CalendarEntry e) {
    lastEntryAdded = e;
  }
```

Java + JML

In this context, we would like to specify that calling `add` actually results in a new calendar entry being displayed on the screen. In order to simulate this effect, we introduce an attribute `lastEntryAdded` that is assigned in the method `addedEntry`. The post-condition of method `add`, `lastEntryAdded == entry`, consequently states that calling `add` eventually raises the signal `addedEntry` with the right argument (see Fig. 4 for an illustration).

5.2 Proof Obligations and Verification

This time, we demonstrate the use of a hand-written Java Card DL proof obligation instead of importing a JML specification into KeY. Formulating a problem directly in DL is more flexible and gives us full control over which assumptions we want to make, but it is also more low-level, more intricate, and requires more knowledge about the logic and the prover (for a larger case study of specification in Java Card DL see [18]). Figure 5 shows the main parts of the file `timeFrameDisplayAdd.key` containing the proof obligation. A full account on the syntax used in KeY input files is given in [1, Appendix B]. As before, we can load `timeFrameDisplayAdd.key` by selecting File → Load ... (or 🟩 in the tool bar) and choosing the file in the appearing dialogue.

The KeY input file in Fig. 5 starts with the path to the Java sources under investigation, and with a part that declares a number of program variables (lines 2–4) used in the specification. The main part of the file describes one particular scenario that we want to simulate:

– In lines 7–8, we assume that `self` and `entry` refer to proper objects of classes `TimeFrameDisplay` resp. `CalendarEntry`. The calendar entry is also supposed to overlap with the attribute `self.timeFrame` (line 9), which is the pre-condition of the method `TimeFrameDisplay::add`.

KeY
```
\javaSource "calendar-sources/";
\programVariables {
  calendar.CalendarEntry entry; calendar.CalendarEntry old_entry;
  TimeFrameDisplay self;                                             }
\problem {
    inReachableState
  & self != null & self.<created> = TRUE
  & entry != null & entry.<created> = TRUE
  & TimeFrameDisplay.overlapping(self.timeFrame, entry) = TRUE

  & self.cal!=null & self.timeFrame!=null & self.timeFrameView!=null
  & self.timeFrameView.listenersNum = 1 & self.cal.listenersNum = 2
  & self.timeFrameView.listeners[0] = self
  & self.cal.listeners[0] = self.cal.completeView
  & self.cal.listeners[1] = self.timeFrameView
  & self.timeFrameView.timeFrame = self.timeFrame

  & \forall calendar.CalendarModificationServer serv;
      (   serv != null & serv.<created> = TRUE
       ->    serv.listeners != null
           & 0 <= serv.listenersNum & 1 <= serv.listeners.length
           & serv.listenersNum <= serv.listeners.length)
  & \forall observerPattern.Subject subj;
      (   subj != null & subj.<created> = TRUE
       ->   subj.observers != null
          & 0 <= subj.observersNum & 1 <= subj.observers.length
          & subj.observersNum <= subj.observers.length)
  & \forall calendar.CalendarEntrySeq entry;
      (   entry != null & entry.<created> = TRUE
       ->   entry.contents != null
          & 0 <= entry.size & 1 <= entry.contents.length
          & entry.size <= entry.contents.length)
  & \forall calendar.Calendar cal;
      (   cal != null & cal.<created> = TRUE
       -> cal.entries != null & cal.completeView != null)
  & \forall calendar.TimeFrameCalendarView view;
      (   view != null & view.<created> = TRUE
       ->   view.completeView != null & view.timeFrame != null
          & view.cal != null)
  & \forall calendar.SortedCalendarView view;
      (view != null & view.<created> = TRUE -> view.entryTree != null)

  -> {old_entry := entry} \<{ self.add(entry)@TimeFrameDisplay; }\>
                                   self.lastEntryAdded = old_entry   }
```
KeY

Fig. 5. The hand-written proof obligation for Sect. 5

– The `TimeFrameDisplay` object `self` has been properly set up and connected to a `Calendar` and to a `TimeFrameCalendarView` (lines 11, 16). The freshly created `TimeFrameCalendarView` has exactly one listener attached, namely the object `self` (lines 12, 13). Likewise, the calendar `self.cal` does not have any listeners registered apart from its `SortedCalendarView` and the `TimeFrameCalendarView` (lines 12, 14, 15).
– In order to perform the verification, we need to assume a number of invariants. Lines 18–32 contain three very similar class invariants for the classes `CalendarModificationServer`, `Subject`, and `CalendarEntrySeq`, mostly expressing that the arrays for storing listeners and calendar entries are sufficiently large. In lines 33–41, we state somewhat simpler invariants for `Calendar`, `TimeFrameCalendarView`, and `SortedCalendarView` that ensure that attributes are non-null.

In this setting, we want to show that an invocation of the method `self.add` with parameter `entry` has the effect of raising a signal `addedEntry`. This property is stated in lines 43–44 using a diamond modal operator.

Loop Handling. Apart from sequential code that can simply be executed symbolically, there are three loops in the system that require our attention in this setting. The loops in the methods `Subject::registerObserver` (① in Fig. 4) and `CalendarEntrySeq::incSize` (② in Fig. 4) are similar in shape and are necessary for handling the dynamically growing arrays of entries and listeners:

—— Java + JML ——
```
public void registerObserver(Observer obs) {
  ++observersNum;
  if ( observersNum > observers.length ) {
    final Observer[] oldAr = observers;
    observers = new Observer[observers.length * 2];
    int i = 0;
    while ( i < oldAr.length ) {observers[i] = oldAr[i]; ++i;}
  }
  observers[observersNum - 1] = obs;
}
...
protected void incSize() {
  ++size;
  if ( size > contents.length ) {
    final CalendarEntry[] oldAr = contents;
    contents = new CalendarEntry[contents.length * 2];
    int i = 0;
    while ( i < oldAr.length ) {contents[i] = oldAr[i]; ++i;}
  }
}
```
—— Java + JML ——

We can handle both loops in the same way (and with the same or similar invariants) as in Sect. 4.3. As before, it is enough to annotate the loops with JML invariants and variants, which can be read and extracted by KeY during the verification.

The third occurrence of a loop is in the class `CalendarModificationServer` in package `calendar` (③ and ⑥ in Fig. 4):

—— KeY ——
```
protected void fireAddedEntry(CalendarEntry entry) {
  int i = 0;
  while ( i != listenersNum ) {
    listeners[i].addedEntry ( entry ); ++i;}
}
```
—— KeY ——

This loop is executed after adding a new entry to the calendar and is responsible for informing all attached listeners about the new entry. In our particular scenario, there are exactly two listeners (the objects `self.cal.completeView` and `self.timeFrameView`), and therefore we can handle this loop by unwinding it twice.

The Actual Verification, Step by Step. After loading the problem file shown in Fig. 5, we select proof search options as in Sect. 4.3:

- Logical splitting: Normal
- Loop treatment: None
- Method treatment: Expand
- Query treatment: None
  (we do not inline queries immediately, because we want to keep the expression `TimeFrameDisplay.overlapping(self.timeFrame,entry)` that occurs in Fig. 5 for later)
- Arithmetic treatment: Basic
- Quantifier treatment: No Splits
- User-specific taclets: all off

Running the prover with these options and about 1000 rule applications gets us to the point where we have to handle the loops of the verification problem. There are three goals left, corresponding to the points ①, ② and ③ in Fig. 4, one for each of the loops that are described in the previous paragraph. This is due to the fact that the loops in the methods `incSize` and `registerObserver` are only executed if it is necessary to increase the size of the arrays involved. Consequently, the proof constructed so far contains two case distinctions and three possible cases. As the loops in `incSize` and `registerObserver` can be eliminated using invariants (exactly as in the previous section), we concentrate on the third loop in method `fireAddedEntry` that is met at point ③ in Fig. 4.

In order to unwind the loop of `fireAddedEntry` once, click on the program block containing the method body and choose the rule unwindWhile. This duplicates the loop body and guards it with a conditional statement. That is, the loop "`while(`*b*`){`*prog*`}`" is replaced by "`if(`*b*`){`*prog*`;while(`*b*`){`*prog*`}}`".

After unwinding the loop, we have to deal with the first object listening for changes in the calendar, which is a `SortedCalendarView`. To continue, select Method treatment: None and run the prover in automode. The prover will stop at the invocation `SortedCalendarView::addedEntry` (④ in Fig. 4), which we can unfold using the rule methodBodyExpand. After that, continue in automode.

Method Contracts. The method `SortedCalendarView::addedEntry` inserts the new `CalendarEntry` into an interval tree to enable subsequent efficient lookups. Consequently, the next point where the prover stops is an invocation of the method `IntervalTree::insert`. The exact behaviour of this insertion is not important for the present verification problem, however, so we get rid of it using a *method contract* that only specifies which parts of the program state could possibly be affected by the insertion operation. Such a contract can be written based on Dynamic Logic and is shown in Fig. 6 (it is contained in the file `timeFrameDisplayAdd.key`). We specify that the pre-condition of the method `IntervalTree::insert` is `ivt != null & iv != null`, that arbitrary things can hold after execution of the method (the post-condition is `true`), but that only certain attributes of classes in the `intervals` package can be modified (the attributes listed behind the keyword `\modifies`).

In order to apply the method contract, we click on the program and select the item Use Operation Contract in the context menu. In the appearing dialogue, we have to select the right contract `intervalTreeInsert`. Besides, we deselect all assumed or ensured invariants: we change to the tabs Assumed Invariants and Ensured Invariants where we press Unselect all.

Applying the contract leads to three new proof goals: one in which the pre-condition of the contract has to be proven, one where the post-condition is assumed and the remaining program has to be handled, and one where the possible abrupt termination of the method has to be taken care of. By continuing in automode, the first and the third goal can easily be closed, and in the second goal the prover will again stop at point ③ in front of the loop of method `fireAddedEntry` (the second iteration of the loop).

Coming Back to `TimeFrameDisplay`. The next and last callback that needs to be handled is the invocation of `TimeFrameCalendarView::addedEntry` at point ⑤. This method checks whether the calendar entry at hand overlaps with the time period `TimeFrameCalendarView::timeFrame`, and in this case it will forward the entry to the `TimeFrameDisplay`:

—— Java + JML ——

```
public void addedEntry(CalendarEntry e) {
  if ( overlapping ( timeFrame, e ) ) fireAddedEntry ( e );
}
```

—— Java + JML ——

The property `overlapping(timeFrame, e)` is given as a pre-condition of the method `TimeFrameDisplay::add` and now occurs as an assumption in the antecedent of the goal:

```
—— KeY ——
\contracts {
  intervalTreeInsert {
    \programVariables {
      intervals.IntervalTree ivt; intervals.Interval iv;
    }
    ivt != null & iv != null
    -> \<{ ivt.insert(iv)@intervals.IntervalTree; }\>
    true
    \modifies { ivt.root,
      \for intervals.IntervalTreeNode n; n.cutPoint,
      \for intervals.IntervalTreeNode n; n.left,
      \for intervals.IntervalTreeNode n; n.right,
      \for intervals.IntervalTreeNode n; n.sortedByStart,
      \for intervals.IntervalTreeNode n; n.sortedByEnd,
      \for intervals.IntervalTreeNode n; n.sortedByStart.size,
      \for intervals.IntervalTreeNode n; n.sortedByStart.contents,
      \for (intervals.IntervalTreeNode n; int i)
        n.sortedByStart.contents[i],
      \for intervals.IntervalTreeNode n; n.sortedByEnd.size,
      \for intervals.IntervalTreeNode n; n.sortedByEnd.contents,
      \for (intervals.IntervalTreeNode n; int i)
        n.sortedByEnd.contents[i] }
  };
}
—— KeY ——
```

Fig. 6. Java Card DL contract for the method `CalendarEntry::insert`

```
TimeFrameDisplay.overlapping(self.timeFrame, entry) = TRUE
```

We can simply continue with symbolic execution on the proof branch. Because we want the prover to take all available information into account and not to stop in front of loops and methods anymore, select Loop treatment: Expand, Method treatment: Expand, and Query treatment: Expand. Choose a maximum number of rule applications of about 5000. Then, click on the sequent arrow `==>` and select Apply rules automatically here. This eventually closes the goal.

6 Conclusion

In this paper we walked step-by-step through two main verification tasks of a non-trivial case study using the KeY prover. Many of the problems encountered here—for example, the frame problem, what to include into invariants, how to modularise proofs—are discussed elsewhere in the research literature, however, typical research papers cannot provide the level of detail that would one enable to actually trace the details. We do not claim that all problems encountered are yet optimally solved in the KeY system, after all, several are the target of active research [19]. What we intended to show is that realistic Java programs actually can be specified and verified in a modern verification system and, moreover, all crucial aspects can be explained within the bounds of a paper while the verification process is to a very large degree automatic. After studying this tutorial, the ambitious reader can complete the remaining verification tasks in the case study.

As for "future work," our ambition is to be able to write this tutorial without technical explanations on how the verification is done while covering at least as many verification tasks. We would like to treat modularisation and invariant selection neatly on the level of JML, and the selection of proof obligations in the GUI. It should not be necessary anymore to mention Java Card DL in any detail. From failed proof attempts, counter examples should be generated and animated without the necessity to inspect Java Card DL proof trees. There is still some way to go to mature formal software verification into a technology usable in the mainstream of software development.

Acknowledgements

We would like to thank Richard Bubel for many discussions on various topics of the paper, and for his enormous contribution to the constant improvement of the KeY system.

References

1. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2007)
2. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. (2007)
3. Bubel, R., Roth, A., Rümmer, P.: Ensuring the correctness of lightweight tactics for JavaCard dynamic logic. Electronic Notes in Theoretical Computer Science **199** (2008) 107–128
4. Hunt, J.J., Jenn, E., Leriche, S., Schmitt, P., Tonin, I., Wonnemann, C.: A case study of specification and verification using JML in an avionics application. In Rochard-Foy, M., Wellings, A., eds.: Proceedings, Fourth Workshop on Java Technologies for Real-time and Embedded Systems, ACM Press (2006) 107–116
5. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In Hutter, D., Ullmann, M., eds.: Proceedings, Second International Conference on Security in Pervasive Computing. Volume 3450 of LNCS, Springer (2005) 193–209
6. Beckert, B., Gladisch, C.: White-box testing by combining deduction-based specification extraction and black-box testing. In Gurevich, Y., Meyer, B., eds.: Proceedings, First International Conference on Tests and Proofs, Zurich, Switzerland. Volume 4454 of LNCS, Springer (2007) 207–216
7. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In Gurevich, Y., Meyer, B., eds.: Proceedings, First International Conference on Tests and Proofs, Zurich, Switzerland. Volume 4454 of LNCS, Springer (2007) 169–188

8. Nanchen, S., Schmid, H., Schmitt, P., Stärk, R.F.: The ASMKeY prover. Technical Report 436, Department of Computer Science, ETH Zürich and Institute for Logic, Complexity and Deduction Systems, Universität Karlsruhe (2003)
9. Giese, M., Larsson, D.: Simplifying transformations of OCL constraints. In Briand, L., Williams, C., eds.: Proceedings, Model Driven Engineering Languages and Systems, Montego Bay, Jamaica. Volume 3713 of LNCS, Springer (2005) 309–323
10. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In Maibaum, T., ed.: Proceedings, Third Internationsl Conference on Fundamental Approaches to Software Engineering. Volume 1783 of LNCS, Springer (2000) 363–366
11. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings, ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, ACM Press (2002) 234–245
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52** (2005) 365–473
13. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: A developer-oriented approach. In Araki, K., Gnesi, S., Mandrioli, D., eds.: Proceedings, 12th International Symposium on Formal Methods. Volume 2805 of LNCS, Springer (2003) 422–439
14. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm, W., Hermanns, H., eds.: Proceedings, 19th International Conference on Computer-Aided Verification, Berlin, Germany. Volume 4590 of LNCS, Springer (2007) 173–177
15. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Proceedings, 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2031 of LNCS, Springer (2001) 299–312
16. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education (2001)
17. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In Hermann, M., Voronkov, A., eds.: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia. Volume 4246 of LNCS, Springer (2006) 422–436
18. Mostowski, W.: Fully verified Java Card API reference implementation. In Beckert, B., ed.: Proceedings, Fourth International Verification Workshop, Bremen, Germany. Volume 259 of CEUR (`http://ceur-ws.org/`) (2007)
19. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing **19** (2007) 159–189

Paper 4

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

Philipp Rümmer

Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning
Phnom Penh, Cambodia, 2006
Volume 4246 of LNCS, Springer (2006) 422–436

This thesis contains an extended version of the paper.

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@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 makes it possible to reason about the relation between the initial and final 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 reads from memory. Sequent calculi for dynamic logic often represent memory using formulae 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-formulae Γ, Δ of a sequent $\Gamma \vdash \langle\, \alpha \,\rangle\, \phi, \Delta$, like in inequalities $0 \dot{<} x$ and equations $x' \doteq x + 1$.

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 determines parts of the program state, for instance the update $x := x + 1$. Compared with side-formulae, 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 Java 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 cover simplification and normalisation. The main contributions of the paper are new update constructs (in particular quantification), the development of a complete meta-theory of updates and its formalisation[1] using the Isabelle/HOL proof assistant [3], including proofs of all lemmas about updates 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 *max* 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:

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

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] `http://www.cse.chalmers.se/~philipp/updates.thy`, ≈ 3500 lines of Isabelle/Isar code

init can be read as a 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 $\leadsto$ and write $\equiv$ for an update simplification step. *init* is used as an abbreviation.

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

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

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

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

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

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

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

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, Formulae, 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 (minimalist)[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 formulae (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 allow 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, formulae and updates are defined by the following grammar, in which $x \in \mathit{Var}$ ranges over variables and $f \in \Sigma$ over functions:*

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

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 $\mathtt{if}\ \phi\ \{u\}$, and quantified updates $\mathtt{for}\ x\ \{u\}$. 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 formulae (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, Formulae, and Updates

The meaning of terms and formulae 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 $\mathit{Loc}_{(\Sigma,\alpha),U}$ of* locations *over (Σ, α) and U by*

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

[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: Every non-empty set* $A \subseteq U$ *contains a least 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* $Loc_{(\Sigma,\alpha),U} \rightharpoonup U$.

A (partial) function $f : M \rightharpoonup N$ 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 other specification languages. For two (partial or total) functions $f, g : M \rightharpoonup N$ 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:

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

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 formulae. 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.

Table 1. Evaluation of Terms, Formulae, and Updates

For terms:

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

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

For formulae:

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

For updates:

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

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

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

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

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

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

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.

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

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

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:*

$$\mathrm{val}_{S,\beta}(\texttt{for}\ x\ \{u\})(loc) \ = \ \mathrm{val}_{S,\beta_x^m}(u)(loc)$$

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

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

Definition 6. *We call two terms, formulae 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 formulae: 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 proven 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)
$\{u\}\ \begin{array}{r}\text{if } \phi \text{ then } t_1\\ \text{else } t_2\end{array}$	$\to$	$\begin{array}{l}\text{if } \{u\}\ \phi \text{ then } \{u\}\ t_1\\ \qquad\qquad \text{else } \{u\}\ t_2\end{array}$		(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)
$\textsc{non-rec}(f(\bar{s}) := r, f, \bar{t})$	$\to$	$\text{if } \bar{t} \doteq \bar{s} \text{ then } r \text{ else } f(\bar{t})$		(R11)
$\textsc{non-rec}(g(\bar{s}) := r, f, \bar{t})$	$\to$	$f(\bar{t})$	$(f \neq g)$	(R12)
$\textsc{non-rec}(u_1 \mid u_2, f, \bar{t})$	$\to$	$\begin{array}{ll}\text{if} & \textsc{in-dom}(f, \bar{t}, u_2)\\ \text{then} & \textsc{non-rec}(u_2, f, \bar{t})\\ \text{else} & \textsc{non-rec}(u_1, f, \bar{t})\end{array}$		(R13)
$\textsc{non-rec}(\texttt{if}\ \phi\ \{u\}, f, \bar{t})$	$\to$	$\begin{array}{ll}\text{if} & \phi\\ \text{then} & \textsc{non-rec}(u, f, \bar{t})\\ \text{else} & f(\bar{t})\end{array}$		(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)
$\textsc{in-dom}(f, \bar{t}, u_1 \mid u_2)$	$\to$	$\begin{array}{l}\textsc{in-dom}(f, \bar{t}, u_1)\\ \vee\ \textsc{in-dom}(f, \bar{t}, u_2)\end{array}$		(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 formulae, i.e., simplification of expressions $\{u\}\ \alpha$, is carried out in two phases: first, the update is propagated to subterms or subformulae. 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, formulae and updates as in Def. 1, but with further constructors ($x \in Var$ ranges over variables and $f \in \Sigma$ over functions):*

$$\begin{aligned} Ter_A &::= \cdots \,\big|\, \{x/Ter_A\}\ Ter_A \,\big|\, \textsc{non-rec}(Upd_A, f, (Ter_A, \ldots, Ter_A)) \\ For_A &::= \cdots \,\big|\, \{x/Ter_A\}\ For_A \,\big|\, \textsc{in-dom}(f, (Ter_A, \ldots, Ter_A), Upd_A) \\ Upd_A &::= \cdots \,\big|\, \{x/Ter_A\}\ Upd_A \,\big|\, \textsc{reject}(Upd_A, \overline{Upd_A}) \end{aligned}$$

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

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

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

$$\mathrm{val}_{S,\beta}(\textsc{non-rec}(u, f, \bar{t})) = I'\langle f, \mathrm{val}_{S,\beta}(\bar{t})\rangle\ ,$$

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

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

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

The difference between non-recursive application $\textsc{non-rec}(u, f, \bar{t})$ 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 $\{u\}\ f(\bar{t}) \equiv \textsc{non-rec}(u, f, \{u\}\ \bar{t})$. The non-recursive operator enables us to separate the syntactic propagation of updates to subterms and subformulae 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 $\{u_1\}\ \{u_2\}\ \alpha$. It seems natural to make an operator for sequential composition compatible with the 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, formulae and updates as in Def. 7, but with two further constructors:*

$$\mathit{Upd}_{AS}\ ::=\ \cdots \;\Big|\; \mathit{Upd}_{AS}\ ;\ \mathit{Upd}_{AS} \;\Big|\; \{\mathit{Upd}_{AS}\}\ \mathit{Upd}_{AS}$$

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 formulae. 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

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

Table 4. Rewriting Rules for Sequential Composition

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

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

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, formulae or updates, i.e., that rewriting is an equivalence transformation:

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

The second lemma characterises the form of terms, formulae 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 $\alpha \in \mathit{Ter}_{AS} \cup \mathit{For}_{AS} \cup \mathit{Upd}_{AS}$ 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., $\alpha \in \mathit{Ter} \cup \mathit{For} \cup \mathit{Upd}$. 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 be 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 formulae 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]

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

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.

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

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]

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

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

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

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

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

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):

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

(we assume that $\mathit{nil} \in \Sigma$ denotes invalid addresses and the beginning and end of lists). The update *init* from Sect. 2 and a list containing the numbers $0, \ldots, n$ can then be set up as follows:

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

[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:

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

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. The approach that is followed in KeY 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]

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

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]

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

Compared to a declarative specification of `arrayCopy` using a post-condition that contains a universally quantified formula, the imperative update can be applied to formulae 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 the 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 allows 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)

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

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

$$\{u_1\}\,\{u_2\}\,\alpha \;\equiv\; \{u_1\,;\,u_2\}\,\alpha \qquad \text{(R51)}$$

$$u_1 \mid (u_2 \mid u_3) \;\equiv\; (u_1 \mid u_2) \mid u_3 \qquad \text{(R52)}$$

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

$$u_1 \mid u_2 \;\equiv\; \textsc{reject}(u_1, \overline{u_2}) \mid u_2 \qquad \text{(R54)}$$

$$u_1 \mid u_2 \;\equiv\; u_2 \mid \textsc{reject}(u_1, \overline{u_2}) \qquad \text{(R55)}$$

$$u \;\equiv\; u \mid \texttt{if}\ \phi\ \{u\} \qquad (\phi \text{ arbitrary}) \quad \text{(R56)}$$

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

$$\texttt{if}\ \phi\ \{u_1 \mid u_2\} \;\equiv\; \texttt{if}\ \phi\ \{u_1\} \mid \texttt{if}\ \phi\ \{u_2\} \qquad \text{(R58)}$$

$$\texttt{if}\ \phi_1\ \{\texttt{if}\ \phi_2\ \{u\}\} \;\equiv\; \texttt{if}\ \phi_1 \wedge \phi_2\ \{u\} \qquad \text{(R59)}$$

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

$$\texttt{for}\ x\ \{\texttt{if}\ \phi\ \{u\}\} \;\equiv\; \texttt{if}\ \exists x.\,\phi\ \{u\} \qquad (x \notin \mathrm{fv}(u)) \quad \text{(R61)}$$

$$\texttt{for}\ x\ \{u_1 \mid u_2\} \;\equiv\; \texttt{for}\ x\ \{u_1\} \mid u_2 \qquad (x \notin \mathrm{fv}(u_2)) \quad \text{(R62)}$$

For $u = \texttt{for}\ z\ \{\texttt{if}\ z \dot{<} x\ \{\{x/z\}\ u_1\}\}$ and $z \neq x$, $z \notin \mathrm{fv}(u_1)$:

$$\texttt{for}\ x\ \{u_1\} \;\equiv\; \texttt{for}\ x\ \{\textsc{reject}(u_1, \overline{u})\} \qquad \text{(R63)}$$

$$\texttt{for}\ x\ \{u_1 \mid u_2\} \;\equiv\; \texttt{for}\ x\ \{u_1\} \mid \texttt{for}\ x\ \{\textsc{reject}(u_2, \overline{u})\} \qquad \text{(R64)}$$

For $u = \texttt{for}\ z\ \{\texttt{if}\ z \dot{<} x\ \{\{x/z\}\ \texttt{for}\ y\ \{u_1\}\}\}$ and $|\{x, y, z\}| = 3$, $z \notin \mathrm{fv}(u_1)$:

$$\texttt{for}\ x\ \{\texttt{for}\ y\ \{u_1\}\} \;\equiv\; \texttt{for}\ y\ \{\texttt{for}\ x\ \{\textsc{reject}(u_1, \overline{u})\}\} \qquad \text{(R65)}$$

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)

$$\begin{array}{l} \ \ \texttt{for}\ x_{1,1}\ \{\texttt{for}\ x_{1,2}\ \{\texttt{for}\ \cdots\ \{\texttt{if}\ \phi_1\ \{s_1 := t_1\}\}\}\} \\ \mid \cdots \\ \mid \texttt{for}\ x_{k,1}\ \{\texttt{for}\ x_{k,2}\ \{\texttt{for}\ \cdots\ \{\texttt{if}\ \phi_k\ \{s_k := t_k\}\}\}\} \end{array} \qquad (1)$$

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 make it possible 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

$\texttt{if}\ \phi\ \{\texttt{skip}\}$	$\to$	$\texttt{skip}$		(R66)	$\texttt{skip} \mid u$	$\to$	u	(R71)
$\texttt{if}\ \mathit{false}\ \{u\}$	$\to$	$\texttt{skip}$		(R67)	$u \mid \texttt{skip}$	$\to$	u	(R72)
$\texttt{if}\ \mathit{true}\ \{u\}$	$\to$	u		(R68)	$\texttt{skip}\,;\, u$	$\to$	u	(R73)
$\texttt{for}\ x\ \{\texttt{skip}\}$	$\to$	$\texttt{skip}$		(R69)	$u\,;\,\texttt{skip}$	$\to$	u	(R74)
$\texttt{for}\ x\ \{u\}$	$\to$	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. Disjointness is relevant for parallel composition (R55) and for quantification (R64), (R65), where permutation can change the order of assignments.

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 formulae, 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 formulae, however, because of the constructors $\min x.\,\phi$ and $\text{if}\ \ \phi\ \ \text{then}\ \ t_1\ \ \text{else}\ \ t_2$). We describe a procedure that turns every

[13] Application of (R51), (R52), (R58), (R59), (R60), (R64) from left to right, Table 6 as well as ordering sequences of parallel updates.

update u into an equivalent update

$$\begin{array}{l} \ \ \texttt{for}\ x_{1,1}\ \{\texttt{for}\ x_{1,2}\ \{\texttt{for}\ \cdots\ \{f_1(x_{1,1}, x_{1,2}, \ldots) := t_1\}\}\} \\ \mid \cdots \\ \mid \texttt{for}\ x_{k,1}\ \{\texttt{for}\ x_{k,2}\ \{\texttt{for}\ \cdots\ \{f_k(x_{k,1}, x_{k,2}, \ldots) := t_k\}\}\} \end{array} \tag{2}$$

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 $loc = \langle f, \bar{a} \rangle$: (i) J can be undefined at point loc (i.e., $J(loc) = \bot$), (ii) J can agree with the interpretation I at point loc (i.e., $J(loc) = I(loc) \neq \bot$), 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., $\bot \neq J(loc) \neq I(loc)$). 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, formulae or updates α, we have:

$$\{a := a\}\ \alpha \quad \equiv \quad \{\texttt{skip}\}\ \alpha \qquad \text{but} \qquad a := a \quad \not\equiv \quad \texttt{skip}$$

We define a coarser equivalence relation that identifies the cases (i) and (ii):

Definition 9. *Two updates $u_1, u_2 \in Upd_{AS}$ 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:

$$a := a \quad \equiv_{\mathrm{md}} \quad \texttt{skip}, \qquad (\texttt{for}\ x\ \{f(x) := f(x)\} \mid f(a) := b) \quad \equiv_{\mathrm{md}} \quad f(a) := b$$

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:

$$a := a \quad \equiv_{\mathrm{md}} \quad \texttt{skip} \qquad \text{but} \qquad a := b \mid a := a \quad \not\equiv_{\mathrm{md}} \quad a := b \mid \texttt{skip}$$

More generally, Table 7 contains a number of implications by which equivalence modulo definedness can be derived syntactically.

Normalisation of Updates: Surprisingly, the notion of equivalence modulo definedness allows to perform normalisation of updates u. By Table 7, we have:

$$v \quad = \quad \texttt{for}\ x_1\ \{\texttt{for}\ x_2\ \{\texttt{for}\ \cdots\ \{f(x_1, x_2, \ldots) := f(x_1, x_2, \ldots)\}\}\} \quad \equiv_{\mathrm{md}} \quad \texttt{skip}$$

Table 7. Compatibilities between $\equiv_{\text{md}}$ and Update Operators

	$f(\bar{t}) := f(\bar{t}) \quad \equiv_{\text{md}} \quad \texttt{skip}$	(R75)
	$u_1 \quad \equiv_{\text{md}} \quad u_1' \quad \text{implies} \quad u_1 \mid u_2 \quad \equiv_{\text{md}} \quad u_1' \mid u_2$	(R76)
	$u_1 \quad \equiv_{\text{md}} \quad u_1',\ u_2 \quad \equiv_{\text{md}} \quad u_2' \quad \text{implies} \quad u_1\,;\, u_2 \quad \equiv_{\text{md}} \quad u_1'\,;\, u_2'$	(R77)
	$u \quad \equiv_{\text{md}} \quad u' \quad \text{implies} \quad \texttt{if}\ \phi\ \{u\} \quad \equiv_{\text{md}} \quad \texttt{if}\ \phi\ \{u'\}$	(R78)
	$u \quad \equiv_{\text{md}} \quad \texttt{skip} \quad \text{implies} \quad \texttt{for}\ x\ \{u\} \quad \equiv_{\text{md}} \quad \texttt{skip}$	(R79)
For $\alpha \in \mathit{Ter}_{AS} \cup \mathit{For}_{AS} \cup \mathit{Upd}_{AS}$:		
	$u \quad \equiv_{\text{md}} \quad u' \quad \text{implies} \quad \{u\}\ \alpha \equiv \{u'\}\ \alpha$	(R80)

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":

$$\begin{array}{rll} & & u \\ (\text{R74}) & \equiv & u\,;\, \texttt{skip} \\ (\text{R77}) & \equiv_{\text{md}} & u\,;\, v \\ (\text{R45}) & \equiv & u \mid \{u\}\ v \\ (\text{R54}) & \equiv & \textsc{reject}(u, \overline{\{u\}\ v}) \mid \{u\}\ v \end{array}$$

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 normal form (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, formulae 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 Spec# [20] (the intermediate language BoogiePL of Spec# is inspired by guarded commands). 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.

In the future, 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, under-specification 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, Richard Bubel, and Steffen Schlager, 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.: Proceedings, First International Workshop on Java on Smart Cards: Programming and Security, Cannes, France. Volume 2041 of LNCS, Springer (2001) 6–24
10. Gedell, T., Hähnle, R.: Automating verification of loops by parallelization. In Hermann, M., Voronkov, A., eds.: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia. Volume 4246 of LNCS, Springer (2006) 332–346

11. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
12. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: Proceedings, 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Warsaw, Poland. 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, Third Asian Symposium on Programming Languages and Systems, Tsukuba, Japan. 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, ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, ACM Press (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.: Proceedings, International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, France. 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, 8th International Conference on Computer-Aided Verification, New Brunswick, USA. 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, Third International Joint Conference on Automated Reasoning, Seattle, USA. Volume 4130 of LNCS, Springer (2006) 266–280

Paper 5

The KeY System (Deduction Component)

Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov,
Philipp Rümmer, Steffen Schlager, and Peter H. Schmitt

Proceedings of the 21st International Conference on Automated Deduction
Bremen, Germany, 2007
Volume 4603 of LNCS, Springer (2007) 379–384

The version in this thesis contains minor typographic changes.

The KeY System (Deduction Component)

Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov, Philipp Rümmer, Steffen Schlager, and Peter H. Schmitt

www.key-project.org

Abstract. The KeY system is a development of the ongoing KeY project, whose aim is to integrate formal specification and deductive verification into the industrial software engineering processes. The deductive component of the KeY system is a novel interactive/automated prover for first-order Dynamic Logic for Java. The KeY prover features a user-friendly graphical interface, a backtracking-free free-variable sequent calculus, a simple and powerful theory formalisation language called "taclets," solution procedures for linear and non-linear integer arithmetic, external theorem prover integration, and facilities for proof reuse, among other aspects. The system is publicly available.

Introduction. The KeY system is the main software product of the KeY project, a joint effort between the University of Karlsruhe, Chalmers University of Technology in Göteborg, and the University of Koblenz. The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. At the core of the system is a deductive verification component, which also can be used as a stand-alone prover. It employs a free-variable sequent calculus for first-order Dynamic Logic for JAVA. The calculus is proof-confluent, i.e., no backtracking is necessary during proof search.

While we constantly strive to increase the degree of automation, user interaction remains indispensable in deductive program verification. The main design goal of the KeY prover is thus a seamless integration of automated and interactive proving. Efficiency must be measured in terms of user plus prover, not just prover alone. Therefore, a combination of a good user interface for proof state presentation and rule application, a high level of automation, extensibility of the rule base, and a calculus without backtracking is the strong point of KeY.

In this paper we concentrate on the description of the KeY prover and the reasoning techniques it employs. The prover consists of ca. 124,000 lines[1] of JAVA code. The standard rule base consists of 1,725 rules that are written in about 15,000 lines of KeY's "taclet" rule description language. About 1,300 of these formalise the semantics of the JAVA programming language. The system has been created by 14 implementors since 1999, who spent a total of about

[1] Not counting comments. These numbers are based on our estimates and the results of the SLOCCount tool (www.dwheeler.com/sloccount).

30 person years. Recently, version 1.0 of the KeY system has been released in connection with the KeY book [1]. The KeY tool is available under GPL and can be downloaded from `www.key-project.org`.

The KeY Program Verification System. KeY supports several languages for specifying properties of object-oriented models. Many people working with UML or model-driven development have familiarity with the specification language OCL (Object Constraint Language), a part of UML 2.0. Another supported specification language, which enjoys popularity among Java developers, is JML (Java Modelling Language). KeY can also translate OCL expressions to natural language (English and German).

The target programming language for verification in KeY is Java Card 2.2.1. KeY is the only publicly available verification tool that supports the full Java Card standard including the persistent/transient memory model of the card devices and the atomic transactions. Rich specifications of the Java Card API are available both in OCL and JML. Java 1.4 programs that respect the limitations of Java Card (no floats, no reflection, no dynamic class loading) can be verified as well. A first prototype for verifying (restricted) multi-threaded programs is also available.

The system is not a classical verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved. For loop- and recursion-free programs, symbolic execution typically is performed in a fully automated manner. Optional plugins to the popular Eclipse IDE and to the Borland Together CASE tool suite have been developed to lower the entry hurdle for users with no or little training in formal methods.

Syntax and Semantics of the KeY Logic. The foundation of the KeY logic is a typed first-order predicate logic with subtyping. This foundation is extended with parameterized modal operators $\langle p \rangle$ and $[p]$, where p can be any sequence of legal Java Card statements. The resulting multi-modal program logic is called Java Card Dynamic Logic or, for short, Java Card DL [1, Chapt. 3].

As is typical for dynamic logic, Java Card DL integrates programs and formulae within a single language. The modal operators refer to the final state of program p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state. For example, “when started in a state where `x` is zero, `x++;` terminates in a state where `x` is one” can be expressed as $\mathtt{x} \doteq 0 \mathrel{->} \langle \mathtt{x++;} \rangle (\mathtt{x} \doteq 1)$. The states used to interpret formulae are first-order structures sharing a common universe.

The type system of the KeY logic is designed to match the Java type system but can be used for other purposes as well. The logic includes *type casts* (changing the static type of a term) and *type predicates* (checking the dynamic type of a

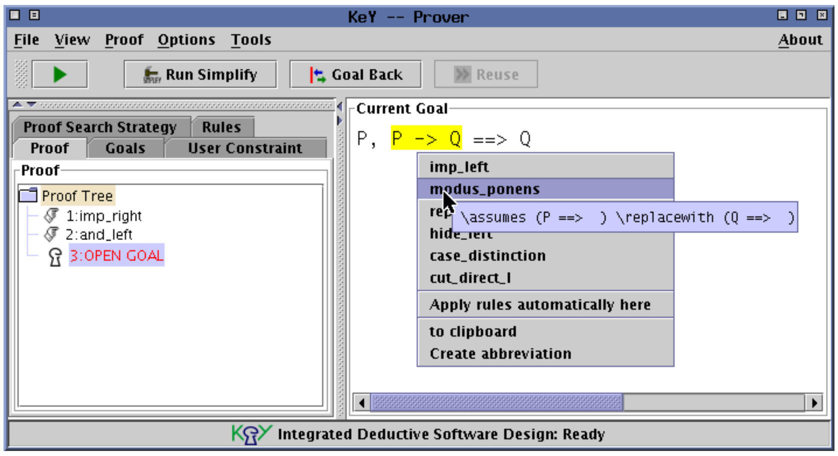


Fig. 1. Screenshot of the KeY prover user interface

term) in order to reason about inheritance and polymorphism [1, Chapter 2]. The type hierarchy contains the types such as boolean, the root reference type Object, and the type Null, which is a subtype of all reference types. It contains a set of user-defined types, which are usually used to represent the interfaces and classes of a given Java Card program. Finally, it contains several integer types, including both the range-limited types of Java and the infinite integer type $\mathbb{Z}$.

Beside built-in symbols (such as type-cast functions, equality, and operations on integers), user-defined functions and predicates can be added to the signature. They can be either *rigid* or *non-rigid*. Intuitively, rigid symbols have the same meaning in all program states (e.g., the addition on integers), whereas the meaning of non-rigid symbols may differ from state to state.

Finally, there is another kind of modal operators called *updates* (see [2] on page 115). They can be seen as a language for describing program transitions. There are simple function updates corresponding to assignments in an imperative programming language, which in turn can be composed sequentially and used to form parallel or quantified updates. Updates play a central role in KeY: the verification calculus transforms Java Card programs into updates. KeY contains a powerful and efficient mechanism for simplifying updates and applying them to formulae.

Rule Formalisation and Application. The user can easily interleave the automated proof search implemented in KeY and interactive rule application. For interactive rule application, the KeY prover has an easy to use graphical user interface that is built around the idea of direct manipulation (Fig. 1). To apply a rule, the user first selects a *focus of application* by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules applicable at this focus. This choice remains manageable even for very large rule bases. Rule schema variable instantiations are mostly inferred by matching.

Another simple way to apply rules and give instantiations is by drag and drop. If the user drags an equation onto a term the system will try to rewrite the term with the equation. If the user drags a term onto a quantifier the system will try to instantiate the quantifier with this term.

The interaction style is closely related to the way rules are formalised in the KeY prover. There are no hard-coded rules; all rules are defined in the *taclet language* instead. Besides the conventional declarative semantics, taclets have a clear operational semantics, as the following example shows—a "modus ponens" rule in textbook notation (left) and as a taclet (right):

$$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi, \phi \rightarrow \psi, \Gamma \vdash \Delta}$$

```
\find (p -> q ==>)          // implication in antecedent
\assumes (p ==>)            // side condition
\replacewith(q ==>)         // action on found focus
\heuristics(simplify)       // strategy information
```

The `find` clause specifies the potential application focus. The taclet will be offered to the user on selecting a matching focus and if the formula mentioned in the `assumes` clause is present in the sequent. The action clauses `replacewith` and `add` allow modifying (or deleting) the formula in focus, as well as adding additional formulae (not present here). The `heuristics` clause provides priority information to the parameterised automated proof search strategy.

The taclet language is quickly mastered and makes the rule base easy to maintain and extend. Taclets can be proven correct against a set of base taclets. A full account of the taclet language is given in [1].

Confluent Calculus. In order to simplify the proof construction, which is typically partly automated and partly interactive, we have developed and employ a proof confluent sequent calculus. This means that automated proof search does not require backtracking over rule applications, which is advantageous for analysing failed proof attempts. The automated search for quantifier instantiations uses rigid free variables (called meta variables) like in a free-variable tableau calculus. Instead of backtracking over meta-variable instantiations, instantiations are postponed to the point where the whole proof can be closed, and an incremental global closure check is used. To minimise the confusion of novice users, meta variables are not visible in normal interactive use, if the user provides all required instantiations. Rule applications requiring particular instantiations (unifications) of meta variables are handled by attaching unification constraints to the resulting formulae [1, Sects. 4.3 and 10.2.2]. Equations are handled by ordered rewriting (currently in an incomplete way, which we have not, however, found to be a limiting factor so far).

The taclet language is designed in such a way that the user can only write rules with local effects on sequents, and the handling of meta variables, skolemisation, constraints, etc. is (mostly) taken care of automatically, to reduce the risk of inadvertently introducing rules that are unsound or damage confluence.

Handling Arithmetics. As the theory of integer arithmetic is omnipresent in program verification, KeY directly provides a number of automatic solution and simplification procedures for different fragments of arithmetic (see [3] on page 149). All procedures are formulated in terms of taclets, which have been verified against a small set of base axioms. The implemented methods target both proving (showing that equations are unsolvable) and construction of counterexamples (finding solutions of equations) for ground integer formulae.

The most basic method is a sequent calculus formulation of integer Gaussian elimination, which is a complete method for solving linear equations. As a prerequisite of the procedure, integer expressions are always fully expanded and sorted. Linear inequalities are handled by Fourier-Motzkin variable elimination, which we combine with systematic case distinctions in order to obtain a complete procedure over the integers.

Reasoning in non-linear integer arithmetic is mainly carried out by heuristic cross-multiplication of inequalities, similar to the approach of the ACL2 prover. In order to reduce expressions as far as possible and handle non-linear equations more efficiently, KeY also computes Gröbner bases over the integers.

The KeY system also features a component for easy integration of external automated theorem provers and (semi-)decision procedures. Proof goals are translated into the standardised input format SMT-LIB and discharged by calling any tool that understands this format, such as Yices or CVC Lite. A similar connector for the theorem prover Simplify is also available. The user benefits from the particular abilities of these tools to decide fragments of arithmetics, heuristically instantiate quantifiers, etc.

Applications. The main application of the KeY prover is to support program verification in the KeY system. Among the major achievements in this field so far are the treatment of the Demoney case study (an electronic purse application provided by Trusted Logic S.A.) and the verification of a Java implementation of the Schorr-Waite graph marking algorithm. This algorithm, originally developed for garbage collectors, has recently become a popular benchmark for program verification tools. Chapters 14 and 15 of the KeY book [1] are devoted to a detailed description of these case studies. A case study [4] performed within the HIJA project has verified with KeY the lateral module of the flight management system, a part of the on-board control software from Thales Avionics.

Lately we have applied the KeY system also to issues of security analysis [5], and in the area of model-based test case generation [6, 7] where, in particular, the prover is used to compute path conditions and to identify infeasible paths. The flexibility of KeY w.r.t. the used logic and calculus further manifests itself in the fact that the prover has been chosen as a reasoning engine for a variety of other purposes. These include the mechanisation of a logic for Abstract State Machines [8] and the implementation of a calculus for simplifying OCL constraints [9].

KeY is also very useful for teaching logic, deduction, and formal methods. Its graphical user interface makes KeY easy to use for students. They can step through proofs with different degrees of automation (using the full verification calculus or just the first-order core rules). The authors have been successfully teaching courses for several years using the KeY system. An overview and course materials are available at `www.key-project.org/teaching`.

References

1. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2007)
2. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In Hermann, M., Voronkov, A., eds.: Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia. Volume 4246 of LNCS, Springer (2006) 422–436
3. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In Beckert, B., ed.: Proceedings, Fourth International Verification Workshop, Bremen, Germany. Volume 259 of CEUR (`http://ceur-ws.org/`) (2007)
4. Hunt, J.J., Jenn, E., Leriche, S., Schmitt, P., Tonin, I., Wonnemann, C.: A case study of specification and verification using JML in an avionics application. In Rochard-Foy, M., Wellings, A., eds.: Proceedings, Fourth Workshop on Java Technologies for Real-time and Embedded Systems, ACM Press (2006) 107–116
5. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In Hutter, D., Ullmann, M., eds.: Proceedings, Second International Conference on Security in Pervasive Computing. Volume 3450 of LNCS, Springer (2005) 193–209
6. Beckert, B., Gladisch, C.: White-box testing by combining deduction-based specification extraction and black-box testing. In Gurevich, Y., Meyer, B., eds.: Proceedings, First International Conference on Tests and Proofs, Zurich, Switzerland. Volume 4454 of LNCS, Springer (2007) 207–216
7. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In Gurevich, Y., Meyer, B., eds.: Proceedings, First International Conference on Tests and Proofs, Zurich, Switzerland. Volume 4454 of LNCS, Springer (2007) 169–188
8. Nanchen, S., Schmid, H., Schmitt, P., Stärk, R.F.: The ASMKeY prover. Technical Report 436, Department of Computer Science, ETH Zürich and Institute for Logic, Complexity and Deduction Systems, Universität Karlsruhe (2003)
9. Giese, M., Larsson, D.: Simplifying transformations of OCL constraints. In Briand, L., Williams, C., eds.: Proceedings, Model Driven Engineering Languages and Systems, Montego Bay, Jamaica. Volume 3713 of LNCS, Springer (2005) 309–323

Paper 6

A Sequent Calculus for Integer Arithmetic with Counterexample Generation

Philipp Rümmer

Proceedings of the 4th International Verification Workshop at CADE 21
Bremen, Germany, 2007
Volume 259 of CEUR (2007)

This thesis contains an extended version of the paper.

A Sequent Calculus for Integer Arithmetic with Counterexample Generation

Philipp Rümmer

Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University, Sweden
`philipp@chalmers.se`

Abstract. We introduce a calculus for handling integer arithmetic in first-order logic. The method is tailored to Java program verification and meant to be used both as a supporting procedure and simplifier during interactive verification and as an automated tool for discharging (ground) proof obligations. There are four main components: a complete procedure for linear equations, a complete procedure for linear inequalities, an incomplete procedure for nonlinear (polynomial) equations, and an incomplete procedure for nonlinear inequalities. The calculus is complete for the generation of counterexamples for invalid ground formula in integer arithmetic. All parts described here have been implemented as part of the KeY verification system.

1 Introduction

We introduce a Gentzen-style sequent calculus for integer arithmetic that is tailored to integrated automated and interactive Java software verification. The calculus was developed for dynamic logic for the Java language [1, Chapter 3] (a classical first-order logic) and integrates well-known as well as new algorithms, with the goal to provide the following features:

– Simplification of arithmetic expressions or formulae so that proof goals are kept small and readable. A calculus for this purpose should always terminate and should not cause proof splitting; completeness is secondary.
– Transparency and the ability to create human-readable proofs and sequences of simplification steps, otherwise it is difficult for a user to resume interactive proving after a number of automated proof steps. The fastest way to understand a proof goal is often to look at the history that led to the goal.
– Handling of nonlinear arithmetic guided by the user, which is necessary for programs that happen to contain multiplication or division operations.
– Generation of counterexamples for invalid formulae, which is useful during specification and when proving with induction and invariants.
– Handling of the actual modular Java integers, which are in our system modelled by a mapping to the mathematical integers [1, Chapter 12]. Reasoning in this setting requires good support for simplifying expressions, for instance

by (implicitly) proving the absence of overflows. The methods that we developed to this end are beyond the scope of the paper, but are based on the presented calculus.
- Simplicity, as interactive usage with a little expertise as possible is an important aspect of the KeY system.

Only some of these points can be solved using external procedures and theorem provers (which are, nevertheless, extremely useful for dealing with many proof obligations). As a complementary approach, we have developed a novel calculus for integer arithmetic that is directly implemented in our theorem prover KeY [1] in form of derived proof rules (i.e., the rules have been proven correct against a set of base rules). The rules are deliberately kept as elementary as possible and are here presented in sequent calculus notation. The calculus is driven by a proof strategy that controls the rule application and realises the following components: (i) a simplification procedure that works on single terms and formulae and is responsible for normalisation of polynomials (Sect. 2), (ii) a complete procedure for systems of linear equations, based on Gaussian elimination and the Euclidian algorithm (Sect. 3), (iii) a complete procedure for systems of linear inequalities, based on Fourier-Motzkin variable elimination (Sect. 4), (iv) an incomplete procedure for nonlinear (polynomial) equations, based on Gröbner bases (Sect. 5), (v) an incomplete procedure for nonlinear inequalities using cross-multiplication of inequalities and systematic case analysis (Sect. 6).

The development of the method was mostly an engineering process with the goal of handling cases that practically occur in Java program verification. It was successful in the sense that many proofs that before only were possible with the help of external provers can now be handled by KeY alone (e.g., the case study [2]), and that many proofs that before were impossible have become feasible.

We do not consider quantifiers or uninterpreted functions in this paper. The calculus is proof confluent (cf. [3]) and can basically be used in two different modes: (i) for simplification, which disables the handling of nonlinear inequalities, prevents case splits and guarantees termination (Procedure 12 in Sect. 5), and (ii) for proving and countermodel construction, which enables all parts (Procedure 14 in Sect. 6).

Introductory example. We start with an example and show how the following statement can be proven within our calculus (in the "full" mode):[1]

$$11a + 7b \doteq 1 \;\vdash\; \langle \texttt{b=a*c-1; if (c>=a) a=a/b;} \rangle\, \mathit{true} \qquad (1)$$

In Java dynamic logic, this sequent expresses that the program in angle brackets terminates normally, i.e., in particular does not raise exceptions, given the assumption $11a + 7b \doteq 1$. A proof is conducted by rewriting the program following the symbolic execution paradigm [4], whereby the calculus presented in this paper is permanently applied on the *path condition* (in (1), $11a + 7b \doteq 1$) and the *symbolic variable assignment* (in (1), the identity).

[1] On an Intel Pentium M processor with 1.6 GHz, the KeY implementation of the procedure needs about 460 inference steps and 2 seconds to find this proof.

$$5c \mathrel{\dot{\geq}} -7e-8,\ e \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash \tag{13}$$

$$ce \mathrel{\dot{\geq}} -c-e-1,\ e \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash \tag{12}$$

$$e \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash \tag{11}$$

$$\cdots \qquad c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash \tag{10}$$

$$\ldots,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash \tag{9}$$

$$a \doteq 7e+2,\ b \doteq -11e-3,\ c \mathrel{\dot{\geq}} 7e+2 \ \vdash \ 7ce+2c-1 \mathrel{\dot{\neq}} 0 \tag{8}$$

$$\cdots \qquad a \doteq 7e+2,\ b \doteq -11e-3,\ c \mathrel{\dot{\geq}} 7e+2 \ \vdash \ \{b := 7ce+2c-1\}\langle \texttt{a=a/b;} \rangle\ \mathit{true} \tag{7}$$

$$a \doteq 7e+2,\ b \doteq -11e-3 \ \vdash \ \{b := 7ce+2c-1\}\langle \texttt{if (c>=a) a=a/b;} \rangle\ \mathit{true} \tag{6}$$

$$a \doteq 7e+2,\ b \doteq -11e-3 \ \vdash \ \{b := a \cdot c-1\}\langle \texttt{if (c>=a) a=a/b;} \rangle\ \mathit{true} \tag{5}$$

$$a \doteq 7e+2,\ b \doteq -11e-3,\ d \doteq 3e+1 \ \vdash \ \langle \texttt{b=a*c-1; if (c>=a) a=a/b;} \rangle\ \mathit{true} \tag{4}$$

$$3a \doteq 7d-1,\ b \doteq -2a+d \ \vdash \ \langle \texttt{b=a*c-1; if (c>=a) a=a/b;} \rangle\ \mathit{true} \tag{3}$$

$$7b \doteq -11a+1 \ \vdash \ \langle \texttt{b=a*c-1; if (c>=a) a=a/b;} \rangle\ \mathit{true} \tag{2}$$

$$11a+7b \doteq 1 \ \vdash \ \langle \texttt{b=a*c-1; if (c>=a) a=a/b;} \rangle\ \mathit{true} \tag{1}$$

Fig. 1. Proof tree for the introductory example

The complete proof is shown in Fig. 1. As first step, all formulae are normalised: we choose an arbitrary well-ordering $<_r$ on the variables in the problem ($a <_r b <_r c$) and move big variables to the left and small variables to the right of the relations $\doteq$, $\dot{\leq}$, $\dot{\geq}$, resulting in (2). We then concentrate on the equation in (2), in order to (eventually) turn the leading coefficient 7 into a 1, by means of the extended Euclidian algorithm [5]). A basis transformation is performed that replaces b with a fresh variable d (such that $a <_r b <_r c <_r d$). One can minimise the coefficient of $11a$ by choosing $b \doteq -2a + d$ and replace the occurrence of b in the original equation with $-2a + d$ (afterwards, the equation is again normalised, sequent (3)). Because the leading coefficient of the first equation is still not 1, a second basis transformation $a \to 2d + e$ is performed (with $d <_r e$). This turns the leading coefficients of all equations into 1 (sequent (4)).

We can now leave out the equation $d \doteq 3e + 1$, because d does not occur in the sequent anymore. No further inferences are possible in the path condition and the first statement of the program is executed, which means that the variable assignment has to be updated accordingly (for simplicity, we assume that no overflows are possible). The assignment $b := 7ce + 2c - 1$ is written in front of the program in (5) and is rewritten and simplified using the equations in (6). The next program statement causes the proof to split on the condition $c \dot{\geq} a$. One branch ($c \dot{<} a$) can immediately be closed because the program contains no further statements. On the other branch (7), we obtain a new assumption $c \dot{\geq} a$ that can be simplified.

The execution of the last assignment yields a new proof obligation (8) in order to prevent division by zero. We prove by contradiction and normalise the new equation in (9) (and also leave out the first two equations, which are no longer needed for the proof). Because all other possibilities fail in the resulting situation, a case split on the sign of one of the "independent" variables c or e is performed. Here, we will choose c and consider the cases $c \dot{\leq} -1$, $c \doteq 0$, and $c \dot{\geq} 1$. The case $c \doteq 0$ contradicts $7ce \doteq -2c + 1$, and the other two cases can be handled in essentially the same way, so we show only the first one in (10).

By transitivity, from the two inequalities in (10) the inequality $7e + 2 \dot{\leq} -1$ can be derived, which is rounded to $e \dot{\leq} -1$ in (11). No further linear inference steps are possible, but we can at this point deduce properties of product ce by *cross-multiplying* the inequalities $e \dot{\leq} -1$ and $c \dot{\leq} -1$, which yields the new inequality $0 \dot{\leq} (-c - 1) \cdot (-e - 1)$ in (12). After multiplying this inequality with 7, it can in (13) be rewritten using the equation $7ce \doteq -2c + 1$ and turned into $-2c + 1 \dot{\geq} 7 \cdot (-c - e - 1) \Leftrightarrow 5c \dot{\geq} -7e - 8$.

Now, a contradiction can be derived by reasoning about linear inequalities. From $5c \dot{\geq} -7e - 8$ and $c \dot{\leq} -1$ we derive $7e \dot{\geq} -3$, which is rounded to $e \dot{\geq} 0$ and a contradiction to $e \dot{\leq} -1$.

2 Normalisation of Arithmetic Expressions

Before starting a derivation and permanently during a proof, our calculus normalises (atomic) formulae. This was already demonstrated in the introductory example, and in a proof tree we denote such simplification steps with SIMP. We always fully expand polynomial expressions and represent them as a sum of monomials $\alpha_1 \cdot m_1 + \cdots + \alpha_n \cdot m_n$, in which $\alpha_1, \ldots, \alpha_n$ are non-zero integer literals and $m_1, \ldots, m_n$ are pairwise distinct products of variables (possibly 1 as the empty product, and possibly 0 as the empty sum). Full expansion is in general obviously a bad idea, but we found that it is a reasonable approach in interactive Java program verification that in the vast majority of cases improves the readability of formulae.

Sorting Terms. We put polynomial expressions into a canonical form by ordering the factors in a monomial and the monomials in a polynomial. The ordering $<_r$ that is used in both cases is a strict monomial ordering [6, 7]:

– We assume that a graded monomial ordering $<_r$ [6, 7] on products of variables is given, i.e., a well-ordering (a total, well-founded ordering) with the properties: (i) $\deg m < \deg m'$ implies $m <_r m'$, and (ii) $m <_r m'$ implies $x \cdot m <_r x \cdot m'$ for all variables x. In practice, we define $<_r$ as a graded lexicographic ordering: we assume that a well-ordering $<_r$ on variables[2] is given and then define $c_1 \cdots c_n <_r d_1 \cdots d_k$ if and only if $n < k$ or $n = k$ and $\{\!\{c_1, \ldots, c_n\}\!\} <_r \{\!\{d_1, \ldots, d_k\}\!\}$ (in the multiset extension of $<_r$ [9]).
– We extend $<_r$ by constructing a well-ordering on integer literals: $0 <_r 1 <_r -1 <_r 2 <_r -2 <_r 3 <_r \cdots$.
– We extend $<_r$ on monomials by $\alpha \cdot m <_r \alpha' \cdot m'$ if and only if $m <_r m'$ or $m = m'$ (modulo associativity and commutativity of $\cdot$) and $\alpha <_r \alpha'$.
– We extend $<_r$ on polynomials by $\alpha_1 m_1 + \cdots + \alpha_n m_n <_r \alpha'_1 m'_1 + \cdots + \alpha'_k m'_k$ if and only if $\{\!\{\alpha_1 m_1, \ldots, \alpha_n m_n\}\!\} <_r \{\!\{\alpha'_1 m'_1, \ldots, \alpha'_n m'_n\}\!\}$ (again using the multiset extension of $<_r$).

For sake of brevity, we will also compare arbitrary terms with $<_r$ and implicitly assume that the terms are first normalised.

Normalisation of Formulae. Atomic formulae are always written in the form $\alpha s * t$ with $* \in \{\dot{\leq}, \doteq, \dot{\geq}\}$, employing equivalences like $s \mathrel{\dot{<}} t \Leftrightarrow s + 1 \mathrel{\dot{\leq}} t$, and transformed so that the left-hand side αs is the $<_r$-greatest monomial of the polynomial $\alpha s - t$ and $\alpha > 0$. Furthermore, all inequalities are moved to the antecedent, and in case $\alpha s - t$ is a constant polynomial an equation or inequality is directly replaced with *true* or *false*.

We always demand that the coefficients of non-constant terms in an equation or inequality are coprime (do not have non-trivial factors in common), and otherwise divide all coefficients by the greatest common divisor. This also detects that equations like $2y \doteq 1 - 6c$ are unsolvable and equivalent to *false*, and that an inequality like $2y \mathrel{\dot{\leq}} 1 - 6c$ can be simplified and rounded to $y \mathrel{\dot{\leq}} -3c$ thanks to the discreteness of the integers.

[2] In reality, instead of variables we have to deal with arbitrary terms whose head-symbol is not $+$ or $\cdot$, which are compared with a lexicographic path ordering [8].

Finally, we add a simple subsumption check for inequalities that eliminates an inequality $s \dot{\leq} t$ from the antecedent in case there is a second inequality $s \dot{\leq} t - \beta$ with $\beta \geq 0$ (correspondingly for $\dot{\geq}$).

3 Equation Handling: Gaussian Variable Elimination

In contrast to many decision procedures or SMT provers, we define a calculus for handling linear equations that works independently from the inequality reasoning. The initial reason for this was that we believe that a reduction of equations to inequalities is not an option for interactive proving. Much later we became aware that we also can design more efficient, elegant and practical calculi for linear integer equations than for inequalities, which afterwards justifies the decision. We believe that this is also an important insight when working with the modular Java arithmetic, where the handling of such equations is essential. The sequent calculus described in this section is based on Gaussian elimination and the Euclidian algorithm.[3] It is complete, does not involve proof splitting, and is fast for all problems and benchmarks that we so far have looked at.

Row Operations. The primary rule of the calculus reduces an expression with the help of an equation in the antecedent. The application of the rule is only allowed if s' is not a subterm of $s \doteq t$ (u is an arbitrary term):[4]

$$\frac{\Gamma, s \doteq t \ \vdash\ \phi[s' + u \cdot (s - t)], \Delta}{\Gamma, s \doteq t \ \vdash\ \phi[s'], \Delta}\ \textsc{red} \qquad \text{if } s' + u \cdot (s-t) <_r s'$$

Example 1. We show how the rules RED and SIMP are used to solve a system of linear equations (with the ordering $x <_r y$):

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{x \doteq -5,\ y \doteq -1 \ \vdash\ x \doteq -5}}{3y \doteq x + 2,\ y \doteq -1 \ \vdash\ x \doteq -5}\ \textsc{red, simp}}{3y \doteq x+2,\ 5y - (3y - x - 2) \doteq x \ \vdash\ x \doteq -5}\ \textsc{simp}}{3y \doteq x + 2,\ 5y \doteq x \ \vdash\ x \doteq -5}\ \textsc{red}$$

Column Operations. It is well-known that the rule RED alone is not a complete calculus for integer equations. An example is the formula $11a + 7b \doteq 1$ in the introductory example, for which no reduction steps are possible. To obtain a complete calculus, we also perform *column operations*—referring to the usual matrix representation of the Gaussian elimination method. Assuming that no more applications of RED are possible in a sequent, and given an equation

[3] The calculus is in parts inspired by [5, Chapter 4.5.2], but in contrast to [5] we perform both row and column operations.

[4] In the rule, we write $\phi[s']$ in the succedent to denote that the term s' can occur in an arbitrary position in the sequent, in particular also in the antecedent.

$\alpha x \doteq s$ of the antecedent, we introduce a fresh unknown x' and perform a basis transformation $x \to u + x'$:

$$\frac{\Gamma, \alpha \cdot (u + x') \doteq s,\ x \doteq u + x' \ \vdash\ \Delta}{\Gamma, \alpha x \doteq s \ \vdash\ \Delta}\ \textsc{col-red}$$

if: x a variable, $\alpha > 1$, $(s - \alpha u) = \min_{<_r} \{s - \alpha u' \mid u' \text{ a term}\}$,
x' a fresh variable, $<_r$-smaller than all previous symbols

The term u is chosen such that the difference $s - \alpha u$ becomes $<_r$-minimal. One subsequent application of SIMP will thus turn the new equation $\alpha(u + x') \doteq s$ into a formula $\beta y \doteq t$ with $\beta <_r \alpha$. Likewise, βy is $<_r$-smaller than the left-hand sides of other equations $\beta' y = t'$, because RED was applied exhaustively prior to COL-RED. This ensures the overall termination of the procedure (Lem. 3 below) and allows to continue with reduction steps as long as linear equations are present whose left-hand side has a non-unit-coefficient.

We do not apply the rule COL-RED to nonlinear equations, due to the experience that the basis transformations performed by COL-RED cause more harm than good in the nonlinear setting. This is because the usage of a good monomial ordering $<_r$ becomes far more important than in the linear setting (COL-RED effectively alters the ordering by introducing a new smallest variable, possibly in a harmful way). We further discuss this issue in Sect. 5.

Procedure 2. *Apply* SIMP *with the highest priority,* RED *with second-highest priority, and* COL-RED *with the lowest priority.*

Lemma 3. *Procedure 2 terminates (for sequents containing arbitrary equations and inequalities). For sequents that only contain linear equations, it is complete and proof confluent.*

Proof on page 166

Example 4. If a proof branch does not get closed by this procedure, the remaining equations are an explicit description of all solutions (counterexamples) of the equations:

$$\frac{\begin{array}{c} x_0 \doteq 125x_3'' - 4,\ x_1 \doteq 25x_3'' - 1,\ x_2 \doteq 20x_3'' - 1,\ x_3 \doteq 16x_3'' - 1, \\ x_0' \doteq 16x_3'',\ x_3' \doteq -3x_3'' \end{array} \vdash}{\begin{array}{c} \vdots \\ x_0 \doteq 5x_1 + 1,\ 4x_1 \doteq 5x_2 + 1,\ 4x_2 \doteq 5x_3 + 1 \ \vdash \end{array}}$$

The equations that define x_0' and x_3' can be removed afterwards, because these symbols do not occur in the original problem and have no impact on its validity. A concrete counterexample is obtained by assigning arbitrary values to the variables that only occur in the right-hand sides of equations (x_3'').

4 Handling of Linear Inequalities: Fourier-Motzkin Variable Elimination and Case Splits

Although Fourier-Motzkin variable elimination [10] generally has a high complexity, it is one of the most popular methods to handle linear inequalities and used in proof assistants like PVS [11], Coq [12], Isabelle [13] or ACL2 [14, 15] and in the SMT-solver CVCLite [16]. We found Fourier-Motzkin to be a suitable base method both for linear and nonlinear inequality handling: most reasoning during verification is rather shallow and most inequalities only share symbols with a small number of other inequalities (sparse constraints), which is a situation where Fourier-Motzkin works well. At the same time, the Fourier-Motzkin elimination rule is suited for interactive proving due to its simplicity and the fact that it directly works on integers, in contrast to more efficient linear programming techniques. The full procedure given in this section is complete over the integers, but it involves proof splitting and does usually not terminate for invalid sequents, which means that it cannot (directly) be used as a simplifier for interactive proving. We therefore also identify a subset of the method that does not cause splitting and always terminates, but which is no longer complete (which hardly ever matters in practice). An example for a program that can be verified using the incomplete procedure (together with axioms for division, modulo and Java arithmetic) is shown in Fig. 2.

The Incomplete Procedure. As equations have already been handled in the previous section, we can implement Fourier-Motzkin with a single rule for "cancelling" two inequalities:

$$\frac{\Gamma, \alpha s \dot{\geq} t, \beta s \dot{\leq} t', \beta t \dot{\leq} \alpha t' \ \vdash\ \Delta}{\Gamma, \alpha s \dot{\geq} t, \beta s \dot{\leq} t' \ \vdash\ \Delta}\ \textsc{fm-elim} \qquad \text{if } \alpha > 0,\ \beta > 0$$

The resulting inequality $\beta t \dot{\leq} \alpha t'$ does no longer contain the monomial s and is therefore $<_r$-smaller than both previous inequalities (after a subsequent application of SIMP). To ensure termination, the rule must never be applied twice on a proof branch to the same pair of inequalities.

The performance of Fourier-Motzkin can be improved by adding a rule that turns two inequalities into an equation, based on the law of anti-symmetry:

$$\frac{\Gamma, s \doteq t \ \vdash\ \Delta}{\Gamma, s \dot{\leq} t, s \dot{\geq} t \ \vdash\ \Delta}\ \textsc{anti-symm}$$

Procedure 5. *Apply Procedure 2 (linear equations) with the highest priority, the rule* ANTI-SYMM *with second highest priority and the rule* FM-ELIM *with lowest priority.*

Proof on page 166

Lemma 6. *The procedure obtained in this way terminates when applied to a sequent containing arbitrary equations and inequalities.*

The Complete Procedure. Fourier-Motzkin is complete for rationals, but incomplete for integers. Our calculus is already more complete than pure Fourier-Motzkin due to the normalisation from Sect. 2 (rounding of inequalities) and the different equation handling of Procedure 2, which are enough to handle many cases that occur in practice (e.g., to show the inconsistency of $4x \dot{\geq} 5 \wedge 4x \dot{\leq} 7$).

Making the calculus actually complete has therefore not been of great importance for us. The following approach to this end is rather simplistic, but it has a counterexample generation property that is practically more relevant.

Our calculus becomes complete by performing a systematic case analysis, i.e., by doing proof splitting, in a way similar to Gomory's cutting-planes [10]. This is realised by the following rule for investigating the borderline case of an inequality:

$$\frac{\Gamma, s \mathrel{\dot{<}} t \vdash \Delta \quad \Gamma, s \doteq t \vdash \Delta}{\Gamma, s \mathrel{\dot{\leq}} t \vdash \Delta}\ \textsc{strengthen}$$

There is a corresponding rule for $\dot{\geq}$. The application of these rules does obviously not terminate in general, but it does for valid sequents (of linear inequalities), provided that a fair application strategy[5] is used and the rule is combined with Procedure 5. For an invalid sequent, a fair strategy eventually produces goals in which all inequalities have been replaced with equations and where Procedure 2 can take over and produce a counterexample.

Case distinctions are also necessary to handle equations in the succedent:

$$\frac{\Gamma \vdash s \mathrel{\dot{\leq}} t, \Delta \quad \Gamma \vdash s \mathrel{\dot{\geq}} t, \Delta}{\Gamma \vdash s \doteq t, \Delta}\ \textsc{split-eq}$$

Procedure 7. *Apply Procedure 5 (the incomplete method) with the highest priority, the rule* SPLIT-EQ *with second highest priority, and the rule* STRENGTHEN *with lowest priority and in a fair manner.*

Lemma 8. *This procedure is complete and proof confluent, and it eventually produces a counterexample for an invalid sequent.*

$\boxed{\text{Proof on page 167}}$

Example 9. Consider the following example taken from [18]: Because Procedure 5 is not able to derive a contraction, we apply STRENGTHEN to $x \mathrel{\dot{\leq}} 2$ and obtain two cases $x \doteq 1$, $x \doteq 2$ (thanks to ANTI-SYMM), the second of which leads to a counterexample:

$$\cfrac{\begin{array}{cc}
\cfrac{\cfrac{*}{y \mathrel{\dot{\geq}} 1,\ y \mathrel{\dot{\leq}} 0,\ x \doteq 1 \vdash}\ \textsc{fm-elim}}{\vdots} &
\cfrac{\cfrac{y \doteq 1,\ x \doteq 2 \vdash}{y \mathrel{\dot{\geq}} 1,\ y \mathrel{\dot{\leq}} 1,\ x \doteq 2 \vdash}\ \textsc{anti-symm}}{\vdots} \\
4y \mathrel{\dot{\geq}} x+1,\ 4y \mathrel{\dot{\leq}} x+2,\ x \doteq 1 \vdash &
4y \mathrel{\dot{\geq}} x+1,\ 4y \mathrel{\dot{\leq}} x+2,\ x \doteq 2 \vdash
\end{array}}{\cfrac{\vdots}{4y \mathrel{\dot{\geq}} x+1,\ 4y \mathrel{\dot{\leq}} x+2,\ x \mathrel{\dot{\leq}} 2,\ x \mathrel{\dot{\geq}} 1 \vdash}\ \textsc{strengthen}}$$

[5] In the presence of subsumption checks (Sect. 2), we consider a strategy as fair if STRENGTHEN is eventually applied to each inequality or to any subsuming inequality.

Java + JML

```
/*@
  @ normal_behavior
  @ requires -Decimal.PRECISION < f && f < Decimal.PRECISION
  @          && e + intPart < 32767 && -32768 < e + intPart;
  @ requires -Decimal.PRECISION < decPart && decPart < Decimal.PRECISION;
  @ modifiable intPart, decPart;
  @ ensures intPart * Decimal.PRECISION + decPart ==
  @        (\old(intPart) + e) * Decimal.PRECISION + \old(decPart) + f;
  @ ensures -Decimal.PRECISION < decPart && decPart < Decimal.PRECISION;
  @*/
public void add(short e, short f) {
  intPart += e;
  if ( intPart > 0 && decPart < 0 ) {
    intPart--; decPart = (short)( decPart + PRECISION );
  } else if ( intPart < 0 && decPart > 0 ) {
    intPart++; decPart = (short)( decPart - PRECISION ); }
  decPart += f;
  if ( intPart > 0 && decPart < 0 ) {
    intPart--; decPart = (short)( decPart + PRECISION );
  } else if ( intPart < 0 && decPart > 0 ) {
    intPart++; decPart = (short)( decPart - PRECISION );
  } else {
    short retenue = 0; short signe = 1;
    if ( decPart < 0 ) {
      signe = -1; decPart = (short)( -decPart ); }
    retenue = (short)( decPart / PRECISION );
    decPart = (short)( decPart % PRECISION );
    retenue *= signe; decPart *= signe; intPart += retenue;
  } }
```

Java + JML

Fig. 2. Addition method of class `Decimal` taken from [17], where it was verified using the Loop tool and PVS [11]. This method is part of a JavaCard Purse applet by Gemplus (`http://www.gemplus.com`). Using the KeY implementation of our calculus, it takes about 45 seconds and 23000 rule applications to automatically verify that the method adheres to its specification, reasoning about the modular arithmetic of Java (on an Intel Core 2 Duo CPU at 2.50GHz).

5 Handling of Nonlinear Polynomial Equations: Pseudo-Reduction and Gröbner Bases

The valid systems of integer equations or inequalities over arbitrary (possibly nonlinear) polynomials are known not to be recursively enumerable [19]. This means that all rules and procedures that we give from now on can never be complete and have been employed or developed with the aim of handling the common cases: when verifying programs, a large amount of the occurring nonlinear proof obligations can and should be taken care of automatically by incomplete calculi. The most important step to this end is to normalise nonlinear expressions (Sect. 2). We describe a comparatively cheap extension—that does not cause any proof splitting and is suited for interactive proving—of Procedure 2 to deal with nonlinear equation.

Pseudo-Reduction. As in Sect. 3, the primary rule for rewriting with (nonlinear) equations is RED. Because we do not apply the rule COL-RED to nonlinear equations, however, there are cases where equations $\alpha s \doteq t$ with $\alpha > 1$ remain in the antecedent that cannot be simplified further. In the sequent $x \dot{\geq} 1,\ y \dot{\geq} 1,\ 2z^2 \doteq y \ \vdash\ xz^2 \dot{\leq} xy$, for instance, none of the rules so far can be applied. In order to handle such cases, we introduce a further reduction rule that is based on pseudo-division and works by first multiplying the target expression with a constant [5]. The rule must only be applied if $\alpha s \doteq t$ and $u \cdot t \doteq \alpha t'$ are different equations:

$$\frac{\Gamma, \alpha s \doteq t \ \vdash\ \phi[u \cdot t \doteq \alpha t'], \Delta}{\Gamma, \alpha s \doteq t \ \vdash\ \phi[s' \doteq t'], \Delta}\ \textsc{pseudo-red} \qquad \text{if} \quad \deg s > 1,\ \alpha > 1,\ s' = u \cdot s$$

There are similar rules for inequalities $s' \dot{\leq} t'$, $s' \dot{\geq} t'$. We apply PSEUDO-RED only if the left-hand side of the equation $\alpha s \doteq t$ is nonlinear and $\alpha > 1$. Otherwise, the normal reduction rule RED can be used, possibly after turning α into 1 with help of COL-RED.

Gröbner Bases. Rewriting with nonlinear equations using the rules RED and PSEUDO-RED is not confluent and is not able to decide ideal membership in a ring of polynomials. Ideal membership is an approximation of semantic entailment of (nonlinear) equations that we can practically decide: we complete the set of antecedent equations by computing a Gröbner basis [6].

Example 10. When starting with a wrong application of PSEUDO-RED in the following example, a dead end is reached and no further reductions are possible:

$$2xy \doteq a,\ 2yz \doteq b,\ az \doteq 2 \ \vdash\ xyz \doteq 1$$

The simplest way to generate a Gröbner basis is to saturate the antecedent with "S-polynomial"-equations by considering all critical pairs of existing integer equations—the Buchberger algorithm [6]. Our calculus produces a non-reduced

Gröbner basis over the field of rational numbers that only consists of polynomial equations with integer coefficients, which are easier to compute and almost as useful for reduction as actual Gröbner bases over the integers. Given two equations with overlapping left-hand sides, S-polynomials are added as follows:

$$\frac{\Gamma, s \doteq t, s' \doteq t', s'_r \cdot t \doteq s_r \cdot t' \ \vdash \ \Delta}{\Gamma, s \doteq t, s' \doteq t' \ \vdash \ \Delta} \ \textsc{s-poly} \qquad \begin{array}{l} s = \gcd(s, s') \cdot s_r, \\ s' = \gcd(s, s') \cdot s'_r, \\ 0 < \deg s_r < \deg s, \\ 0 < \deg s'_r < \deg s' \end{array}$$

Similarly to the Fourier-Motzkin elimination rule, this rule must not be applied repeatedly for the same pair of equations to ensure termination. The performance of this naive implementation of Buchberger's algorithm is not comparable with more advanced methods, of course. We have yet to find, however, a verification problem where this would be a problem.

Example 11. We can solve Ex. 10 as follows. After the second application of S-POLY, the antecedent equations form a Gröbner basis:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{*}{2xy \doteq a,\ 2yz \doteq b,\ az \doteq 2,\ bx \doteq 2,\ ab \doteq 4y\ \vdash\ bx \doteq 2}}{2xy \doteq a,\ 2yz \doteq b,\ az \doteq 2,\ bx \doteq 2,\ ab \doteq 4y\ \vdash\ xyz \doteq 1}\ \textsc{pseudo-red}}{2xy \doteq a,\ 2yz \doteq b,\ az \doteq 2,\ bx \doteq 2\ \vdash\ xyz \doteq 1}\ \textsc{s-poly}}{2xy \doteq a,\ 2yz \doteq b,\ az \doteq 2,\ az \doteq bx\ \vdash\ xyz \doteq 1}\ \textsc{red, simp}}{2xy \doteq a,\ 2yz \doteq b,\ az \doteq 2\ \vdash\ xyz \doteq 1}\ \textsc{s-poly}$$

Procedure 12. *Apply Procedure 2 (linear equations) with highest priority, the rule* PSEUDO-RED *with second highest priority, the rule* S-POLY *with third highest priority, and Procedure 5 (linear inequalities) with lowest priority.*

Proof on page 169

Lemma 13. *Procedure 12 terminates when applied to a sequent containing arbitrary equations and inequalities.*

6 Handling of Nonlinear Polynomial Inequalities: Cross-Multiplication and Case Splits

The handling of nonlinear polynomial inequalities is realised as an extension of the linear inequality handling (Sect. 4). In order to apply linear reasoning to nonlinear arithmetic, we generate linear approximations of products and incrementally strengthen the precision of the approximations through case distinctions. Likewise, case splits are used to ensure the *existence* of linear approximations. Our method has been developed as a heuristic, and we do not have an exact description of the fragment of nonlinear arithmetic that it can handle. The main application areas where the method has proven to be extremely useful are correctness proofs for lemma rules that can be loaded by the prover KeY [1], and the verification of programs with the actual modular integer semantics of Java.

Similarly to the approach in ACL2 [15, 20] (and using their terminology), the primary rule to handle nonlinear inequalities is *cross-multiplication:*

$$\frac{\Gamma, s \dot{\leq} t, s' \dot{\leq} t', 0 \dot{\leq} (t-s) \cdot (t'-s') \ \vdash \ \Delta}{\Gamma, s \dot{\leq} t, s' \dot{\leq} t' \ \vdash \ \Delta} \ \textsc{cross-mult}$$

There are corresponding rules for $\dot{\geq}$ and for mixed pairs of inequalities. As usual in order to ensure termination, CROSS-MULT must not be applied repeatedly to the same pair of inequalities.

We can give a geometric interpretation of cross-multiplication: for two linear inequalities $x \dot{\leq} \alpha$, $y \dot{\leq} \beta$, cross-multiplication introduces a linear approximation of the product (the bilinear term) xy. In this particular case, the right-hand side of the new inequality $xy \dot{\geq} \beta x + \alpha y - \alpha\beta$ is the greatest plane that bounds the expression xy from below (under the assumptions $x \dot{\leq} \alpha$, $y \dot{\leq} \beta$). More generally, the result of cross-multiplication is a bound on the value of a monomial in terms of $<_r$-smaller monomials. Deriving such bounds is, in practical cases, often sufficient to prove statements in nonlinear arithmetic.

Restricting Cross-Multiplication. An unrestricted application of the rule CROSS-MULT can produce arbitrarily many inequalities and does not terminate. As a heuristic, we only use CROSS-MULT if the product $s \cdot s'$ already occurs as a factor within a left-hand side of an equation or inequality (ignoring the coefficient of $s \cdot s'$). Although this is not strong enough to ensure termination, it guarantees that the total degree of occurring monomials is bounded. We found this heuristic to work reasonably well for most cases (a counterexample is Ex. 16 below).

Case Splits. For two reasons, it is crucial to combine cross-multiplication with case distinctions: (i) nonlinear monomials over the complete set of integers do in general not have linear bounds (observe, for instance, that the term xy is not bounded from above or below by any linear expression in x and y). (ii) case distinctions are in general the only way to strengthen linear bounds (again, consider the term xy under the assumptions $x \dot{\leq} \alpha$, $y \dot{\leq} \beta$, for which no more precise linear lower bound exists than $\beta x + \alpha y - \alpha\beta$).

To account for (i), we introduce a rule that splits over the sign of the value of a term. We apply this rule for variables x that occur in the left-hand side of equations or inequalities:

$$\frac{\Gamma, x \dot{<} 0 \ \vdash \ \Delta \quad \Gamma, x \doteq 0 \ \vdash \ \Delta \quad \Gamma, x \dot{>} 0 \ \vdash \ \Delta}{\Gamma \ \vdash \ \Delta} \ \textsc{sign-cases}$$

Ternary splits are motivated by the observation that the case $x \doteq 0$ usually is easy to handle (significantly easier than the original problem), while at the same time a strict inequality $x \dot{>} 0$ appears to be of much greater use in cross-multiplication than $x \dot{\geq} 0$ (and correspondingly for $x \dot{<} 0$). In our experience, the rule SIGN-CASES outperforms binary cuts.

Point (ii) is accommodated by using the rule STRENGTHEN from Sect. 4, which we apply to linear inequalities in order to incrementally restrict the domain of a variable. For the example above, after strengthening the inequality $x \dot{\leq} \alpha$ to $x \dot{\leq} \alpha - 1$, we can also derive a better bound $\beta x + (\alpha - 1)y - \alpha\beta + \beta$ for the value of xy.

Procedure 14. *Apply Procedure 12 (equations handling and the incomplete procedure for linear inequalities) with the highest priority, the rule* SPLIT-EQ *with second highest priority, and the rules* CROSS-MULT, SIGN-CASES *and* STRENGTHEN *with the lowest priority and in a fair manner.*

Example 15. We give three further examples that can be proven using Procedure 14 (the last two ones are taken from [15, 20]). In practice, it can often be observed that Procedure 14 is able to solve nonlinear equational problems that cannot be proven using Procedure 12 (only using Gröbner bases).

$$xy \doteq 0 \vdash x \doteq 0,\ y \doteq 0 \qquad x^2 \doteq 2 \vdash \qquad 0 \dot{<} ab,\ 0 \dot{<} cd,\ 0 \dot{<} ac \vdash 0 \dot{<} bd$$

Example 16. A valid sequent that is not provable due to the restriction on the application of CROSS-MULT is $ac \dot{\leq} bd - 1,\ de \dot{\leq} a,\ c \dot{\geq} 1,\ ce \doteq b \vdash$. The problem can be solved by cross-multiplying $de \dot{\leq} a$ and $c \dot{\geq} 1$.

Proof on page 169

Lemma 17. *When applied to an invalid sequent (containing arbitrary equations and inequalities), Procedure 14 will eventually produce a counterexample.*

7 Related Work

Most similar to our approach is the arithmetic handling in ACL2 [14, 15], which also employs Fourier-Motzkin for linear and cross-multiplication for nonlinear arithmetic. Concerning differences, ACL2 runs arithmetic handling as a purely automated procedure, supports also rationals, does not have separate procedures for equations and does not seem to perform a systematic case analysis.

An method for handling linear equations and inequalities similar to our approach (but lacking counterexample generation) is described in [18] and implemented in the Tecton tool. Related is also [21] about the extension of linear reasoning to nonlinear reasoning.

Higher-order proof assistants usually support integer arithmetic and are so general that arbitrary procedures can be implemented on top of them, often targeting mathematical proofs. In comparison, we tried to develop a simple calculus/procedure specifically for Java verification that works “out of the box” and requires little expertise. The PVS proof assistant [11] can handle linear integer arithmetic and can simplify nonlinear expressions (involving multiplication and division) to some degree, but does (apparently) not go as far as our approach or ACL2. The Coq system [12] implements an incomplete version of the Omega method for deciding Presburger arithmetic (linear integer arithmetic with quantifiers) that essentially boils down to Fourier-Motzkin. Coq can also simplify ring expressions like polynomials. For HOL light [22], a number of tactics and decision procedures for arithmetic have been implemented, including Cooper’s method for deciding Presburger arithmetic, handling of congruences and simplification of polynomial expressions. Similarly, a variety of procedures for linear integer arithmetic are available in Isabelle/HOL [13, 23], as well as procedures for rings that can be instantiated to the integers. Cooper's method and the Omega test for deciding Presburger arithmetic have also been implemented for the HOL system [24].

Linear arithmetic is one of the most important theories supported by SMT solvers (which generally provide incomparably better performance for linear arithmetic than our implementation based on a general theorem prover framework), see [25] for a list. To the best of our knowledge, no SMT solver offers support for nonlinear arithmetic similar to our approach or ACL2. SMT solvers typically use linear programming techniques like Simplex, combined with methods like branch-and-bound or Gomory's cutting planes to realise completeness on the integers.

8 Conclusions and Future Work

We have presented the main components of a proof procedure for linear and nonlinear integer arithmetic, represented as sequent calculus rules together with application strategies. The procedure is completely implemented, and the soundness of the implementation is verified in the prover KeY itself. In addition to the calculus shown here, KeY also supports division and modulo operations and provides further methods like polynomial division. Based on this, we have formalised the Java semantics of integer operations.

For the future, we are considering a more efficient stand-alone implementation of the calculus, possibly based on the DPLL(T) framework. As a more conceptual extension, we plan to combine the calculus with free-variable reasoning for handling quantifiers. The general approach for this is described in [26], but needs to be investigated more carefully. Finally, we would like to add support for bit-wise operations (as they can be found in Java).

Acknowledgements

I want to thank Wolfgang Ahrendt and Richard Bubel for many inspiring discussions and comments on this paper. Thanks are also due to the anonymous referees for helpful comments.

References

1. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2007)
2. Mostowski, W.: Fully verified Java Card API reference implementation. In Beckert, B., ed.: Proceedings, Fourth International Verification Workshop, Bremen, Germany. Volume 259 of CEUR (`http://ceur-ws.org/`) (2007)

3. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
4. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
5. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms. Addison-Wesley (1997) Third edition.
6. Buchberger, B.: An algorithmical criterion for the solvability of algebraic systems. Aequationes Mathematicae **4** (1970) 374–383 (German).
7. Buchberger, B.: A critical-pair/completion algorithm for finitely generated ideals in rings. In: Proceedings, Symposium "Rekursive Kombinatorik" on Logic and Machines: Decision Problems and Complexity. LNCS, Springer (1984) 137–161
8. Dershowitz, N.: Termination of rewriting. J. Symb. Comput. **3** (1987) 69–116
9. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Communications of the ACM **22** (1979) 465–476
10. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
11. 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, 8th International Conference on Computer-Aided Verification, New Brunswick, USA. Volume 1102 of LNCS, Springer (1996) 411–414
12. 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.
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
14. Kaufmann, M., Moore, J.S.: ACL2: An industrial strength version of Nqthm. In: Proceedings, 11th Annual Conference on Computer Assurance, Gaithersburg, Maryland, National Institute of Standards and Technology (1996)
15. Warren A. Hunt, J., Krug, R.B., Moore, J.S.: Linear and nonlinear arithmetic in ACL2. In: Proceedings, Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference. Volume 2860 of LNCS, Springer (2003) 319–333
16. Ganesh, V.: Decision procedures for bit-vectors, arrays and integers. PhD thesis, Stanford, CA, USA (2007) Adviser: David L. Dill.
17. Breunesse, C.B., Jacobs, B., van den Berg, J.: Specifying and verifying a decimal representation in Java for smart cards. In: Proceedings, 9th International Conference on Algebraic Methodology and Software Technology. Volume 2422 of LNCS, Springer (2002) 304–318
18. Kapur, D., Nie, X.: Reasoning about numbers in Tecton. In: Proceedings, International Symposium on Methodologies for Intelligent Systems, Charlotte, North Carolina. (1994)
19. Matijasevic, Y.: Enumerable sets are diophantine (Russian). Dokl. Akad. Nauk SSSR **191** (1970) 279–282 Translation in Soviet Math Doklady, Vol 11, 1970.
20. Warren A. Hunt, J., Krug, R.B., Moore, J.S.: Integrating nonlinear arithmetic into ACL2. In: Proceedings, 5th International Workshop on the ACL2 Theorem Prover and Its Applications. (2004)
21. Kapur, D., Cyrluk, D.: Reasoning about nonlinear inequality constraints: a multi-level approach. In: Proceedings, Image understanding workshop, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1989) 904–915
22. Harrison, J.: The HOL light manual (1.1) (2000)

23. Chaieb, A., Nipkow, T.: Verifying and reflecting quantifier elimination for presburger arithmetic. 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) 367–380
24. Norrish, M.: Complete integer decision procedures as derived rules in HOL. In: Proceedings, Theorem Proving in Higher Order Logics. Volume 2758 of LNCS, Springer (2003) 71–86
25. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smt-lib.org` (2008)
26. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In Gurevich, Y., Meyer, B., eds.: Proceedings, First International Conference on Tests and Proofs, Zurich, Switzerland. Volume 4454 of LNCS, Springer (2007) 41–60
27. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM **8** (1992) 102–114

A Proof Outlines

Lemma 3 (Properties of Gaussian Elimination)

Termination. The termination of SIMP and RED is immediate. We call the left-hand sides x of equations $x \doteq s$ (x a variable) in the antecedent "defined variables," and all other variables "independent variables." When applying RED exhaustively, each defined variable will eventually occur in exactly one place in the sequent (namely, in the defining equation).

For proving termination when COL-RED is added, we show that the leading coefficients $\alpha > 1$ of equations $\alpha x \doteq s$ constantly get smaller. We introduce a well-founded ordering on the set of multisets over $\mathbb{N} \cup \{\infty\}$ by lexicographic comparison: for $a_1 \leq \cdots \leq a_n$, $b_1 \leq \cdots \leq b_m$, we define:

$$\{\!\{a_1, \ldots, a_n\}\!\} <_m \{\!\{b_1, \ldots, b_m\}\!\} \quad \text{iff} \\ n < m \text{ or } (n = m \text{ and } (a_1, \ldots, a_n) <_{\text{lex}} (b_1, \ldots, b_m))$$

For a sequent and an independent variable x, we then consider the divisors $\gcd(\alpha_1, \ldots, \alpha_n) \in \mathbb{N} \cup \{\infty\}$, where $\alpha_1, \ldots, \alpha_n$ are all coefficients of equations $\alpha_i x \doteq s_i$ in the antecedent (we define $\gcd() = \infty$). The multiset of such gcds for all independent variables gets $<_m$-smaller for each application of COL-RED, and it gets $<_m$-smaller or stays the same when RED is applied (each time potentially followed by an application of SIMP). This proves termination.

Completeness and proof confluence. Assume that no further rules can be applied, but the proof branch at hand is not closed. This implies that the coefficient of the left-hand side of all equations is 1 (otherwise, SIMP or COL-RED can be applied), and that no left-hand side term occurs in two places in the sequent (otherwise, RED can be applied). Due to the fact that 0 is the only polynomial whose value is constantly 0 (and correspondingly for tuples of polynomials), there is a countermodel for the equations in the succedent (a valuation of the independent variables). We extend this valuation on the defined variables according to the equations in the antecedent. When investigating RED and COL-RED, it can be seen that this countermodel also is a countermodel of the original sequent.

Lemma 6 (Termination of Fourier-Motzkin Elimination)

To see that the application of FM-ELIM terminates, consider the multiset of pairs of inequalities in the antecedent to which FM-ELIM can but has not yet been applied. Pairs of inequalities can be compared lexicographically using $<_r$, and multisets of pairs can be compared using the multiset extension of this ordering. As the multiset gets smaller in this well-founded ordering each time FM-ELIM is applied, termination is guaranteed.

The rule ANTI-SYMM can introduce new equations. Such a new equation does not contain any left-hand sides of existing equations in the antecedent (because

RED has been applied exhaustively), and it therefore reduces the number of independent variables by one. In the last case, Fourier-Motzkin basically has to start over once Procedure 2 has done its job, but this can only happen a finite number of times.

Lemma 8 (Properties of Inequality Procedure)

Completeness. We show completeness referring to the main theorem of the Omega test [27], which is the following (we use the same notation as [24]):

Theorem 18 (Pugh, 1992). *Suppose $L(x) = \bigwedge_i a_i \leq \alpha_i x$ is a conjunction of lower bounds and $U(x) = \bigwedge_j \beta_j x \leq b_i$ is a conjunction of upper bounds, in which all α_i and β_j are positive integers and a_i, b_j are arbitrary terms that do not contain x. Then:*

$$\begin{array}{rcc} \exists x. L(x) \wedge U(x) & \iff & \bigwedge_{i,j} (\alpha_i - 1)(\beta_j - 1) \leq \alpha_i b_j - a_i \beta_j \\ & & \vee \\ & & \bigvee_i \bigvee_{k=0}^{m_i} \exists x. \big(\alpha_i x = a_i + k \wedge L(x) \wedge U(x)\big) \end{array}$$

The constants m and m_i are defined as follows (in case there are no upper bounds, we define $m = m_i = -1$):

$$m = \max_j \beta_j, \qquad m_i = \left\lfloor \frac{m\alpha_i - \alpha_i - m}{m} \right\rfloor$$

We also introduce a somewhat modified version of this theorem:

Lemma 19.

$$\begin{array}{rcc} \exists x. L(x) \wedge U(x) & \iff & \bigwedge_{i,j} (m_i + 1)\beta_j \leq \alpha_i b_j - a_i \beta_j \\ & & \vee \\ & & \bigvee_i \bigvee_{k=0}^{m_i} \exists x. \big(\alpha_i x = a_i + k \wedge L(x) \wedge U(x)\big) \end{array}$$

Proof. Theorem 18 has the form $E \Leftrightarrow D_1 \vee D_2$, Lem. 19 the form $E \Leftrightarrow D_1' \vee D_2$. To prove the latter equivalence, we show the two implications $D_1' \Rightarrow D_1$ and $E \wedge \neg D_1' \Rightarrow D_2$.

– $D_1' \Rightarrow D_1$: For all i and j the following inequality holds:

$$\begin{array}{rcl} (m_i + 1)\beta_j & = & \left\lfloor \frac{m\alpha_i - \alpha_i - m}{m} \right\rfloor \beta_j + \beta_j \\ & \geq & \left\lfloor \frac{\beta_j\alpha_i - \alpha_i - \beta_j}{\beta_j} \right\rfloor \beta_j + \beta_j \\ & \geq & (\beta_j\alpha_i - \alpha_i - \beta_j) - \beta_j + 1 + \beta_j \\ & = & (\alpha_i - 1)(\beta_j - 1) \end{array}$$

– $E \wedge \neg D_1' \Rightarrow D_2$: Assume $L(x_0) \wedge U(x_0)$ and that D_1' is violated because of:

$$(m_{i_0} + 1)\beta_{j_0} \;>\; \alpha_{i_0} b_{j_0} - a_{i_0}\beta_{j_0}$$

From $U(x_0)$ we can conclude $\beta_{j_0} x_0 \leq b_{j_0}$, i.e., $\alpha_{i_0}\beta_{j_0} x_0 \leq \alpha_{i_0} b_{j_0}$, and therefore:

$$\begin{array}{rrcl} & \alpha_{i_0}\beta_{j_0}x_0 \;\leq\; \alpha_{i_0}b_{j_0} & < & (m_{i_0}+1)\beta_{j_0} + a_{i_0}\beta_{j_0} \\ \Longrightarrow & \beta_{j_0}(\alpha_{i_0}x_0 - a_{i_0}) & \leq & (m_{i_0}+1)\beta_{j_0} - 1 \\ \Longrightarrow & \alpha_{i_0}x_0 - a_{i_0} & \leq & m_{i_0} + 1 - \beta_{j_0}^{-1} \\ \Longrightarrow & \alpha_{i_0}x_0 - a_{i_0} & \leq & m_{i_0} \end{array}$$

As we also know $a_{i_0} \leq \alpha_{i_0} x_0$ because of $L(x_0)$, one of the disjuncts of D_2 (for $i = i_0$ and $k = \alpha_{i_0}x_0 - a_{i_0}$) has to be true.

We prove the completeness of Procedure 7 by contradiction: assume that the procedure has been applied to a valid formula ϕ (involving equations and inequalities), and it did not construct a closed proof. If this proof attempt is finite, then none of its goals contains inequalities (because otherwise the rule STRENGTHEN would be applicable). By Lem. 3, this means that a countermodel exists for at least one goal, and then ϕ is not valid either because all involved rules are equivalence transformations (contradiction).

We can, thus, assume that the proof attempt has an infinite branch. All but finitely many steps on this branch have to be done using the rules FM-ELIM, STRENGTHEN, and SIMP: Procedure 5 terminates, the rule SPLIT-EQ can only be applied finitely often because each application eliminates one equations in the succedent, and each time ANTI-SYMM is applied the number of independent constants is decreased by one. By the same argument, when STRENGTHEN is applied on the branch, then in all but finitely many cases the branch passes through the left premiss of the rule (i.e., $s \dot{\leq} t$ is strengthened to $s \dot{<} t \Leftrightarrow s \dot{\leq} t - 1$). From now on, we only consider the infinite part of the branch that consists of applications of FM-ELIM, STRENGTHEN, and SIMP, and where always the left premiss of STRENGTHEN is considered. Again, because all involved rules are equivalence transformations and because left-hand sides of equations do not occur in inequalities, the conjunction of inequalities in each antecedent on the branch is unsatisfiable.

Pick an arbitrary sequent on the branch:

$$\Gamma, \{\alpha_i x \dot{\geq} a_i\}_i, \{\beta_j x \dot{\leq} b_i\}_j \;\vdash$$

where Γ contains equations and further inequalities, and where x is the $<_r$-largest left-hand side variable x of all inequalities. Lem. 19 tells how a proof for this sequent can be constructed: we apply STRENGTHEN repeatedly so that each inequality $\alpha_i x \dot{\geq} a_i$ is strengthened to $\alpha_i x \dot{\geq} a_i + m_i + 1$. Subsequent applications of FM-ELIM generates the same inequalities as in the right-hand side of Lem. 19:

$$\{(m_i + 1)\beta_j \leq \alpha_i b_j - a_i\beta_j\}_{i,j}$$

The same process can be repeated for the second-$<_r$-largest left-hand side variable, etc., which eventually has to yield a contradiction due to the equivalence stated in Lem. 19.

The same contradiction must have occurred on the infinite branch at hand: once all Fourier-Motzkin inferences have been generated by FM-ELIM, the effect of STRENGTHEN (followed by further applications of FM-ELIM and SIMP) is that one or multiple inequalities $s \dot{\leq} t - \alpha$ (resp., $s \dot{\geq} t + \alpha$) are turned into $s \dot{\leq} t - (\alpha + \beta)$ (resp., $s \dot{\geq} t + (\alpha + \beta)$) for some $\beta > 0$. Because STRENGTHEN is applied in a fair manner, this implies that whenever an inequality $s \dot{\leq} t - \alpha$ (resp., $s \dot{\geq} t + \alpha$) occurs somewhere on the branch, then for each $\beta > 0$ there is a $\gamma \geq 0$ such that the inequality $s \dot{\leq} t - (\alpha + \beta + \gamma)$ (resp., $s \dot{\geq} t + (\alpha + \beta + \gamma)$) occurs on the branch (upper and lower bounds of terms are strengthened infinitely often).

Counterexamples. The rule STRENGTHEN leads to a systematic enumeration of all solutions of inequalities in the antecedent.

Lemma 13 (Termination of Gröbner Basis Computation)

Procedures 2 and 5 terminate by previous lemmas, and the rules PSEUDO-RED and S-POLY form the standard Buchberger algorithm that itself is also known to terminate. Buchberger's algorithm and Procedure 5 can have an influence on procedures with a higher priority by introducing new equations. If Buchberger's algorithm or Procedure 5 produce a new linear equation, the same reasoning as in the proof of Lem. 6 applies. Likewise, if Procedure 5 (the rule ANTI-SYMM) produces a new nonlinear equation, Buchberger's algorithm has to recompute the Gröbner basis. Such an equation has to be outside of the ideal generated by the existing equations (because the rules RED and PSEUDO-RED can also be applied to inequalities). Because rational polynomials over a finite number of variables form a Noetherian ring, this can only happen finitely often.

Lemma 17 (Counterexample Generation)

All parts of the procedure that have a higher priority than then rules SIGN-CASES and STRENGTHEN terminate. Applied in a fair manner, SIGN-CASES and STRENGTHEN enumerate all possible valuations of the variables that a problem contains. If the sequent is invalid, this eventually has to find a countermodel.

Paper 7

A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic

Philipp Rümmer

Accepted at the 15th International Conference on Logic for Programming,
Artificial Intelligence and Reasoning (LPAR)
Doha, Qatar, 2008
To appear in LNCS, Springer

This thesis contains an extended version of the paper.

A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic

Philipp Rümmer

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

Abstract. First-order logic modulo the theory of integer arithmetic is the basis for reasoning in many areas, including deductive software verification and software model checking. While satisfiability checking for ground formulae in this logic is well understood, it is still an open question how the general case of quantified formulae can be handled in an efficient and systematic way. As a possible answer, we introduce a sequent calculus that combines ideas from free-variable constraint tableaux with the Omega quantifier elimination procedure. The calculus is complete for theorems of first-order logic (without functions, but with arbitrary uninterpreted predicates), can decide Presburger arithmetic, and is complete for a substantial fragment of the combination of both.

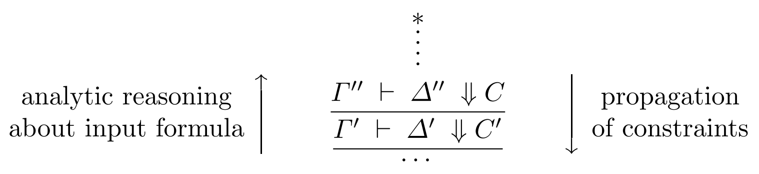
1 Introduction

One of the main challenges in automated theorem proving is to combine reasoning about full first-order logic (FOL), including quantifiers, with reasoning about theories like the integers. At the time, there are efficient provers for handling formulae in first-order logic, as well as SMT-solvers that can efficiently handle ground problems modulo many theories, but the support for the combination of both is typically weak. In this paper, we develop a novel calculus for reasoning about first-order logic modulo linear integer arithmetic that is complete for both the first-order part and the theory part, and that can handle a substantial fragment of the combination of both. Because the calculus is close to the DPLL(T) architecture, techniques and optimisations used in SMT-solvers are readily applicable when working on ground problems, but can be combined with free-variable techniques to treat quantifiers more systematically.

We start from two existing approaches: free-variable tableaux with incremental closure, following the work by Martin Giese [1], and the Omega quantifier elimination procedure [2] for deciding Presburger arithmetic (PA) [3]. From the former method, our calculus inherits the concept of generating *constraints* that describe valuations of free variables for which a formula is satisfied. The latter method provides the basic rules for dealing with linear integer arithmetic, and the concept of recursive application of a calculus in order to handle nested and alternating quantifiers. The resulting calculus accepts arbitrary formulae of PA enriched with arbitrary uninterpreted predicates as input. Uninterpreted functions are not directly supported, but can be treated by a translation to uninterpreted predicates and functionality and totality axioms.

Our calculus operates on *constrained sequents* $\Gamma \vdash \Delta \Downarrow C$, which consist of two sets Γ, Δ of formulae (the antecedent and the succedent) and one further formula C (the constraint). In this paper, C will always be a formula of PA. The semantics of a constrained sequent is the same as of the implication $C \Rightarrow (\Gamma \vdash \Delta)$, i.e., we call the sequent valid if the constraint C implies the ordinary sequent $\Gamma \vdash \Delta$ (and the ordinary sequent holds iff the formula $\bigwedge \Gamma \to \bigvee \Delta$ holds). In this sense, we can say that the constraint C is an approximation of the sequent $\Gamma \vdash \Delta$. The sequent $\forall x.(x \dot{\geq} 0 \to p(x)) \vdash p(c) \Downarrow c \dot{\geq} 0$ is valid, for instance, as are the sequents $\forall x.(x \dot{\geq} 0 \to p(x)) \vdash p(c) \Downarrow c \doteq 3$ and $\Gamma \vdash \Delta \Downarrow \mathit{false}$.

In practice, the constraints of sequents will be unknown during the construction of a proof. Reasoning about constrained sequents thus consists of two or more phases: starting with a problem $\Gamma \vdash \Delta \Downarrow ?$ with unknown constraint, a proof procedure will first apply analytic rules to the antecedent and succedent and build a proof tree, similarly as in a normal Gentzen-style sequent calculus. At some point when it seems appropriate, the procedure will start to close branches by synthesising sufficient constraints, which are subsequently propagated downwards from the leaves to the root of the tree. If the constraint that reaches the root is found to be valid, the validity of the input problem $\Gamma \vdash \Delta$ has been shown; otherwise, the procedure will continue to expand the proof tree and later update the resulting constraints.


analytic reasoning about input formula
$*$
$\Gamma'' \vdash \Delta'' \Downarrow C$
$\Gamma' \vdash \Delta' \Downarrow C'$
$\cdots$
propagation of constraints

If the input problem $\Gamma \vdash \Delta$ does not contain uninterpreted predicates (i.e., corresponds to a PA formula), it is always possible to find proofs such that the resulting constraint is equivalent to $\Gamma \vdash \Delta$ (we will call such proofs *exhaustive*). This allows us to use the calculus as a quantifier elimination procedure for PA.

Our main contributions are: the introduction of the calculus, completeness results for a number of fragments (including FOL and PA), a complete and terminating proof strategy for the PA fragment, and the result that fair proof construction is complete for formulae that are provable at all. We describe two important refinements of the calculus.

The paper is organised as follows: After giving basic definitions in Sect. 2, we introduce our calculus in three steps: Sect. 3 gives a version for pure first-order logic, Sect. 4 a minimalist version for first-order logic modulo integer arithmetic, together with completeness results, and Sect. 5 an equivalent but more refined calculus. Sect. 6 contains the result that fair proof strategies are complete. Two optimisations for the calculus are described in Sect. 7 and 8. Information about the prototypical implementation of the calculus and initial experimental results are given in Sect. 9. Finally, Sect. 10 summarises related work and Sect. 11 concludes.

2 Preliminaries

We assume that the reader is familiar with classical first-order logic and Gentzen-style sequent calculi, see [4] for an introduction. Assuming that $x \in X$ ranges over an infinite set of variables, $c \in A$ over an infinite set of constants, $p \in P$ over a set of uninterpreted predicates with fixed arity, and $\alpha \in \mathbb{Z}$ over integers, the syntactic categories of terms t and formulae ϕ are defined by:

$$\begin{aligned} t &::= \alpha \;\big|\; x \;\big|\; c \;\big|\; \alpha t + \cdots + \alpha t \\ \phi &::= \phi \wedge \phi \;\big|\; \phi \vee \phi \;\big|\; \neg\phi \;\big|\; \forall x.\phi \;\big|\; \exists x.\phi \;\big|\; t \doteq 0 \;\big|\; t \mathrel{\dot{\geq}} 0 \;\big|\; t \mathrel{\dot{\leq}} 0 \;\big|\; \alpha \mid t \;\big|\; p(t, \ldots, t) \end{aligned}$$

For reasons of simplicity, we only allow 0 as right-hand side of equations and inequalities, although we deviate from this convention in some places for sake of clarity. The explicit divisibility operator $\alpha \mid t$ is added for presentation purposes only and does not add any expressiveness (divisibility can also be expressed with an existentially quantified equation). Further, we use the abbreviations *true*, *false* for the equations $0 \doteq 0$, $1 \doteq 0$ and $\phi \rightarrow \psi$ as abbreviation for $\neg\phi \vee \psi$.

Simultaneous substitution of terms $t_1, \ldots, t_n$ for variables $x_1, \ldots, x_n$ is denoted by $[x_1/t_1, \ldots, x_n/t_n]\phi$, whereby we assume that variable capture is avoided by renaming bound variables when necessary. As short-hand notations, we sometimes also substitute terms for constants (as in $[c/t]\phi$), quantify over constants (as in $\forall c.\phi$), or quantify over sets of constants (as in $\forall U.\phi$).

Semantics. The only universe considered for evaluation are the integers $\mathbb{Z}$ (an exception is Sect. 3, where we treat normal first-order logic). A variable assignment $\beta : X \rightarrow \mathbb{Z}$ is a mapping from variables to integers, a constant assignment $\delta : A \rightarrow \mathbb{Z}$ a mapping from constants to integers, and an interpretation $I : P \rightarrow \mathcal{P}(\mathbb{Z}^*)$ a mapping from predicates to sets of $\mathbb{Z}$-tuples. The evaluation function $val_{I,\beta,\delta}$ for terms and formulae is then defined as is common and gives the arithmetic operations their normal meaning, for instance:

$$\begin{aligned} val_{I,\beta,\delta}(\alpha_1 t_1 + \cdots \alpha_n t_n) &= \sum_{i=1}^{n} \alpha_i \cdot val_{I,\beta,\delta}(t_i) \\ val_{I,\beta,\delta}(t \doteq 0) &= tt \quad \text{iff } val_{I,\beta,\delta}(t) = 0 \\ val_{I,\beta,\delta}(\alpha \mid t) &= tt \quad \text{iff there is } a \in \mathbb{Z} \text{ with } \alpha \cdot a = val_{I,\beta,\delta}(t) \\ val_{I,\beta,\delta}(p(t_1, \ldots, t_n)) &= tt \quad \text{iff } (val_{I,\beta,\delta}(t_1), \ldots, val_{I,\beta,\delta}(t_n)) \in I(p) \end{aligned}$$

We call a formula ϕ valid if $val_{I,\beta,\delta}(\phi)$ is true for all I, β, δ.

Sequents. If Γ, Δ are finite sets of formulae and C is a formula, all of which do not contain free variables, then $\Gamma \vdash \Delta$ is an (ordinary) sequent and $\Gamma \vdash \Delta \Downarrow C$ is a (constrained) sequent. We sometimes identify sequents with the formulae $\bigwedge \Gamma \to \bigvee \Delta$ (resp., $\bigwedge \Gamma \wedge C \to \bigvee \Delta$). A calculus rule is a binary relation between finite sets of constrained sequents (the premisses) and constrained sequents (the conclusion). A sequent calculus rule is called sound, iff, for all instances

$$\frac{\Gamma_1 \vdash \Delta_1 \Downarrow C_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n \Downarrow C_n}{\Gamma \vdash \Delta \Downarrow C}$$

it holds that: if all premisses $\Gamma_1 \vdash \Delta_1 \Downarrow C_1, \ldots, \Gamma_n \vdash \Delta_n \Downarrow C_n$ are valid, then $\Gamma \vdash \Delta \Downarrow C$ is valid. Proof trees are defined as is common as trees growing upwards in which each node is labelled with a constrained sequent, and in which each node that is not a leaf is related with the nodes directly above through an instance of a calculus rule. A proof is closed if it is finite, and if all leaves are justified by a rule instance without premisses.

Simplification. We denote elementary simplification steps on terms and atomic formulae in a proof with SIMP, without showing more details about the applied transformation (in an implementation, SIMP might be a part of the datastructures for formulae). SIMP normalises terms to the form $\alpha_1 t_1 + \cdots + \alpha_n t_n$, in which $\alpha_1, \ldots, \alpha_n$ are non-zero integers and $t_1, \ldots, t_n$ are pairwise distinct variables, constants, or 1 (possibly 0 as the empty sum). Further, terms are put into a canonical form by sorting summands according to a well-founded ordering $<_r$:

– on variables, constants and integers, $<_r$ is an arbitrary well-ordering such that variables are bigger than constants, constants are bigger than integers, and: $0 <_r 1 <_r -1 <_r 2 <_r -2 <_r 3 <_r \cdots$.
– on terms with coefficients, $<_r$ is defined by $\alpha t <_r \alpha' t'$ if and only if $t <_r t'$ or $t = t'$ and $\alpha <_r \alpha'$.
– on linear combinations, $\alpha_1 t_1 + \cdots + \alpha_n t_n <_r \alpha'_1 t'_1 + \cdots + \alpha'_k t'_k$ holds if and only if $\{\!\{\alpha_1 t_1, \ldots, \alpha_n t_n\}\!\} <_r \{\!\{\alpha'_1 t'_1, \ldots, \alpha'_n t'_n\}\!\}$ in the multiset extension of $<_r$ [5].
– for arbitrary terms t, t', let $\alpha_1 t_1 + \cdots + \alpha_n t_n$ and $\alpha'_1 t'_1 + \cdots + \alpha'_k t'_k$ denote equivalent linear combinations (as above, $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_k$ are pairwise distinct variables, constants, or 1). The relationship $t <_r t'$ holds if and only if $\alpha_1 t_1 + \cdots + \alpha_n t_n <_r \alpha'_1 t'_1 + \cdots + \alpha'_k t'_k$.

Atomic formulae $t \doteq 0$, $t \dot{\geq} 0$, $t \dot{\leq} 0$ are normalised by SIMP such that the coefficients of non-constant terms in t are coprime (do not have non-trivial factors in common), and such that the leading coefficient is non-negative. This also detects that equations like $2y - 6c + 1 \doteq 0$ are unsolvable and equivalent to *false*, and that an inequality like $2y - 6c + 1 \dot{\leq} 0$ can be simplified and rounded to $y - 3c + 1 \dot{\leq} 0$ thanks to the discreteness of the integers. All inequalities in the succedent are moved to the antecedent. A divisibility judgement $\alpha \mid t$ is normalised like an equation $\alpha x + t \doteq 0$, and it is ensured that α and the leading coefficient of t are positive.

$$\frac{\Gamma \vdash \phi, \Delta \Downarrow C \qquad \Gamma \vdash \psi, \Delta \Downarrow D}{\Gamma \vdash \phi \wedge \psi, \Delta \Downarrow C \wedge D} \text{ AND-RIGHT}$$

$$\frac{\Gamma, \phi \vdash \Delta \Downarrow C \qquad \Gamma, \psi \vdash \Delta \Downarrow D}{\Gamma, \phi \vee \psi \vdash \Delta \Downarrow C \wedge D} \text{ OR-LEFT}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta \Downarrow C}{\Gamma, \phi \wedge \psi \vdash \Delta \Downarrow C} \text{ AND-LEFT} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta \Downarrow C}{\Gamma \vdash \phi \vee \psi, \Delta \Downarrow C} \text{ OR-RIGHT}$$

$$\frac{\Gamma \vdash \phi, \Delta \Downarrow C}{\Gamma, \neg\phi \vdash \Delta \Downarrow C} \text{ NOT-LEFT} \qquad \frac{\Gamma, \phi \vdash \Delta \Downarrow C}{\Gamma \vdash \neg\phi, \Delta \Downarrow C} \text{ NOT-RIGHT}$$

$$\frac{\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.C} \text{ EX-RIGHT} \qquad \frac{\Gamma, [x/c]\phi, \forall x.\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow \exists x.C} \text{ ALL-LEFT}$$

$$\frac{\Gamma \vdash [x/c]\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \forall x.\phi, \Delta \Downarrow \forall x.C} \text{ ALL-RIGHT} \qquad \frac{\Gamma, [x/c]\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \exists x.\phi \vdash \Delta \Downarrow \forall x.C} \text{ EX-LEFT}$$

Fig. 1. The rules for first-order predicate logic (without equality). In all rules, c is a constant that does not occur in the conclusion: in contrast to the usage of Skolem functions and free variables in tableaux, the same kinds of symbols (constants) are used to handle both existential and universal quantifiers. Arbitrary renaming of bound variables is allowed in the constraints when necessary to avoid variable capture.

3 A Constraint Sequent Calculus for First-Order Logic

We first introduce a very restricted calculus for pure first-order logic, in order to illustrate how the framework of constrained sequents is related to normal free-variable tableau calculi. This section is exceptional in that we do *not* assume evaluation of formulae over the universe $\mathbb{Z}$ of integers, and that we allow equations $s \doteq t$ whose right-hand side is not 0. The rules from Fig. 1, together with the following closure rule, form the calculus Pred^C:

$$\frac{*}{\Gamma, p(s_1, \ldots, s_n) \vdash p(t_1, \ldots, t_n), \Delta \Downarrow \bigwedge_i s_i \doteq t_i} \text{ PRED-CLOSE}$$

Instead of unifying complementary literals, a conjunction of equations about the predicate arguments is generated and propagated as a constraint.

Example 1. We show a proof for the sequent $\forall x.\exists y.p(x,y) \vdash \exists z.p(a,z)$:

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{\ldots, p(c,d) \vdash \ldots, p(a,e) \Downarrow c \doteq a \wedge d \doteq e} \text{ PRED-CLOSE}}{\ldots, p(c,d) \vdash \exists z.p(a,z) \Downarrow \exists z.(c \doteq a \wedge d \doteq z)} \text{ EX-RIGHT}}{\ldots, \exists y.p(c,y) \vdash \exists z.p(a,z) \Downarrow \forall y.\exists z.(c \doteq a \wedge y \doteq z)} \text{ EX-LEFT}}{\forall x.\exists y.p(x,y) \vdash \exists z.p(a,z) \Downarrow \exists x.\forall y.\exists z.(x \doteq a \wedge y \doteq z)} \text{ ALL-LEFT}$$

In order to instantiate existential and universal quantifiers, fresh constants c, d, e are introduced. The constraints on the right-hand side are practically filled in *after* applying PRED-CLOSE. Because $\exists x.\forall y.\exists z.(x \doteq a \land y \doteq z)$ is valid, also the validity of the original problem is proven.

It is easy to see that a constraint C produced by a proof can only consist of equations over variables and constants, conjunctions, and quantifiers (because these are the only constructs that are introduced in constraints by the rules of Pred^C). The validity of constraints/formulae of this kind is decidable and corresponds to simultaneous unification, which makes the calculus effective.

Proof on page 198

Lemma 2 (Soundness). *If a sequent $\Gamma \vdash \Delta \Downarrow C$ is provable in $Pred^C$, then it is valid (holds in all first-order structures).*

Proof on page 198

Lemma 3 (Completeness). *Suppose ϕ is closed, valid (holds in all first-order structures), and does not contain constants. Then there is a valid constraint C such that $\vdash \phi \Downarrow C$ is provable in $Pred^C$.*

It can be observed that Pred^C is also proof confluent, which strengthens Lem. 3. In order to continue ("complement") a partial proof, it can be both necessary to expand branches further and to update constraints anywhere in the proof:

Proof on page 203

Lemma 4 (Proof confluence). *Suppose that ϕ is valid. Any (partial) $Pred^C$-proof with root $\vdash \phi \Downarrow ?$ that does not contain applications of* PRED-CLOSE *can be complemented to a closed proof tree with root $\vdash \phi \Downarrow C$ for some valid constraint C.*

4 Adding Integer Arithmetic

Relatively few changes to the calculus Pred^C from the previous section are necessary to reason about problems in integer arithmetic. In this section, we describe a minimalist approach in which all integer reasoning happens during the constraint solving and investigate fragments on which the resulting method is complete. Later in the paper, the calculus is refined and optimised. From now on and in contrast to the previous section, assume that formulae and terms are evaluated over first-order structures with the universe $\mathbb{Z}$ as described in Sect. 2.

In contrast to the previous section, to handle integer arithmetic disjunctive constraints also need to be considered. We thus split the rule PRED-CLOSE into two new rules, one of which (PRED-UNIFY) generates unification conditions for complementary pairs, while the other one (CLOSE) allows to synthesise a constraint from arbitrary formulae in a sequent:

$$\frac{\Gamma, p(s_1,\ldots,s_n) \vdash p(t_1,\ldots,t_n), \bigwedge_i s_i - t_i \doteq 0, \Delta \Downarrow C}{\Gamma, p(s_1,\ldots,s_n) \vdash p(t_1,\ldots,t_n), \Delta \Downarrow C}\ \textsc{pred-unify}$$

$$\frac{*}{\Gamma, \phi_1,\ldots,\phi_n \vdash \psi_1,\ldots,\psi_m, \Delta \Downarrow \neg\phi_1 \lor \cdots \lor \neg\phi_n \lor \psi_1 \lor \cdots \lor \psi_m}\ \textsc{close}$$

($\phi_1,\ldots,\phi_n,\psi_1,\ldots,\psi_m$ do not contain uninterpreted predicates)

$$
\begin{array}{cl}
* & \\ \hline
\ldots \vdash \ldots, 2d - c - 10 \doteq 0, c - 2e - 1 \doteq 0 \ \Downarrow C_1 & \textsc{close} \\ \hline
p(2d), \ldots, p(c) \vdash p(c+10), p(2e+1) \ \Downarrow C_1 & \textsc{pred-unify} \times 2 \\ \hline
\ldots, p(2d), \forall x. \neg p(2x+1), p(c) \vdash p(c+10) \ \Downarrow C_2 & \textsc{all-left}, \textsc{not-left} \\ \hline
\forall x. p(2x), \forall x. \neg p(2x+1), p(c) \vdash p(c+10) \ \Downarrow C_3 & \textsc{all-left} \\ \hline
\forall x. p(2x), \forall x. \neg p(2x+1) \vdash \neg p(c) \vee p(c+10) \ \Downarrow C_3 & \textsc{or-right}, \textsc{not-right} \\ \hline
\forall x. p(2x), \forall x. \neg p(2x+1) \vdash \forall y.(p(y) \to p(y+10)) \ \Downarrow C_4 & \textsc{all-right}
\end{array}
$$

The constraints resulting from the proof are:

$$
\begin{array}{lclcl}
C_1 & = & & & 2d - c - 10 \doteq 0 \vee c - 2e - 1 \doteq 0 \\
C_2 & = & \exists y.[e/y]C_1 & = & \exists y.(2d - c - 10 \doteq 0 \vee c - 2y - 1 \doteq 0) \\
C_3 & = & \exists x.[d/x]C_2 & = & \exists x. \exists y.(2x - c - 10 \doteq 0 \vee c - 2y - 1 \doteq 0) \\
 & & & \equiv & 2 \mid (c+10) \vee 2 \mid (c-1) \\
C_4 & = & \forall x.[c/x]C_3 & = & \forall x.(2 \mid (x+10) \vee 2 \mid (x-1)) \\
 & & & \equiv & \mathit{true}
\end{array}
$$

Fig. 2. An example proof in the calculus $\mathrm{PresPred}_S^C$.

Besides these two rules, $\mathrm{PresPred}_S^C$ contains all rules given in Fig. 1. It is obvious that any proof in Pred^C can be translated to a proof in $\mathrm{PresPred}_S^C$ by replacing applications of PRED-CLOSE with applications of PRED-UNIFY, followed by CLOSE, which means that $\mathrm{PresPred}_S^C$ is complete for first-order logic.

Because uninterpreted predicates are excluded in CLOSE, the constraint resulting from a proof is always a formula in Presburger arithmetic and can in principle be handled using any decision procedure for PA (e.g. [6, 2], also see Sect. 5.3). We come back to this issue later in the paper and assume for the time being that some procedure is available for deciding the validity of constraints.

As an implication of a more general result (Lem. 17), it can be observed that $\mathrm{PresPred}_S^C$ is proof-confluent: if ϕ is provable, then every partial proof of $\vdash \phi \Downarrow ?$ can be extended to a closed proof of a sequent $\vdash \phi \Downarrow C$ with valid constraint C.

Example 5. We show a proof for the following sequent (Fig. 2):

$$\forall x. p(2x), \forall x. \neg p(2x+1) \vdash \forall y.(p(y) \to p(y+10))$$

The sequent is proven by first building the "main proof" (upwards) to a point where CLOSE can be applied. The constraints $C_1, \ldots, C_4$ are then filled in and propagated downwards. Because C_4 is valid, we have proven the validity of the original formula. The constraint simplification is explained in more detail later.

Completeness on fragments. Two fragments on which $\mathrm{PresPred}_S^C$ is complete are the classes of purely universal and of purely existential formulae. We call positions in the antecedent/succedent of a sequent *positive* if they are underneath an odd/even number of negations. All other positions are called *negative.*

Proof on page 203

Lemma 6. *If $\Gamma \vdash \Delta$ is a valid sequent in which $\exists$ only occurs in negative and $\forall$ only in positive positions, then there is a valid PA constraint C such that $\Gamma \vdash \Delta \Downarrow C$ has a proof in the calculus PresPred^C_S.*

Proof on page 204

Lemma 7. *If $\Gamma \vdash \Delta$ is a valid sequent (without constants) in which $\exists$ only occurs in positive and $\forall$ only in negative positions, then there is a valid PA constraint C such that $\Gamma \vdash \Delta \Downarrow C$ has a proof in the calculus PresPred^C_S.*

Comparison with $\mathcal{ME}$(LIA). We can also show that the calculus PresPred^C_S is complete on the fragment of first-order logic modulo linear integer arithmetic that can be handled by Model Evolution modulo linear integer arithmetic [7]. Ignoring minor syntactic issues and the fact that $\mathcal{ME}$(LIA) works on clauses, $\mathcal{ME}$(LIA) is a sound and complete calculus for proving the unsatisfiability of formulae of the shape $\exists \bar{a}.(\phi \wedge \psi)$, where:

- $\bar{a} = (a_1, \ldots, a_m)$ is a vector of existentially quantified variables,
- ϕ is a PA formula over $\bar{a}$ that only has finitely many solutions, and
- ψ is an arbitrary formula over $\bar{a}$ in which $\exists$ only occurs in negative and $\forall$ only in positive positions.

Proof on page 205

Lemma 8. *If $\exists \bar{a}.(\phi \wedge \psi)$ as above is an unsatisfiable formula that does not contain constants or free variables, then there is a valid constraint C such that the sequent $\exists \bar{a}.(\phi \wedge \psi) \vdash \ \Downarrow C$ has a proof in PresPred^C_S.*

5 Built-In Handling of Presburger Arithmetic

Although the calculus from the previous section is in principle usable, it practically has a number of shortcomings: the handling of arithmetic in constraints provides little guidance for the construction of proofs, so that large constraints are produced in a very indeterministic manner that cannot be solved efficiently. Moreover, constraints are even needed to handle ground problems, for which branch-local reasoning should be sufficient. The main goal when refining the calculus is, therefore, to reduce the usage of constraints as far as possible.

In this section, we define built-in rules for handling linear integer arithmetic that can be interleaved with the rules from the previous section. The rules make it possible to handle ground problems branch-locally: proof trees for ground problems can be constructed depth-first (non-iteratively), similarly to the way in which SMT-solvers work. Together with the refinement in Sect. 7, it can be achieved that the only constraints that can result from a subproof in case of ground problems are *true* or *false*. Branch-local reasoning is also possible for innermost $\forall$-quantifiers in positive and $\exists$ in negative positions. The arithmetic rules also yield a decision procedure for PA that can be used to decide constraints (Sect. 5.3).

$$\frac{\Gamma \vdash [x/c]\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.C}\ \textsc{ex-right-d} \qquad \frac{\Gamma, [x/c]\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow \exists x.C}\ \textsc{all-left-d}$$

(c a constant that does not occur in the conclusion, ϕ does not contain uninterpreted predicates)

$$\frac{\Gamma, t \doteq 0 \vdash \phi[s + \alpha \cdot t], \Delta \Downarrow C}{\Gamma, t \doteq 0 \vdash \phi[s], \Delta \Downarrow C}\ \textsc{red}$$

(α a literal, or t a literal and α an arbitrary term)

$$\frac{\Gamma, \alpha(u + c') + t \doteq 0, c - u - c' \doteq 0 \vdash \Delta \Downarrow [x/c']C}{\Gamma, \alpha c + t \doteq 0 \vdash \Delta \Downarrow \forall x.C}\ \textsc{col-red}$$

(c' a constant that does not occur in the conclusion or in u)

$$\frac{\Gamma, \alpha(u + c') + t \doteq 0, c - u - c' \doteq 0 \vdash \Delta \Downarrow [x/c']C}{\Gamma, \alpha c + t \doteq 0 \vdash \Delta \Downarrow [x/c - u]C}\ \textsc{col-red-subst}$$

(c' a constant that does not occur in the conclusion or in u)

$$\frac{\Gamma, \exists x.\alpha x + t \doteq 0 \vdash \Delta \Downarrow C}{\Gamma, \alpha \mid t \vdash \Delta \Downarrow C}\ \textsc{div-left}$$

(x an arbitrary variable)

$$\frac{\Gamma, (\alpha \mid t + 1) \vee \cdots \vee (\alpha \mid t + \alpha - 1) \vdash \Delta \Downarrow C}{\Gamma \vdash \alpha \mid t, \Delta \Downarrow C}\ \textsc{div-right} \qquad (\alpha > 0)$$

$$\frac{\Gamma, \alpha c - t \doteq 0 \vdash \Delta \Downarrow C}{\Gamma, \alpha c - t \doteq 0 \vdash \Delta \Downarrow [x/t]C' \vee \alpha \nmid t}\ \textsc{div-close}$$

(c does not occur in t or in C', C' a PA formula such that $C \Leftrightarrow [x/\alpha c]C'$)

$$\frac{\Gamma \vdash t \dot{\leq} 0, \Delta \Downarrow C \quad \Gamma \vdash t \dot{\geq} 0, \Delta \Downarrow D}{\Gamma \vdash t \doteq 0, \Delta \Downarrow C \wedge D}\ \textsc{split-eq}$$

$$\frac{\Gamma, t \doteq 0 \vdash \Delta \Downarrow C}{\Gamma, t \dot{\leq} 0, t \dot{\geq} 0 \vdash \Delta \Downarrow C}\ \textsc{anti-symm}$$

$$\frac{\Gamma, \alpha c + s \dot{\geq} 0, \beta c + t \dot{\leq} 0, \beta s - \alpha t \dot{\geq} 0 \vdash \Delta \Downarrow C}{\Gamma, \alpha c + s \dot{\geq} 0, \beta c + t \dot{\leq} 0 \vdash \Delta \Downarrow C}\ \textsc{fm-elim}$$

$(\alpha > 0,\ \beta > 0)$

$$\frac{\begin{array}{lcr} & \bigwedge_{i,j} \alpha_i b_j - a_i \beta_j - (\alpha_i - 1)(\beta_j - 1) \dot{\geq} 0 & \\ \Gamma, & \vee & \vdash \Delta \Downarrow C \\ & \bigvee_i \bigvee_{k=0}^{m_i} \left(\begin{array}{c} \alpha_i c - a_i - k \doteq 0 \wedge \\ \bigwedge_i \alpha_i c - a_i \dot{\geq} 0 \wedge \bigwedge_j \beta_j c - b_j \dot{\leq} 0 \end{array} \right) & \end{array}}{\Gamma, \{\alpha_i c - a_i \dot{\geq} 0\}_i, \{\beta_j c - b_j \dot{\leq} 0\}_j \vdash \Delta \Downarrow C}\ \textsc{omega-elim}$$

$(\alpha_i > 0,\ \beta_j > 0)$

Fig. 3. Rules for equations, inequalities, and divisibility judgements. In RED, we write $\phi[s]$ in the succedent to denote that s occurs in an arbitrary formula in the sequent, which can in particular also be in the antecedent. m_i in OMEGA-ELIM as on page 182.

The rules in detail. The calculus PresPred^C consists of the rules given in Fig. 3, together with all rules from the calculus PresPred^C_S and the simplification rule SIMP. We introduce new rules EX-RIGHT-D, ALL-LEFT-D that instantiate quantified formulae destructively, because formulae that do not contain uninterpreted predicates never have to be instantiated twice (also see Lem. 17 below).

The equality handling follows the calculus given in [8] and can solve arbitrary equations in the antecedent, in the sense that the equations are rewritten until the leading coefficients are all 1 and the leading terms of equations occur in exactly one place. Speaking in terms of matrices, RED is the rule for performing row operations, while COL-RED(-SUBST) is responsible for column operations. We define a suitable strategy for guiding the rules below.

The rules DIV-RIGHT and DIV-LEFT translate divisibility statements to equations, while DIV-CLOSE synthesises divisibility statements from equations. The formula C' in DIV-CLOSE can be found through pseudo-division (multiplying equations, inequalities or divisibility statements in C with non-zero factors). For $C = (c + d \doteq 0)$ and $\alpha = 3$, for instance, we would choose $C' = (x + 3d \doteq 0)$.

Inequalities are handled based on the Omega test [2], which is an extension of the Fourier-Motzkin variable elimination method [9] for integer problems. The central rule is OMEGA-ELIM for replacing a conjunction of inequalities with a disjunction over simpler cases. The literal m_i in the rule is defined by:

$$m = \max_j \beta_j, \qquad m_i = \left\lfloor \frac{m\alpha_i - \alpha_i - m}{m} \right\rfloor$$

In case there are no upper bounds, we define $m = m_i = -1$. OMEGA-ELIM is directly based on the main theorem [2] underlying the Omega test, which is the following (we use the notation from [10] where also a proof is provided).

Theorem 9 (Pugh, 1992). *Suppose $L(x) = \bigwedge_i a_i \leq \alpha_i x$ is a conjunction of lower bounds and $U(x) = \bigwedge_j \beta_j x \leq b_i$ is a conjunction of upper bounds, in which all α_i and β_j are positive integers and a_i, b_j are arbitrary terms that do not contain x. Then:*

$$\exists x. L(x) \wedge U(x) \quad \iff \quad \begin{array}{c} \bigwedge_{i,j} (\alpha_i - 1)(\beta_j - 1) \leq \alpha_i b_j - a_i \beta_j \\ \vee \\ \bigvee_i \bigvee_{k=0}^{m_i} \exists x. \big(\alpha_i x = a_i + k \wedge L(x) \wedge U(x)\big) \end{array}$$

Appealing to a geometric interpretation, the first disjunct on the right-hand side is called the “dark shadow,” whereas the existentially quantified disjuncts are called “splinters.” In case all of the α_is or all of the β_js are 1, the equivalence boils down to the normal Fourier-Motzkin rule:

$$\exists x. L(x) \wedge U(x) \quad \iff \quad \bigwedge_{i,j} a_i \beta_j \leq \alpha_i b_j$$

The application of OMEGA-ELIM is only meaningful if c does not occur in formulae other than inequalities. Note, that if there are no lower or no upper bounds, the rule will replace all inequalities whose leading term is c with *true*.

Because we avoid the application of OMEGA-ELIM in certain common situations (for instance, whenever the constant c occurs as argument of uninterpreted predicates), we also introduce a rule FM-ELIM for normal Fourier-Motzkin elimination. FM-ELIM can be applied with higher priority than OMEGA-ELIM and is often able to close proofs faster than OMEGA-ELIM, reducing the need to resort to the more complex rule. Further, we define two rules to convert between equations and inequalities. While the rule SPLIT-EQ is strictly necessary for certain problems, ANTI-SYMM is introduced only for reasons of efficiency.

Lemma 10 (Soundness). *If a sequent $\Gamma \vdash \Delta \Downarrow C$ is provable in PresPred^C, then it is valid.*

Proof on page 209

5.1 Exhaustive Proofs

The existence of a closed proof for a sequent $\Gamma \vdash \Delta \Downarrow C$ guarantees that the implication $C \Rightarrow (\Gamma \vdash \Delta)$ holds (this is the soundness of the calculus, Lem. 10). In the special case that the sequent $\Gamma \vdash \Delta$ does not contain uninterpreted predicates, it is possible to distinguish particular closed proofs that also guarantee the opposite implication $(\Gamma \vdash \Delta) \Rightarrow C$, and thus $(\Gamma \vdash \Delta) \Leftrightarrow C$. While this can be achieved in a trivial way by always applying CLOSE such that *all* formulae in a sequent are selected, it is sufficient to impose a weaker condition on proof trees that leads to smaller constraints and also makes it possible to eliminate quantifiers (Sect. 5.3). To this end, it is necessary to remember whether a constant was introduced by an existential rule (like EX-RIGHT) or a universal rule (like ALL-RIGHT), and whether other existential rules were applied in between. A generalisation of the condition that allows to select even fewer formulae is described in Sect. 8.

Assume that a PresPred^C-proof is given. We annotate the sequents in the proof with sets U of "universal" constants that the calculus attempts to eliminate. More formally, the proof is called *exhaustive* iff there is a mapping from proof nodes (constrained sequents) to sets U of constants subject to the following conditions:

1. The rules AND-*, OR-*, NOT-*, PRED-UNIFY, RED, DIV-*, SPLIT-EQ, ANTI-SYMM, FM-ELIM, and SIMP keep or reduce the set: if the conclusion is annotated with U, the premisses are annotated with arbitrary subsets of U.
2. The rules EX-RIGHT(-D), ALL-LEFT(-D) erase the set: the premiss is annotated with $\emptyset$.
3. The rules EX-LEFT and ALL-RIGHT may add the introduced constant c to the set: if the conclusion is annotated with U, then the premiss is annotated with a subset of $U \cup \{c\}$.
4. The rule COL-RED is only applied if the conclusion is annotated with U such that $c \in U$. In this case, the premiss is annotated with a subset of $U \cup \{c'\}$.
5. The rule COL-RED-SUBST is only applied if the conclusion is annotated with U such that $c \notin U$, and if u does not contain any constants from U. In this case, the premiss is annotated with a subset of U.

6. The rule OMEGA-ELIM is only applied if the conclusion is annotated with U such that $c \in U$ and if c does not occur in Γ or Δ. In this case, the premiss is annotated with an arbitrary subset of U.
7. The rule DIV-CLOSE is only applied if the conclusion is annotated with U such that $c \in U$. In this case, the premiss is annotated with a subset of U.
8. The rule CLOSE is always applied such that all formulae without uninterpreted predicates are selected, apart from (possibly) those equations in the succedent that contain constants from U that exclusively occur in equations in the succedent.

Proof on page 210

Lemma 11 (Constraint completeness). *Suppose that a PresPred^C-proof is closed and exhaustive. For each sequent $\Gamma \vdash \Delta \Downarrow C$ in the tree that is annotated with a set U, let Γ_p, Δ_p denote the subsets of PA formulae in Γ, Δ. The following implication holds for each sequent:*

$$\forall U.\ (\Gamma_p \vdash \Delta_p) \;\Rightarrow\; \forall U.\ C \tag{1}$$

Example 12. The formula $\neg\exists x.\exists y.(2x - c - 10 \doteq 0 \vee 2y - c + 1 \doteq 0)$ from Example 5 can be simplified to $2 \nmid (c + 10) \wedge 2 \nmid (c - 1)$ by constructing an exhaustive proof (Fig. 4).

Example 13. The constraint $\forall x.(2 \mid (x + 10) \vee 2 \mid (x - 1))$ from Example 5 is simplified to *true* by constructing the following proof:

$$\begin{array}{cl}
* & \\ \hline
c + 2d' + 1 \doteq 0, d - d' + 5 \doteq 0, \mathit{false} \vdash \;\Downarrow \mathit{true} & \textsc{close} \\ \hline
c + 2d' + 1 \doteq 0, d - d' + 5 \doteq 0, \exists y.\mathit{false} \vdash \;\Downarrow \mathit{true} & \textsc{ex-left} \\ \hline
c + 2d' + 1 \doteq 0, d - d' + 5 \doteq 0, \exists y.2y - 2d' - 1 \doteq 0 \vdash \;\Downarrow \mathit{true} & \textsc{simp} \\ \hline
c + 2d' + 1 \doteq 0, d - d' + 5 \doteq 0, \exists y.2y + c \doteq 0 \vdash \;\Downarrow \mathit{true} & \textsc{red} \\ \hline
2(-5 + d') + c + 11 \doteq 0, d - (-5) - d' \doteq 0, \exists y.2y + c \doteq 0 \vdash \;\Downarrow \mathit{true} & \textsc{simp} \times 2 \\ \hline
2d + c + 11 \doteq 0, \exists y.2y + c \doteq 0 \vdash \;\Downarrow \mathit{true} & \textsc{col-red} \\ \hline
\exists x.2x + c + 11 \doteq 0, \exists y.2y + c \doteq 0 \vdash \;\Downarrow \mathit{true} & \textsc{ex-left} \\ \hline
2 \mid (c + 11), 2 \mid c \vdash \;\Downarrow \mathit{true} & \textsc{div-left} \times 2 \\ \hline
\vdash 2 \mid (c + 10), 2 \mid (c - 1) \Downarrow \mathit{true} & \textsc{div-right} \times 2, \textsc{simp} \\ \hline
\vdash \forall x.(2 \mid (x + 10) \vee 2 \mid (x - 1)) \Downarrow \mathit{true} & \textsc{all-right}, \textsc{or-right}
\end{array}$$

5.2 The Construction of Exhaustive Proofs for PA Problems

We define a strategy to apply the PresPred^C-rules to a sequent $\Gamma \vdash \Delta \Downarrow ?$ that only contains PA formulae. The strategy is guaranteed to terminate and to produce a closed and exhaustive proof, and it is deterministic in the sense that no search is required, every ordering of rule applications (that is consistent with given priorities) leads to an exhaustive proof. In order to guide the proof construction, the strategy maintains a set U of constants (which is initially empty) and a term ordering $<_r$ (as in Sect. 2) that are updated when new constants

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{2d - c - 10 \doteq 0 \ \vdash\ \Downarrow D_1}\ \textsc{close}}{2d - c - 10 \doteq 0 \ \vdash\ \Downarrow D_2}\ \textsc{div-close} \qquad \dfrac{\dfrac{*}{2e - c + 1 \doteq 0 \ \vdash\ \Downarrow D_3}\ \textsc{close}}{2e - c + 1 \doteq 0 \ \vdash\ \Downarrow D_4}\ \textsc{div-close}}{2d - c - 10 \doteq 0 \vee 2e - c + 1 \doteq 0 \ \vdash\ \Downarrow D_2 \wedge D_4}\ \textsc{or-left}}{\exists x. \exists y. (2x - c - 10 \doteq 0 \vee 2y - c + 1 \doteq 0) \ \vdash\ \Downarrow D_5}\ \textsc{ex-left} \times 2$$

The constraints resulting from the proof are:

$$\begin{array}{lcllcl}
D_1 & = & & & & 2d - c - 10 \not\doteq 0 \\
D_2 & = & [2d/c + 10]D_1 \vee 2 \nmid (c + 10) & = & & (c + 10) - c - 10 \not\doteq 0 \vee 2 \nmid (c + 10) \\
 & & & \equiv & & 2 \nmid (c + 10) \\
D_3 & = & & & & 2e - c + 1 \not\doteq 0 \\
D_4 & = & [2e/c - 1]D_3 \vee 2 \nmid (c - 1) & = & & (c - 1) - c + 1 \not\doteq 0 \vee 2 \nmid (c - 1) \\
 & & & \equiv & & 2 \nmid (c - 1) \\
D_5 & = & \exists x.[d/x]\exists y.[e/y](D_2 \wedge D_4) & = & & \exists x. \exists y. (2 \nmid (c + 10) \wedge 2 \nmid (c - 1)) \\
 & & & \equiv & & 2 \nmid (c + 10) \wedge 2 \nmid (c - 1)
\end{array}$$

Fig. 4. Simplification of the formula $\neg \exists x. \exists y. (2x - c - 10 \doteq 0 \vee 2y - c + 1 \doteq 0)$ from Example 5 by constructing a proof. To see that the proof is exhaustive, the sequent with constraint D_5 is annotated with $\emptyset$, the sequent with D_1 with $\{e\}$, the sequent with D_3 with $\{d\}$, and all other sequents with the set $\{d, e\}$. This implies that the original formula is equivalent to D_5.

are introduced or existing constants need to be reordered. The ordering $<_r$ is always chosen such that the constants in U are bigger than all constants that are not in U. Both U and $<_r$ are branch-local: different branches in a proof tree can be built using different Us and $<_r$s.

We list the rules that the strategy applies to a proof goal with descending priority: step 2 will only be carried out if step 1 is impossible, etc.

1. apply SIMP (if possible).
2. apply RED if an α exists such that $s + \alpha \cdot t <_r s$ (and if $s \neq t$ or $\phi[s]$ is not an equation in the antecedent).
3. if the antecedent contains an equation $\alpha c + t \doteq 0$ with $\alpha > 1$, then:
   - if $c \notin U$, apply COL-RED-SUBST. The fresh constant c' is inserted in the term ordering $<_r$ such that it becomes minimal, and u is chosen such that $(\alpha u + t) = \min_{<_r} \{\alpha u' + t \mid u' \text{ a term}\}$.
   - if $c \in U$ and t contains at least one further constant from U whose coefficient is not a multiple of α, apply COL-RED. The fresh constant c' is added to U and is inserted in the term ordering $<_r$ such that it becomes smaller than all other constants in U, but bigger than all constants not in U. u is again chosen such that $(\alpha u + t) = \min_{<_r} \{\alpha u' + t \mid u' \text{ a term}\}$.
4. if the antecedent contains an equation $\alpha c + t \doteq 0$ with $c \in U$, apply DIV-CLOSE, remove c from U, and update $<_r$ such that c becomes minimal. (This is also possible for $\alpha = 1$)
5. if possible, apply any of the following rules:
   - ANTI-SYMM.

   - FM-ELIM, if the result is not subsumed by an inequality in the antecedent.
   - any of the rules AND-*, OR-*, NOT-*.
6. if possible, apply any of the following rules:
   - SPLIT-EQ: if an equation can be split that contains a constant $c \in U$ that also occurs as leading term of an inequality in the antecedent.
   - OMEGA-ELIM: if inequalities $\{\alpha_i c - a_i \dot{\geq} 0\}_i$, $\{\beta_j c - b_j \dot{\leq} 0\}_j$ occur in the antecedent and $c \in U$, and if c does not occur in any other formula.
   - ALL-RIGHT, EX-LEFT: add the fresh constant c to U and insert it into $<_r$ such that it becomes maximal.
   - EX-RIGHT-D, ALL-LEFT-D: set U to $\emptyset$ and insert c arbitrarily into $<_r$.
   - DIV-LEFT, DIV-RIGHT.
7. apply CLOSE and select exactly those formulae that do not contain constants from U or uninterpreted predicates.

The steps 1–4 of the strategy work by eliminating all U-constants that occur in equations in the antecedent. Similarly as in [8], in the antecedent only equations will be left whose leading coefficient is 1 and whose leading term does not occur in other places in the sequent anymore. The steps 5–6 handle inequalities by first applying the Fourier-Motzkin rule exhaustively, and by eliminating constants using the Omega rule whenever possible. Also quantifiers, propositional connectives and divisibility judgements are treated in step 5–6. A proof that is constructed using this procedure is shown in Example 12.

Proof on page 211

Lemma 14 (Termination and exhaustiveness). *If a sequent $\Gamma \vdash \Delta \Downarrow ?$ does not contain uninterpreted predicates, the strategy from above terminates and produces a closed exhaustive proof.*

5.3 Deciding Presburger Arithmetic by Recursive Proving

The anticipated way to decide constraints in proofs is to eliminate quantifiers already during the constraint propagation, i.e., at the points where the rules EX-RIGHT(-D), ALL-LEFT(-D), ALL-RIGHT, EX-LEFT or COL-RED are applied and cause quantifiers to occur in constraints. By eliminating such quantifiers right away, each subproof of the proof can be annotated with a constraint that is a quantifier-free PA formula. When building proofs incrementally, this makes it possible to easily distinguish between unsatisfiable subproofs (i.e., subproofs with an unsatisfiable constraint) that need to be expanded further, and satisfiable subproofs whose expansion can be postponed. Besides, due to Lem. 14 and as only quantifier-free constraints occur, the resulting procedure decides PA.

. The calculus PresPred^C itself can be used to eliminate quantifiers. This is possible because we can observe that the strategy from the previous section is always able to eliminate one level of universal quantifiers:

Proof on page 215

Lemma 15 (Quantifier elimination). *Suppose a formula ϕ does not contain uninterpreted predicates and $\forall$ occurs in ϕ only in positive positions and $\exists$ only in negative positions. The strategy from the previous section produces a proof with root $\vdash \phi \Downarrow C$ in which C does not contain quantifiers (more precisely, if C contains a quantified subformula $Qx.\psi$, then x does not occur in ψ).*

This means that, in order to eliminate universal quantifiers from a formula ϕ, we can construct an exhaustive proof with root $\vdash \phi \Downarrow C$ and extract the constraint C. Similarly, existential quantifiers can be eliminated by constructing a proof for $\phi \vdash \;\Downarrow C$ (also see Example 12).

6 Fair Construction of Proofs

We now compare the calculus $\mathrm{PresPred}^C$ with the more restricted calculus $\mathrm{PresPred}^C_S$ from Sect. 4. Because the former calculus is a superset of the latter, it is a trivial observation that any sequent provable in $\mathrm{PresPred}^C_S$ is also provable in $\mathrm{PresPred}^C$. It can also be shown that $\mathrm{PresPred}^C$ cannot prove more sequents than $\mathrm{PresPred}^C_S$, which means that the two calculi are equivalent.

Lemma 16. *Suppose that a PresPred^C-proof for the sequent $\Gamma \vdash \Delta \Downarrow C$ exists. For some constraint D with $C \Rightarrow D$, there is a PresPred^C_S-proof of the sequent $\Gamma \vdash \Delta \Downarrow D$.*

Proof on page 216

Proofs in $\mathrm{PresPred}^C$ can be found by a backtracking-free fair application strategy. Rules that are specific to integer arithmetic (Fig. 3) are mostly irrelevant for this result: such rules do not hinder the construction of proofs, but their application is not necessary either. Practically, the rules can help to find shorter proofs and reduce the size of constraints involved, however.

To define the notion “fair” formally, it has to be observed that formulae in a $\mathrm{PresPred}^C$-proof can be rewritten by applying RED or SIMP. When this happens, it is possible to identify a unique successor of the modified formula in the premiss of the rule application (vice versa, a formula can have multiple predecessors because distinct formulae could become equal when applying a rule).

A *fair PresPred^C-proof* for a sequent $\Gamma \vdash \Delta \Downarrow ?$ is a possibly infinite proof in $\mathrm{PresPred}^C$ in which all constraints are ? and all branches have the properties:

– *Fair treatment of formulae with uninterpreted predicates:* whenever at some point on the branch one of the rules in Fig. 1 is applicable to a formula that contains uninterpreted predicates, the rule is applied to the formula or to a successor of the formula at some later point on the branch. (This implies that ALL-LEFT and EX-RIGHT are applied infinitely often to each universally quantified formula with uninterpreted predicates).
– *Fair unification of complementary literals:* if there is a sequent on the branch of the shape $\Gamma, p(\bar{t}) \vdash p(\bar{s}), \Delta \Downarrow ?$, the rule PRED-UNIFY is applied at least once on the branch to the pair $p(\bar{t})$, $p(\bar{s})$ or to successors of these formulae.
– *Exhaustiveness:* all proof nodes can be annotated with sets U as in Sect. 5.1.

We say that a constraint C is *generated* by a fair proof of $\Gamma \vdash \Delta \Downarrow ?$ if a (finite) proof for $\Gamma \vdash \Delta \Downarrow C$ can be obtained by chopping off all branches of the fair proof at some point, applying CLOSE in some way to the leaves and propagating the resulting constraints through the proof.

Lemma 17 (Fair construction). *Suppose that a PresPred^C_S-proof for the sequent $\Gamma \vdash \Delta \Downarrow C$ exists. Every fair PresPred^C-proof of $\Gamma \vdash \Delta \Downarrow ?$ whose root is annotated with the set U generates a constraint D with $\forall U.C \Rightarrow \forall U.D$.*

Proof on page 219

Intuitively, this means that every fair proof Q of a provable sequent $\Gamma \vdash \Delta \Downarrow ?$ contains a finite proof Q' of the sequent $\Gamma \vdash \Delta \Downarrow C$ for some valid constraint C (applications of CLOSE have be added to close Q', of course). Moreover, because of Lem. 11, it can be observed that every closed exhaustive proof of $\Gamma \vdash \Delta \Downarrow ?$ that contains Q' as an initial part has a valid constraint. This implies the completeness of proof construction with fair rule, formula, and branch selection.

7 Weakening to Eliminate Irrelevant Formulae

The calculus PresPred^C allows to ignore unneeded formulae when the rule CLOSE is applied, which is used in Sect. 5.1 and 5.2 by selecting only those formulae that do not contain U-constants. Leaving out the formulae that contain U-constants is important for two reasons: it is required for the quantifier elimination lemma (Lem. 15), but it also helps to keep constraints as small as possible. Concerning the latter argument, the precision of the U-criterion can be improved by eliminating irrelevant formulae as early as possible instead of waiting until CLOSE is applied. Since the conditions for exhaustive proofs in Sect. 5.1 require that the set U is reset to $\emptyset$ whenever the rules EX-RIGHT(-D) and ALL-LEFT(-D) occur, it can otherwise happen that formulae that were at some point identified as unnecessary can later in the proof again be considered relevant.

The classical weakening rule for a sequent calculus can directly be carried over to constrained sequents and is sound:

$$\frac{\Gamma \vdash \Delta \Downarrow C}{\Gamma, \Gamma' \vdash \Delta', \Delta \Downarrow C} \text{ WEAKEN}$$

The application of this rule has to be restricted, however, so that Lem. 11 (constraint completeness) and Lem. 17 (fair proof construction) are preserved. In the style of the conditions given in Sect. 5.1, we can assume that the conclusion and the premiss of an application of WEAKEN are both annotated with a set U of constants (in principle, one could also choose different sets for the premiss and the conclusion, but this would not lead to any interesting generalisations at this point). A sufficient condition to preserve Lem. 11 is:

$$\forall U.\ (\Gamma_p, \Gamma_p' \vdash \Delta_p', \Delta_p) \;\Rightarrow\; \forall U.\ (\Gamma_p \vdash \Delta_p)$$

Concerning Lem. 17, the critical point is to ensure that repeated application of PRED-UNIFY to the same complementary literals is not necessary (see the proof of the lemma on page 219).

Two possible criteria that preserve the lemmas are:

- *Elimination of antecedent equations:* $\Gamma' = \{c + t \doteq 0\}, \Delta' = \emptyset$, where $c \in U$ is a constant that does not occur in Γ, Δ.
- *Elimination of a group of satisfiable literals:* in certain cases, a group of inequalities, inequations and divisibility judgements can simultaneously be eliminated:

$$\Gamma' \;=\; \{t_i \dot{\geq} 0\}_i \cup \{t_j' \dot{\leq} 0\}_j, \quad \Delta' \;=\; \{s_k \doteq 0\}_k \cup \{\alpha_l \mid u_l\}_l$$

This is possible if the invalidity of the literals is ensured through a constant $c \in U$ such that:

- no formula in Γ, Δ contains c;
- Γ' contains only lower or only upper bounds on c, i.e., c occurs in each t_i with a positive coefficient and in each t'_j with a negative coefficient, or vice versa;
- c occurs in each s_k with a non-zero coefficient;
- c occurs in each u_l of a divisibility judgement $\alpha_l \mid u_l$ with the non-zero coefficient β_l, and:

$$\sum_l \frac{|\gcd(\alpha_l, \beta_l)|}{|\alpha_l|} < 1 \tag{2}$$

To understand the last requirement, note that the integers (values of c) that satisfy a judgement $\alpha \mid (\beta c + t)$, provided that there are any, are periodical with the following period:

$$\frac{|\operatorname{lcm}(\alpha, \beta)|}{|\beta|} = \frac{|\alpha|}{|\gcd(\alpha, \beta)|}$$

The inequality (2) ensures that there are values for c such that none of the divisibility judgements holds (equivalently, there are infinitely many such values).

In both cases, the constant $c \in U$ that justifies the weakening must not occur in *any* formula in Γ, Δ, i.e., neither in PA formulae nor in formulae that contain uninterpreted predicates. This ensures that Lem. 17 is preserved: unification conditions that can be generated by PRED-UNIFY cannot contain c and are therefore independent of the formulae that are removed by WEAKEN.

8 Refined Constraint Propagation

All calculi that we have defined so far have a severe disadvantage compared to normal FOL calculi: there is no notion of "non-unifiability," because the rule PRED-UNIFY can be applied in a very unrestricted manner to arbitrary pairs of literals that start with the same predicate symbol. This can lead to constraints that contain redundant information and to unnecessary proof splitting. For instance, in the following proof the rule PRED-UNIFY is applicable and introduces a conjunction that can lead to a splitting of the branch:

$$\cfrac{\cfrac{p(c,c) \ \vdash \ c - d \doteq 0 \wedge c - e \doteq 0, p(d,e) \ \Downarrow ?}{p(c,c) \ \vdash \ p(d,e) \ \Downarrow ?}\ \text{PRED-UNIFY}}{\cfrac{\exists x. p(x,x) \ \vdash \ \forall x, y. p(x,y) \ \Downarrow ?}{\cdots}}\ \text{EX-LEFT, ALL-RIGHT}$$

The conjunction $c - d \doteq 0 \wedge c - e \doteq 0$ describes a special case, however, and can be falsified by choosing suitable values for the universally quantified symbols c, d, e. It is therefore not helpful to select this formula when applying CLOSE.

It is not possible to exclude equations involving (only) constants that stem from universal quantifiers altogether. This is because constraints can also contain negated equations or inequalities, so that the validity problem for constraints resembles semantic unification modulo assumptions. Moreover, as the following example shows, it is not possible to decide locally for a proof goal whether an equation is irrelevant or not.

Example 18. A proof in which also seemingly "non-unifiable" equations are essential is the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash\ a \doteq 0, b \doteq 0\ \Downarrow ?}{\vdash\ a \doteq 0 \lor b \doteq 0\ \Downarrow ?} \qquad \dfrac{\vdash\ a \doteq 0 \land b - 1 \doteq 0, a \not\doteq 0 \land b \doteq 0\ \Downarrow ?}{\vdash\ a \doteq 0 \land b - 1 \doteq 0 \lor a \not\doteq 0 \land b \doteq 0\ \Downarrow ?}}{\vdash\ (a \doteq 0 \lor b \doteq 0) \land (a \doteq 0 \land b - 1 \doteq 0 \lor a \not\doteq 0 \land b \doteq 0)\ \Downarrow ?}}{\vdash\ \exists y.(a \doteq 0 \lor y \doteq 0) \land (a \doteq 0 \land y - 1 \doteq 0 \lor a \not\doteq 0 \land y \doteq 0)\ \Downarrow ?}}{\vdash\ \forall x.\exists y.(x \doteq 0 \lor y \doteq 0) \land (x \doteq 0 \land y - 1 \doteq 0 \lor x \not\doteq 0 \land y \doteq 0)\ \Downarrow ?}$$

It can be observed that the formula in the root of the proof is valid. Although the constant a comes from the universal quantifier $\forall x$, none of the formulae $a \doteq 0$ and $a \doteq 0 \land b \doteq 0$ can be left out when applying CLOSE without invalidating the constraint resulting from the proof.

In this section, we define a global criterion that tells which formulae can be ignored when applying CLOSE. This is done by distinguishing those constants in a proof that are introduced by universal quantifiers and that do not occur in illegal positions (we will call such constants *free*). Because there is no necessity to apply rules other than CLOSE to PA formulae (according to the notion of a fair proof in Sect. 6), the application of the rule PRED-UNIFY can be skipped as well if it can be predicted that the generated conjunction is irrelevant.

As a prerequisite, we need to replace the rule DIV-CLOSE with a modified version DIV-CLOSE' that enables us to order the constants in a proof in a more fine-grained way:

$$\dfrac{\Gamma, \alpha c' - t \doteq 0, c - c' \doteq 0\ \vdash\ \Delta\ \Downarrow C}{\Gamma, \alpha c - t \doteq 0\ \vdash\ \Delta\ \Downarrow [x/t]C' \lor \alpha \nmid t}\ \text{DIV-CLOSE'}$$

where c' does not occur in the conclusion and C' is a PA formula such that $C \Leftrightarrow [x/\alpha c']C'$. The rule can essentially be used in the same way as DIV-CLOSE and is not in conflict with any other part of the article. The rule has not been introduced earlier mainly because it would have made the previous sections unnecessarily complicated (but it is, in fact, the rule that is used in the implementation of the calculus, see Sect. 9).

Everywhere in this section, assume that P is an open PresPred^C-proof in which CLOSE is never applied. For reasons of presentation, we further assume that the constants that are introduced in P by the rules ALL-LEFT, ALL-RIGHT, etc. are all pairwise distinct (and also different from "global" constants that are not explicitly introduced by any rule), which can be achieved by renaming.

The proof P induces a strict partial order $\prec_P$ on the set of all constants occurring in P, based on the order of introduction: we define $c \prec_P d$ to hold iff c is a global constant and d is not, or if the rule application that introduces c is on the path from the root of P to the rule application that introduces d (d is introduced "after" c).

Given $\prec_P$, we say that a formula ϕ is *shielded* by a constant c if ϕ is equivalent to a formula $\alpha c + t \doteq 0 \land \psi$ such that $\alpha \neq 0$ and $d \prec_P c$ for all constants d that occur in t. We say that ϕ is shielded by a set of constants M if ϕ is shielded by some constant $c \in M$. This definition is chosen such that the formula $\forall c.\psi$ is equivalent to *false* if ψ is a finite disjunction of formulae that are shielded by c, and similarly for sets M. The maximality condition ensures that shieldedness is preserved by quantification over bigger constants.

We say that Q is a set of *free* constants for the proof P if there is a superset $Q_c \supseteq Q$ of constants such that the following conditions are satisfied:

- all constants in Q are universal in P, i.e., are introduced by the rules ALL-RIGHT, EX-LEFT, or COL-RED;
- whenever COL-RED-SUBST is applied and the term $c - u$ contains a constant from Q_c, then also $c' \in Q_c$;
- whenever DIV-CLOSE' is applied, the term t does not contain any constants from Q_c.

Given such sets Q, Q_c, we now consider two ways to close the proof P by applying CLOSE to each of the goals. The resulting closed proofs are called P_1, P_2, and we demand that they have the following property: whenever CLOSE is applied in P_1, then (i) the disjunction C of selected PA formulae does not contain any constants from Q_c, and (ii) the disjunction of PA formulae selected in P_2 by the corresponding application of CLOSE is equivalent to $C \lor \bigvee_{i=1}^n \phi_i$ such that each formula ϕ_i is shielded by Q and only contains constants that occur in the considered proof goal.

Lemma 19 (Shielded Constraints). *Let C_1 be the constraint of any proof node in P_1 and C_2 the constraint of the corresponding node in P_2. Then (i) C_1 does not contain any constants from Q_c, and (ii) C_2 is equivalent to $C_1 \lor \bigvee_{i=1}^n \phi_i$ where each formula ϕ_i is closed, shielded by Q, and only contains constants that are global or introduced on the path from the proof root to the location of C_2.*

Proof on page 224

It is an implication of the lemma that the constraints that arrive at the roots of P_1 and P_2 are equivalent. This means that the additional formulae that are selected in P_2 when applying CLOSE, compared to P_1, did not contribute to the constraint and could have been left out right away. In case CLOSE is applied in P_2 in the most liberal way (in each goal, all PA formulae are selected), this tells which of the formulae are irrelevant for the proof.

Example 20. We show how the criterion rules out non-unifiable pairs of literals:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f(a,2), f(b,3) \;\vdash\; f(b,c), \ldots \;\Downarrow ? \qquad f(a,2), f(b,3) \;\vdash\; c \dot{\geq} 0, \ldots \;\Downarrow ?}{f(a,2), f(b,3) \;\vdash\; f(b,c) \land c \dot{\geq} 0, \ldots \;\Downarrow ?}\;\textsc{and-right}}{f(a,2), f(b,3) \;\vdash\; \exists z.(f(b,z) \land z \dot{\geq} 0) \;\Downarrow ?}\;\textsc{ex-right}}{f(a,2) \land f(b,3) \;\vdash\; \exists z.(f(b,z) \land z \dot{\geq} 0) \;\Downarrow ?}\;\textsc{and-left}}{\vdash\; f(a,2) \land f(b,3) \to \exists z.(f(b,z) \land z \dot{\geq} 0) \;\Downarrow ?}\;\textsc{or-right, not-right}}{\vdash\; \forall x, y.(f(x,2) \land f(y,3) \to \exists z.(f(y,z) \land z \dot{\geq} 0) \;\Downarrow ?}\;\textsc{all-right} \times 2$$

In the left goal, the rule PRED-UNIFY can be applied to the pairs $f(a,2), f(b,c)$ and $f(b,3), f(b,c)$. Because the constants $Q = Q_c = \{a, b\}$ are free, the first pair can be ignored as it would generate the formula $b - a \doteq 0 \land c - 2 \doteq 0$ that is shielded by b in the first equation.

In the proof in Example 18, the formula $a \not\doteq 0 \land b \doteq 0$ is only shielded by the constant b that comes from an existential quantifier. This means that if the formula is to be selected for CLOSE, neither can a be a free constant, and thus none of the formulae $a \doteq 0$ and $a \doteq 0 \land b \doteq 0$ is shielded either.

There are several ways to generalise the approach described in this section:

- It is possible to vary the definition of "shielded formulae," e.g., to also consider formulae that are shielded through inequalities.
- The ordering $\prec_P$ on constants can be defined less total, again liberalising the notion of shielded formulae: if a sequence of quantifiers of the same kind is instantiated, there is no need to order the introduced constants. In Example 20, this would apply to the constants a, b.
- Sets Q of free constants can be localised, it is not necessary to use the same sets for a whole proof. It can be the case that the conditions for freeness are generally true in a proof, but are violated in a small subproof. In this situation, it might be possible to use a smaller set Q for this particular subproof. In the right subproof of the next example, for instance, c is not free because it occurs in the unshielded formula $d - c \doteq 0$. Because c does not occur in the constraint *true* of the subproof as a whole anymore, however, this is irrelevant for the rest of the proof. Consequently, it might be possible to avoid the unification of $p(c)$ and $p(e)$ in the left subproof.

$$\dfrac{p(c) \;\vdash\; p(e) \;\Downarrow ? \qquad \dfrac{\dfrac{\dfrac{*}{p(c) \;\vdash\; d - c \doteq 0, p(d), \exists x.p(x) \;\Downarrow d - c \doteq 0}}{p(c) \;\vdash\; p(d), \exists x.p(x) \;\Downarrow d - c \doteq 0}}{p(c) \;\vdash\; \exists x.p(x) \;\Downarrow \mathit{true}}}{\cdots}$$

9 Implementation and Initial Experimental Results

We have implemented the calculus defined in this paper (essentially in the version of Sect. 5) in the theorem prover Princess, which is available for download.[1]

[1] `http://www.cse.chalmers.se/~philipp/princess`

QF_LIA			
Directory	*solved/total*	*Directory*	*solved/total*
Averest	10/ 19	nec-smt/small	17/ 35
CIRC	33/ 51	nec-smt/med	3/ 364
RTCL	2/ 2	nec-smt/large	0/2381
check	5/ 5	rings	53/ 293
mathsat	55/121	wisa	4/ 5
QF_IDL			
Averest	195/252	planning	5/ 45
cellar	0/ 14	qlock	0/ 72
check	3/ 3	queens_bench	53/ 297
diamonds	18/ 36	RTCL	30/ 33
DTP	0/ 60	sal	27/ 50
mathsat	96/146	sep	17/ 17
parity	34/248		

Fig. 5. Princess statistics for the categories QF_LIA and QF_IDL of the SMT library [14] (Dual Core AMD Opteron 270 with 2GHz, 1.5GB of heapspace, timeout of 1000s). Detailed results are available at `http://www.cse.chalmers.se/~philipp/princess`.

At the time of writing this section, all features introduced in the paper are implemented, apart from the optimisation of Sect. 8. In certain situations, Princess additionally uses analytic cuts [11] and formula simplification [12] to avoid redundancy in proofs. Constraints are simplified using the approach of Sect. 5.3, which means that no separate decision procedure for Presburger arithmetic is necessary. Princess is written in the Scala programming language [13] and runs on a Java virtual machine.

Most available benchmarks for SMT-solvers require uninterpreted functions or further theories like arrays that have to be handled using appropriate encodings in our calculus. Although we plan to add preprocessors for these theories to Princess, implementations of such encodings are not available yet (and also require further research in some cases). Our experimental results up to now are, therefore, restricted to the categories QF_LIA (quantifier-free linear integer arithmetic) and QF_IDL (quantifier-free integer difference logic) of the SMT library [14], see Fig. 5.

Although Princess is not primarily designed for the problems in the tested categories (in contrast to SMT-solvers), the results are reasonably good. Unsurprisingly, Princess performs better for problems that focus on arithmetic (like CIRC) than for problems that are essentially combinatoric (like queens_bench), for which more advanced techniques developed for SAT- and SMT-solvers are necessary. This fact might also explain the poor results for the nec-smt directories. In the SMT competition in 2007,[2] Princess would have solved 109 out of 203 selected problems in QF_LIA and 85 out of 203 problems in QF_IDL. For 2008, the result drops to only 10 out of 205 problems in QF_LIA and 5 out of 203 in QF_IDL,

[2] `http://www.smtcomp.org/`

presumably because of the poor performance for combinatoric problems and the lack of lemma learning (QF_LIA is in 2008 dominated by nec-smt problems).

10 Related Work

Model evolution modulo linear integer arithmetic [7] is a recently proposed variant of the Model Evolution calculus that is similar to our calculus in that it supports PA enhanced with uninterpreted predicates (and without functions) as input language, and that its architecture resembles tableau calculi. Model Evolution does not use rigid free variables that are shared among different branches in the way tableaux do, however, which means that also constraints can be kept branch-local. Further differences are that $\mathcal{ME}$(LIA) works on clauses, only supports a restricted form of existential quantification, and has a more explicit representation of candidate models.

SMT-solvers based on the DPLL(T) architecture [15] can handle ground problems modulo integer arithmetic (and many other theories) efficiently, but only offer heuristic quantifier handling. Because of the similarity between DPLL and sequent calculi, the work presented in this paper can be seen as an alternative approach to handling quantifiers that should also be applicable to DPLL(T).

Our approach has similarities with the framework in [16] for integrating theories into tableau calculi by distinguishing between a foreground reasoner (handling FOL) and a background reasoner (handling the theory). According to this nomenclature, the rules in Fig. 3 implement the (partial) background reasoner. Because our theory rules operate destructively on sequents, we integrate background and foreground reasoning more closely than proposed in [16]. The biggest difference between our approach and [16] is that no theory unification is performed in our calculus, it is only necessary to check the validity of constraints.

The simplification of formulae by the rules in Fig. 3 is roughly comparable with deduction modulo [17]. The concept is here integrated in a setting that resembles free-variable tableaux to treat quantifiers more efficiently.

An approach to embed algebraic constraints in tableau calculi is described in [18], where quantifier elimination tasks in real arithmetic (possibly involving more than one proof goal) are carried out by an external procedure, in a manner comparable to the simultaneous solving of constraints from multiple proof goals described here. Uninterpreted functions or predicates are not handled.

There are a number of approaches to include theories into resolution-based calculi. [19] works with constraints that are solved in a theory, but requires to enumerate the solutions of constraints (whereas it is enough to check the validity of constraints in our work). In [20], while it is enough to check satisfiability of constraints, no uninterpreted functions or predicates are supported. A recent calculus to handle rational arithmetic is given in [21], and is similar to our work in that it has built-in rules to solve systems of equations and inequalities (based on Fourier-Motzkin). The calculus is complete under restrictions that effectively prevent quantification over rationals. It remains to be investigated how this fragment is related to the fragments discussed here.

Constraint (logic) programming [22] combines constraint solving (in arithmetic or other theories) with ordinary programming and can solve problems in the fragment that is considered in Lem. 7 (formulae in which $\exists$ only occurs in positive and $\forall$ only in negative positions). While the lemma shows that our calculus is complete for a certain class of constraint programs, the calculus is applicable to a more general set of formulae containing, e.g., arbitrary quantifiers.

11 Conclusions and Future Work

We have presented a novel calculus to reason about problems in first-order logic modulo linear integer arithmetic. The calculus is complete for function-free first-order logic (on such problems, proofs in the calculus resemble free-variable tableaux with incremental closure [1]) and can decide Presburger arithmetic (in a manner that is similar to the Omega test [2]). As further results, we have shown that the calculus is at least as complete as the calculus $\mathcal{ME}$(LIA), and allows the fair construction of proofs. We have also described refinements of the calculus and given experimental results.

Apart from continuing the implementation and further benchmarks, there are a number of concepts that require more research, among others: the encoding and handling of functions and further theories; the integration of lemma learning; the integration of connectivity conditions to make proof search more directed; the elimination of cuts in proofs; an analysis of the complexity of the calculus as a decision procedure for PA. We also plan to extend our calculus to support nonlinear arithmetic (following the work in [8]), and possibly rational arithmetic.

Acknowledgements

I want to thank Wolfgang Ahrendt, Nikolaj Bjørner, Richard Bubel, Reiner Hähnle, and Henrik Johansson for discussions and comments during different stages of the work presented in this paper. I am also grateful for comments from the anonymous referees.

References

1. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, First International Joint Conference on Automated Reasoning, Siena, Italy. Volume 2083 of LNCS, Springer (2001) 545–560
2. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM **8** (1992) 102–114
3. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. (On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation). In: Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929, Warsaw, Poland (1930) 92–101,395

4. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
5. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Communications of the ACM **22** (1979) 465–476
6. Cooper, D.: Theorem proving in arithmetic without multiplication. Machine Intelligence **7** (1972) 91–99
7. Baumgartner, P., Fuchs, A., Tinelli, C.: MELIA – model evolution with linear integer arithmetic constraints. To appear (2008)
8. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In Beckert, B., ed.: Proceedings, Fourth International Verification Workshop, Bremen, Germany. Volume 259 of CEUR (`http://ceur-ws.org/`) (2007)
9. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
10. Norrish, M.: Complete integer decision procedures as derived rules in HOL. In: Proceedings, Theorem Proving in Higher Order Logics. Volume 2758 of LNCS, Springer (2003) 71–86
11. D'Agostino, M.: Tableaux methods for classical propositional logic. In D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer, Dordrecht (1999) 45–123
12. Massacci, F.: Simplification: A general constraint propagation technique for propositional and modal tableaux. In de Swart, H., ed.: Proceedings, International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oosterwijk, The Netherlands. Volume 1397 of LNCS, Springer (1998) 217–232
13. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc., Mountain View, CA, USA (2008) To appear.
14. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smt-lib.org` (2008)
15. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM **53** (2006) 937–977
16. Beckert, B.: Equality and other theories. In D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J., eds.: Handbook of Tableau Methods. Kluwer, Dordrecht (1999) 197–254
17. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. Journal of Automated Reasoning **31** (2003) 33–72
18. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning **41** (2008) 143–189
19. Stickel, M.E.: Automated deduction by theory resolution. Journal of Automated Reasoning **1** (1985) 333–355
20. Bürckert, H.J.: A resolution principle for clauses with constraints. In Stickel, M.E., ed.: Proceedings, 10th International Conference on Automated Deduction. Volume 449 of LNCS, Springer (1990) 178–192
21. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In Duparc, J., Henzinger, T.A., eds.: Proceedings, 21st International Workshop on Computer Science Logic. Volume 4646 of LNCS, Springer (2007) 223–237
22. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings, 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM (1987) 111–119
23. Hintikka, J., Sandu, G.: Game-theoretical semantics. In Benthem, J.F.V., Meulen, A.G.T., eds.: Handbook of Logic and Language. MIT Press, Cambridge, Massachusetts (1997)

24. Gale, D., Stewart, F.M.: Infinite games with perfect information. In Kuhn, H.W., Tucker, A.W., eds.: Contributions to the Theory of Games II. Volume 28 of Annals of Mathematics Studies. Princeton University Press, Princeton NJ (1953) 245–266

A Proofs

Lemma 2 (Soundness of Pred^C)

All rules of the calculus Pred^C are sound in the sense introduced in Sect. 2 (but considering evaluation over arbitrary first-order structures (U, I) with an arbitrary non-empty universe U). Some of the cases are:

- AND-RIGHT: Assume $\Gamma \vdash \phi \wedge \psi, \Delta \Downarrow C \wedge D$ is invalid, i.e., for some structure (U, I) and some constant assignment δ we have $val_{(U,I),\delta}(C \wedge D) = tt$ but $val_{(U,I),\delta}(\Gamma \vdash \phi \wedge \psi, \Delta) = ff$. This implies:
  - $val_{(U,I),\delta}(C) = tt$ and $val_{(U,I),\delta}(D) = tt$,
  - $val_{(U,I),\delta}(\phi) = ff$ or $val_{(U,I),\delta}(\psi) = ff$.

  Then $val_{(U,I),\delta}(\Gamma \vdash \phi, \Delta \Downarrow C) = ff$ or $val_{(U,I),\delta}(\Gamma \vdash \psi, \Delta \Downarrow D) = ff$ and one of the premisses has to be invalid.
- OR-RIGHT: Assume that $\Gamma \vdash \phi \vee \psi, \Delta \Downarrow C$ is invalid, i.e., for some structure (U, I) and some constant assignment δ we have $val_{(U,I),\delta}(C) = tt$ but $val_{(U,I),\delta}(\Gamma \vdash \phi \vee \psi, \Delta) = ff$. This implies:
  - $val_{(U,I),\delta}(\phi) = ff$ and $val_{(U,I),\delta}(\psi) = ff$.

  Then also $val_{(U,I),\delta}(\Gamma \vdash \phi, \psi, \Delta \Downarrow C) = ff$ and the premiss is invalid.
- ALL-RIGHT: Assume that $\Gamma \vdash \forall x.\phi, \Delta \Downarrow \forall x.C$ is invalid, i.e., for some structure (U, I) and some constant assignment δ we have $val_{(U,I),\delta}(\forall x.C) = tt$ but $val_{(U,I),\delta}(\Gamma \vdash \forall x.\phi, \Delta) = ff$. Because c is a fresh constant, this implies that there is an assignment δ' that agrees with δ on all constants but c such that $val_{(U,I),\delta'}([x/c]\phi) = ff$. For this δ', also $val_{(U,I),\delta'}([x/c]C) = tt$ holds. Then $val_{(U,I),\delta'}(\Gamma \vdash [x/c]\phi, \Delta \Downarrow [x/c]C) = ff$ and therefore the premiss is invalid.
- EX-RIGHT: Assume that $\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.C$ is invalid, i.e., for some structure (U, I) and some constant assignment δ we have $val_{(U,I),\delta}(\exists x.C) = tt$ but $val_{(U,I),\delta}(\Gamma \vdash \exists x.\phi, \Delta) = ff$. Because c is a fresh constant, this implies that there is an assignment δ' that agrees with δ on all constants but c such that $val_{(U,I),\delta'}([x/c]C) = tt$ and $val_{(U,I),\delta'}([x/c]\phi) = ff$. Then $val_{(U,I),\delta'}(\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow [x/c]C) = ff$ and the premiss is invalid.

The conjecture follows by a simple induction on the size of proofs.

Lemma 3 (Completeness of Pred^C)

We assume that $\vdash \phi \Downarrow C$ is unprovable for all valid formulae C and deduce that ϕ is not valid. The main difficulty with the approach is to use the knowledge

"All constraints that can be derived during a proof attempt are invalid"

to identify a saturated proof branch from which a countermodel can be constructed using the normal Hintikka-construction [4]. We bridge this gap by introducing a notion of "most general constraints" (mgc), which are basically constraints that are the disjunction of all closing constraints that can be derived from a proof. Because saturated proofs are usually infinite, also mgc can be infinitary. Importantly, we can show that the mgc is valid iff there is a valid closing constraint. From the fact that the mgc is invalid (if a formula is unprovable), we can derive a saturated proof branch that we turn into a countermodel.

To derive most general constraints, we modify the closure rules of Pred^C and allow to introduce constraints with disjunctions. The calculus Pred^C_J consists of the same rules as Pred^C apart from PRED-CLOSE, which is replaced with the two following rules:

$$\frac{\Gamma, p(s_1, \ldots, s_n) \vdash p(t_1, \ldots, t_n), \Delta \Downarrow C}{\Gamma, p(s_1, \ldots, s_n) \vdash p(t_1, \ldots, t_n), \Delta \Downarrow C \lor \bigwedge_i s_i \doteq t_i} \text{ DISJ-CLOSE}$$

$$\frac{*}{\Gamma \vdash \Delta \Downarrow \mathit{false}} \text{ FALSE-CLOSE}$$

We call a (possibly infinite) proof tree in Pred^C_J *fair* if

– all structural/propositional rules and skolemisation are eventually applied whenever they are applicable,
– the rule DISJ-CLOSE is applied infinitely often to each complementary pair on each branch,
– the rule ALL-LEFT/EX-RIGHT is applied infinitely often to every universally/existentially quantified formula in the antecedent/succedent on each branch,
– the rule FALSE-CLOSE is only applied if no other rule is applicable.

The constraints generated by fair proof trees are called *most general constraints.* Such constraints can be infinitary and contain infinitely many quantifiers, disjunctions (due to infinite branches in a proof tree) or conjunctions (due to infinitely many branches in a proof tree). We consider infinitary formulae as infinite trees, in which conjunctions and disjunctions are always seen as binary connectives. Because no function symbols are involved, terms are always single variables or constants, and literals occurring in the formulae are always finite. We use game semantics (see [23] for an introduction) to give meaning to infinitary formulae.

Infinitary Formulae

Starting with a fixed non-empty domain U, initial variable/constant assignments β, δ and at the root of a formula, two players (the *verifier* and the *falsifier*) play against each other:

– the verifier tries to show that the formula is true. When arriving at $\lor$ or $\exists x$, the verifier has to choose the subformula to continue with, or the value that x is to be given.
– the falsifier tries to show that the formula is false. When arriving at $\land$ or $\forall x$, the falsifier has to choose the subformula to continue with, or the the value that x is to be given.

The verifier wins if the game ends and arrives at a literal that evaluates to tt. The falsifier wins if the game does not terminate,[3] or if it ends at a literal that evaluates to ff. A formula is true (in a certain structure) if the verifier has a (deterministic) winning strategy, i.e., the verifier can win whatever the falsifier does, and false otherwise. By the theorem of Gale-Stewart [24], in the second case the falsifier has a (deterministic) winning strategy. A formula is valid if the verifier has winning strategies for each domain U and assignments β, δ.

It is well-known that the above game semantics coincides with Tarski semantics for finite formulae.

Properties of Most-General Constraints

The important property of the most-general constraint obtained from a fair Pred_J^C-proof is that it subsumes all constraints that the original calculus Pred^C could possibly have generated. By *possibly generated constraints* we mean all constraints that can be obtained from the Pred_J^C-proof by turning the proof into a Pred^C-proof:

- remove all applications of FALSE-CLOSE and DISJ-CLOSE,
- chop off all infinite branches at some point to make the proof tree finite, and
- apply PRED-CLOSE in some way to all open branches.

The constraint C that arrives at the root $\Gamma \vdash \Delta \Downarrow C$ of the resulting proof is called *possible generated.*

Lemma 21. *The most-general constraint of a fair Pred_J^C-proof is valid iff the proof possibly generates a valid constraint.*

Assume that all possibly generated constraints of a fair Pred_J^C-proof are invalid. By the lemma, this means that the mgc of this proof is invalid: for some particular domain, the falsifier has a winning strategy for the mgc. We can then discover the right branch in the proof tree and simultaneously construct a countermodel based on the above domain by playing a game (that the falsifier wins).

Proof (Lem. 21).

"$\Longleftarrow$:" Assume that a fair Pred_J^C-proof possibly generates a valid constraint. We fix a domain U, variable/constant assignments β, δ, and a winning strategy S_1 for the verifier for this valid constraint (note, that S_1 is only responsible for nodes $\exists x$). We can derive a winning strategy S_2 for the verifier for the mgc:

- Initially, S_1 and S_2 behave in the same way when arriving at a node $\exists x$. When arriving at $\vee$ in the mgc, which has to be introduced by DISJ-CLOSE, S_2 chooses the left branch (and not the conjunction $\bigwedge_i s_i \doteq t_i$).
- Once the game has reached a point where the rule PRED-CLOSE was applied to produce a conjunction ϕ in the possibly generated constraint, S_2 changes its behaviour:

[3] Note, that this implies that $true \wedge true \wedge \cdots$ is false, which is the intended semantics.

- when arriving at a node $\exists x$, the value of x is chosen arbitrarily;
- when arriving at $\vee$, which has to be introduced by DISJ-CLOSE, S_2 chooses the left branch if it leads to the conjunction ϕ, otherwise the right branch.

Because of fairness, DISJ-CLOSE is eventually applied to every complementary pair on every branch, so that S_2 is guaranteed to win after a finite number of steps.

"$\Longrightarrow$:" Assume that the mgc is valid. Further, assume that all variables that are bound in the mgc are pairwise distinct, and that for each variable x the symbol f_x denotes a function symbol whose arity equals the number of variables that are bound above the location where x is bound. Finally, let U be the domain of the free term algebra over the vocabulary consisting of the functions f_x (we possibly have to add further functions that do not belong to any variables to ensure that U is non-empty).

We fix U as the domain of evaluation (and arbitrary assignments β, δ) as well as a winning verifier strategy S. For each branch b in the proof tree, we construct a falsifier strategy T_b:

– When arriving at a quantifier $\forall x$, the falsifier chooses $f_x(\beta(y_1), \dots, \beta(y_k))$ as the value of x, where $\beta(y_1), \dots, \beta(y_k)$ are the values given to variables bound above $\forall x$.
– When arriving at $\wedge$ that was introduced by AND-RIGHT or OR-LEFT, the falsifier follows the branch b.
– When arriving at other $\wedge$, the falsifier chooses an arbitrary branch.

Because S is a winning strategy, the verifier wins against each of the strategies T_b in a finite number of steps, which means that S picks one particular application of DISJ-CLOSE on each branch. Because S is deterministic and the strategies T_b only differ in the treatment of $\wedge$, no two selected DISJ-CLOSE applications are located on the same branch. This implies that the selected DISJ-CLOSE applications can be replaced with PRED-CLOSE to turn the fair Pred_J^C-proof into a Pred^C-proof (removing all other applications of DISJ-CLOSE and FALSE-CLOSE).

The constraint of the resulting Pred^C-proof is valid: because U is the domain of a term algebra, the values chosen by the verifier for the existentially quantified variables describe a simultaneous unifier of the equations produced by PRED-CLOSE.

Selection of the right branch

We will now show how the reasoning of the previous pages can be used to detect the right saturated branch in a proof tree and to construct a countermodel of the formulae on this branch. To this end, assume that all possibly generated constraints of a fair Pred_J^C-proof are invalid. By Lem. 21, this means that the mgc of this proof is invalid: for some particular domain, the falsifier has a winning strategy for the mgc (note, that the mgc only contains equations between variables as atoms). Wlog, we may assume that the domain is countable and consists of the elements $a_1, a_2, \dots$ (this follows by the same argument as in the proof of Lem. 21: if no such countable domain would exist, we could construct a countable term algebra for which the verifier has a winning strategy and conclude the validity of the mgc).

Assume that the constants that the rules EX-RIGHT, ALL-LEFT introduce in the Pred^C_J-proof are all pairwise distinct. We now discover the right branch in the proof tree and simultaneously construct a countermodel based on the above domain by playing a game. The falsifier will use its winning strategy, whereas we assume that the verifier behaves as follows:

- When arriving at $\vee$ in the mgc, which has to be introduced by DISJ-CLOSE, the verifier chooses the left branch (and not the conjunction $\bigwedge_i s_i \doteq t_i$);
- when arriving at quantifiers $\exists x$ in the mgc, i.e., at a quantified formula $\exists x.\phi$ (succedent) or $\forall x.\phi$ (antecedent) in the proof that is instantiated by EX-RIGHT or ALL-LEFT, the verifier chooses a domain element a_i as value of x. The value a_i is taken when the formula $\exists x.\phi$ or $\forall x.\phi$ is visited the i-time in the game, which means that all domain elements are systematically enumerated for each formula $\exists x.\phi$ or $\forall x.\phi$.

The path chosen by the game corresponds to one branch $S_0, S_1, \dots$ (a sequence of sequents) in the proof tree.

Countermodel Construction. In order to construct a countermodel, we first define the notion of *persistent formulae* on the selected branch. By

$$Lit(\phi_1, \dots, \phi_n \;\vdash\; \psi_1, \dots, \psi_m \;\Downarrow ?) \;:=\; \{\neg\phi_1, \dots, \neg\phi_n, \psi_1, \dots, \psi_m\}$$

we denote the set of literals represented by a sequent. The set of *persistent formulae* of a sequence of sequents is then defined as

$$PF \;:=\; PF(S_0, S_1, \dots) \;:=\; \bigcup_i \bigcap_{j \geq i} Lit(S_j)$$

For predicate calculus, there are two kinds of formulae that can be persistent: existentially quantified formulae (which have to be instantiated multiple times and never disappear from a branch) and literals (to which no further rules apart from closure rules can be applied).

It is now simple to find a countermodel of all persistent atoms:

- We choose the same domain as for the game that was played in the previous section.
- We interpret constants with the values that were chosen by the verifier and the falsifier during the game. Because we assumed that the constants that are introduced by EX-RIGHT, ALL-LEFT are pairwise distinct, this yields a consistent valuation.
- We evaluate all persistent literals $p(\bar{t})$ with *ff* and all persistent literals $\neg p(\bar{t})$ with *ff* as well (i.e., $p(\bar{t})$ with *tt*). This is consistent, because whenever possibly conflicting literals $p(\bar{t})$ and $\neg p(\bar{s})$ are both persistent, we know that

DISJ-CLOSE has been applied to the pair and that the formula $\bigwedge_i s_i \doteq t_i$ evaluates to *ff* for the chosen valuation of constants (otherwise, the verifier could have won the game, which contradicts the assumption that the falsifier has a winning strategy).

To show that the chosen structure is a countermodel of *all* formulae on the proof branch, we perform the usual Hintikka-style induction on the size of formulae. The quantifier cases are the most interesting ones:

- $\forall x.\phi$ in the succedent: we know that ALL-RIGHT has been applied to the formula, and that $[x/c]\phi$ evaluates to *ff* for some constant c. Then also the quantified formula evaluates to *ff*.
- $\exists x.\phi$ in the succedent: EX-RIGHT has been applied infinitely often to the formula, and by the choice of the verifier the values of the introduced constants $c_1, c_2, \ldots$ enumerate all domain elements. Because all formulae $[x/c_1]\phi$, $[x/c_2]\phi, \ldots$ are known to evaluate to *ff*, also $\exists x.\phi$ evaluates to *ff*.

If we have found a countermodel for all sequents on a proof branch, then also the root of the proof tree is invalid:

Lemma 22. *If all possibly generated constraints of a fair* Pred^C_J*-proof for the sequent* $\Gamma \vdash \Delta \Downarrow ?$ *are invalid, then the root* $\Gamma \vdash \Delta$ *of the proof is invalid.*

This implies Lem. 3.

To see that also Lem. 4 holds, observe that every partial proof of $\vdash \phi \Downarrow ?$ (as described in the lemma) can be extended to a fair Pred^C_J-proof. By Lem. 22 and because ϕ is valid, this implies that some possibly generated constraint is valid as well. Because atoms are always persistent in Pred^C_J, this constraint is also generated by a finite extension of the original Pred^C-proof.

Lemma 6 (Universal Completeness of PresPred^C_S)

Exhaustive application of all rules apart from ALL-LEFT, EX-RIGHT and CLOSE terminates. Subsequently, apply CLOSE on each goal as liberally as possible, selecting all PA formulae in the goal (the literals containing uninterpreted predicates have to be left out). If the resulting constraint C (for the whole proof tree) is not valid, a countermodel can be constructed as follows:

- Because C is not valid, there has to be an assignment δ of the constants introduced when applying EX-LEFT, ALL-RIGHT such that the constraint extracted from one of the proof goals evaluates to *ff*. Denote this proof goal by $\Gamma' \vdash \Delta' \Downarrow D$.
- Because PRED-UNIFY has been applied exhaustively in the proof, for any complementary pair $p(\bar{t}) \in \Gamma'$, $p(\bar{s}) \in \Delta'$ the argument vectors evaluate to different integer vectors given the constant assignment δ. This means that a consistent interpretation I of the predicates can be constructed from $\Gamma' \vdash \Delta' \Downarrow D$.
- Using the normal Hintikka construction, it can be shown that I is a countermodel of the original sequent $\Gamma \vdash \Delta$.

Lemma 7 (Existential Completeness of PresPred^C_S)

To prove this, we first need a further lemma:

Lemma 23 (Constant Substitution in Proofs). *Suppose $\Gamma \vdash \Delta$ is a sequent, $\sigma = [c_1/\alpha_1, \ldots, c_n/\alpha_n]$ a substitution that replaces constants with integer literals, and C a constraint, such that $\sigma(\Gamma) \vdash \sigma(\Delta) \Downarrow C$ has a proof in the calculus PresPred^C_S. Then there is a constraint D such that (i) $\Gamma \vdash \Delta \Downarrow D$ has a proof in PresPred^C_S, and (ii) the implication $C \Rightarrow \sigma(D)$ holds.*

Proof. By induction on the size of the proof of $\sigma(\Gamma) \vdash \sigma(\Delta) \Downarrow C$. We can first observe that the existence of a proof for $\Gamma \vdash \Delta \Downarrow C$ implies the existence of proofs for $\Gamma, \Gamma' \vdash \Delta', \Delta \Downarrow C$. Some of the cases are then:

– The last rule applied in the proof is CLOSE:

$$\frac{*}{\sigma(\Gamma, \phi_1, \ldots, \phi_n) \vdash \sigma(\psi_1, \ldots, \psi_m, \Delta) \Downarrow \sigma(\neg\phi_1 \vee \cdots \vee \neg\phi_n \vee \psi_1 \vee \cdots \vee \psi_m)}\ \text{C.}$$

The non-ground proof can then simply be constructed as:

$$\frac{*}{\Gamma, \phi_1, \ldots, \phi_n \vdash \psi_1, \ldots, \psi_m, \Delta \Downarrow \neg\phi_1 \vee \cdots \vee \neg\phi_n \vee \psi_1 \vee \cdots \vee \psi_m}\ \text{CLOSE}$$

– The last rule applied in the proof is NOT-RIGHT:

$$\frac{\begin{array}{c}\vdots\\ \sigma(\Gamma), \sigma(\phi) \vdash \sigma(\Delta \backslash \sigma^{-1}(\{\sigma(\neg\phi)\})) \Downarrow C\end{array}}{\sigma(\Gamma) \vdash \sigma(\neg\phi), \sigma(\Delta) \Downarrow C}\ \text{NOT-RIGHT}$$

Using the induction hypothesis, for a suitable constraint D there is a proof of $\Gamma, \phi \vdash \Delta \backslash \sigma^{-1}(\{\sigma(\neg\phi)\}) \Downarrow D$, therefore also of $\Gamma, \phi \vdash \Delta \Downarrow D$, which can be continued as follows:

$$\frac{\begin{array}{c}\vdots\\ \Gamma, \phi \vdash \Delta \Downarrow D\end{array}}{\Gamma \vdash \neg\phi, \Delta \Downarrow D}\ \text{NOT-RIGHT}$$

– The last rule applied in the proof is EX-RIGHT:

$$\frac{\begin{array}{c}\vdots\\ \sigma(\Gamma) \vdash \sigma([x/c]\phi), \sigma(\exists x.\phi), \sigma(\Delta) \Downarrow [x/c]C\end{array}}{\sigma(\Gamma) \vdash \sigma(\exists x.\phi), \sigma(\Delta) \Downarrow \exists x.C}\ \text{EX-RIGHT}$$

Again, for some constraint D such that $[x/c]C \Rightarrow \sigma(D)$ there is a proof of $\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow D$, and by renaming we can establish $c \notin \{c_1, \ldots, c_n\}$. The proof can be continued as

$$\frac{\begin{array}{c}\vdots\\ \Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow D\end{array}}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.[c/x]D}\ \text{EX-RIGHT}$$

and the implication $\exists x.C \Rightarrow \exists x.[c/x]\sigma(D) \Leftrightarrow \sigma(\exists x.[c/x]D)$ holds.

Proof (Lem. 7). As the first step, we consider a calculus PresPred^C_G that coincides with PresPred^C_S, only that the rules ALL-LEFT, EX-RIGHT are replaced with “ground versions” (α ranges over integer literals):

$$\frac{\Gamma \vdash [x/\alpha]\phi, \exists x.\phi, \Delta \Downarrow C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow C}\ \text{EX-RIGHT-G}$$

$$\frac{\Gamma, [x/\alpha]\phi, \forall x.\phi \vdash \Delta \Downarrow C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow C}\ \text{ALL-LEFT-G}$$

The valid sequent $\Gamma \vdash \Delta$ from the lemma has a proof with valid constraint in PresPred^C_G: otherwise, construct a (possibly infinite) proof tree in which every quantified formulae on every branch has been instantiated with every integer literal. Using the normal Hintikka construction, a countermodel of $\Gamma \vdash \Delta$ can be found.

By induction on the size of the PresPred^C_G-proof, we can transform the proof into a PresPred^C_S-proof. Most steps in the proof are left untouched, the only changes are applied to EX-RIGHT-G, ALL-LEFT-G. For the first case (the latter case is similar), suppose the ground proof ends with:

$$\frac{\Gamma \vdash [x/\alpha]\phi, \exists x.\phi, \Delta \Downarrow C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow C}\ \text{EX-RIGHT-G}$$

where C is valid. We replace the application of EX-RIGHT-G with EX-RIGHT:

$$\frac{\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow ?}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow ?}\ \text{EX-RIGHT}$$

Because $\Gamma \vdash [x/\alpha]\phi, \exists x.\phi, \Delta \Downarrow C$ has a proof in PresPred^C_S, by Lem. 23 there is a proof of $\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow D$ for a suitable D such that $C \Rightarrow [c/\alpha]D$, i.e., $[c/\alpha]D$ is valid. This means that also $\exists x.[c/x]D$ is valid and the translated proof is:

$$\frac{\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow D}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.[c/x]D}\ \text{EX-RIGHT}$$

Lemma 8 (Completeness on the $\mathcal{ME}$(LIA) fragment)

By constructing a PresPred^C_S-proof for each solution of ϕ (with the help of Lem. 7), which can then be combined into a single proof. We first need a further lemma that allows us to restructure proofs:

Lemma 24. *Suppose a PresPred^C_S-proof exists for the sequent $\Gamma \vdash \Delta \Downarrow C$, and $\Gamma \vdash \Delta$ contains a formula ϕ to which one of the rules* AND-*, OR-*, NOT-*, EX-LEFT, ALL-RIGHT *is applicable. For some D with $C \Rightarrow D$ there is a PresPred^C_S-proof of $\Gamma \vdash \Delta \Downarrow D$ in which the first rule application is performed on ϕ. The depth of the new proof (the length of the longest branch) is at most* 1 *bigger than the depth of the original proof, and the first rule application of the* *original proof is the first or the second rule application on all branches in the new proof. Further, if the original proof does not contain any rule applications to PA formulae apart from* CLOSE*, then the new proof does not contain any such applications apart from (possibly) the first rule application and* CLOSE*.*

Proof. Call the original proof P. The main difficulty in the proof of the lemma comes from the fact that sequents consist of sets of formulae (not of multisets), which means that multiple occurrences of a formula are implicitly contracted to only one occurrence. We therefore prove the lemma in two steps: we first show it under the assumption that antecedents and succedents in fact are multisets; as second step, it is then shown that a proof with multiset sequents can be transformed to a proof with ordinary constrained sequents.

Step 1 (proofs with multiset sequents). If P does not contain any rule application to ϕ, we simply add one as first rule application in the new proof and are finished (note, that the constraint of the proof stays equivalent). Otherwise, we show that applications of AND-*, OR-*, NOT-*, EX-LEFT, ALL-RIGHT to ϕ can be shifted towards the root in P. By an inductive argument on the size of P, assume that the second rule application on all branches of P is an application to ϕ, whereas the first rule application is done to a different formula. Note, that whenever the constraint of an inner proof node is weakened, also the constraint of the whole proof becomes weaker or does not change. The cases to be considered are:

– ϕ starts with $\neg$, with $\wedge$ and is in the antecedent, or with $\vee$ and is in the succedent. In all cases, we can simply permute the first and the second rule application in P. Because the rules that can be applied to ϕ do not affect the constraint, the overall constraint stays the same.
– ϕ starts with $\vee$ and is in the antecedent, or with $\wedge$ and is in the succedent. Again, we can permute the first and the second rule application in P, but have to argue that the constraint stays equivalent or becomes weaker. The most interesting situation is the one where the first rule application in P is ALL-LEFT or EX-RIGHT, e.g.:

$$\dfrac{\dfrac{\Gamma \vdash \ldots, \phi_1, \Delta \Downarrow [x/c]C \qquad \Gamma \vdash \ldots, \phi_2, \Delta \Downarrow [x/c]D}{\Gamma \vdash \exists x.\psi, [x/c]\psi, \phi_1 \wedge \phi_2, \Delta \Downarrow [x/c](C \wedge D)}\ \textsc{and-right}}{\Gamma \vdash \exists x.\psi, \phi_1 \wedge \phi_2, \Delta \Downarrow \exists x.(C \wedge D)}\ \textsc{ex-right}$$

is transformed into:

$$\dfrac{\dfrac{\Gamma \vdash \ldots, [x/c]\psi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \exists x.\psi, \phi_1, \Delta \Downarrow \exists x.C}\ \textsc{ex-r.} \qquad \dfrac{\Gamma \vdash \ldots, [x/c]\psi, \Delta \Downarrow [x/c]D}{\Gamma \vdash \exists x.\psi, \phi_2, \Delta \Downarrow \exists x.D}\ \textsc{ex-r.}}{\Gamma \vdash \exists x.\psi, \phi_1 \wedge \phi_2, \Delta \Downarrow \exists x.C \wedge \exists x.D}\ \textsc{and-r.}$$

Because of $\exists x.(C \wedge D) \Rightarrow \exists x.C \wedge \exists x.D$, the resulting constraint does not becomes stronger.
– ϕ starts with $\exists$ and is in the antecedent, or with $\forall$ and is in the succedent. Again, we can permute the first and the second rule application in P. The

most interesting situation is the one where the first rule application in P is ALL-LEFT or EX-RIGHT, e.g.:

$$\cfrac{\cfrac{\Gamma \vdash \exists x.\psi, [x/c]\psi, [y/d]\phi', \Delta \Downarrow [y/d][x/c]C}{\Gamma \vdash \exists x.\psi, [x/c]\psi, \forall y.\phi', \Delta \Downarrow \forall y.[x/c]C}\ \textsc{all-right}}{\Gamma \vdash \exists x.\psi, \forall y.\phi', \Delta \Downarrow \exists x.\forall y.C}\ \textsc{ex-right}$$

is transformed into:

$$\cfrac{\cfrac{\Gamma \vdash \exists x.\psi, [x/c]\psi, [y/d]\phi', \Delta \Downarrow [y/d][x/c]C}{\Gamma \vdash \exists x.\psi, [y/d]\phi', \Delta \Downarrow \exists x.[y/d]C}\ \textsc{ex-right}}{\Gamma \vdash \exists x.\psi, \forall y.\phi', \Delta \Downarrow \forall y.\exists x.C}\ \textsc{all-right}$$

Because of $\exists x.\forall y.C \Rightarrow \forall y.\exists x.C$, the resulting constraint does not become stronger.

Step 2 (elimination of multiple occurrences of a formula). By induction on the size of a proof Q of a multiset sequent $\Gamma \vdash \phi, \phi, \Delta \Downarrow C$ with two (or more) occurrences of a formula ϕ in the succedent (or, analogously, in the antecedent), we show that: there is a proof Q' of the sequent $\Gamma \vdash \phi, \Delta \Downarrow D$ that is not bigger that Q, such that $C \Rightarrow D$. If Q does not contain any rule applications to PA formulae apart from CLOSE, then neither does Q'.

Wlog., we can assume that the first rule application in Q is done to one of the occurrences of ϕ (otherwise, consider the maximal subtrees of Q with this property, which are strictly smaller than Q and can be handled by the induction hypothesis. Outside of the maximal subtrees, no rules are applied to ϕ and the two occurrences can be replaced with only one occurrence right away). There are the following cases:

– ϕ starts with $\neg$, with $\wedge$ and is in the antecedent, or with $\vee$ and is in the succedent. By *Step 1*, we can transform Q into a proof Q_2 in which the second rule application on all branches is done to the second occurrence of ϕ (the depth of Q_2 is at most 1 bigger than that of Q). E.g.:

$$\cfrac{\cfrac{\Gamma \vdash \phi_1, \phi_2, \phi_1, \phi_2, \Delta \Downarrow C}{\Gamma \vdash \phi_1, \phi_2, \phi_1 \vee \phi_2, \Delta \Downarrow C}\ \textsc{or-right}}{\Gamma \vdash \phi_1 \vee \phi_2, \phi_1 \vee \phi_2, \Delta \Downarrow C}\ \textsc{or-right}$$

The induction hypothesis allows to replace the subproof for the sequent $\Gamma \vdash \phi_1, \phi_2, \phi_1, \phi_2, \Delta \Downarrow C$ with a proof of $\Gamma \vdash \phi_1, \phi_2, \Delta \Downarrow D$ with $C \Rightarrow D$ that is not bigger. Simultaneously, one of the formulae $\phi_1 \vee \phi_2$ and one of the OR-RIGHT applications can be eliminated. (Similarly for the other cases.)

– ϕ starts with $\vee$ and is in the antecedent, or with $\wedge$ and is in the succedent. By *Step 1*, we can transform Q into a proof Q_2 in which the second rule application on all branches is done to the second occurrence of ϕ. E.g.:

$$\dfrac{\mathcal{A} \qquad \mathcal{B}}{\Gamma \vdash \phi_1 \wedge \phi_2, \phi_1 \wedge \phi_2, \Delta \Downarrow C \wedge D \wedge E \wedge F}$$

$$\dfrac{\dfrac{\Gamma \vdash \phi_1, \phi_1, \Delta \Downarrow C \quad \Gamma \vdash \phi_1, \phi_2, \Delta \Downarrow D}{\Gamma \vdash \phi_1, \phi_1 \wedge \phi_2, \Delta \Downarrow C \wedge D}}{\mathcal{A}}$$

$$\dfrac{\dfrac{\Gamma \vdash \phi_2, \phi_1, \Delta \Downarrow E \quad \Gamma \vdash \phi_2, \phi_2, \Delta \Downarrow F}{\Gamma \vdash \phi_2, \phi_1 \wedge \phi_2, \Delta \Downarrow E \wedge F}}{\mathcal{B}}$$

As before, the subproofs for $\Gamma \vdash \phi_1, \phi_1, \Delta \Downarrow C$ and $\Gamma \vdash \phi_2, \phi_2, \Delta \Downarrow F$ can be replaced with proofs of $\Gamma \vdash \phi_1, \Delta \Downarrow C'$ and $\Gamma \vdash \phi_2, \Delta \Downarrow F'$ with $C \Rightarrow C'$ and $F \Rightarrow F'$. Simultaneously, one of the formulae $\phi_1 \wedge \phi_2$ and one of the AND-RIGHT applications can be eliminated. The resulting constraint is $C' \wedge F'$ and has the property $C \wedge D \wedge E \wedge F \Rightarrow C' \wedge F'$. (Similarly for the other case.)

– ϕ starts with $\exists$ and is in the antecedent, or with $\forall$ and is in the succedent. By *Step 1*, we can transform Q into a proof Q_2 in which the second rule application on all branches is done to the second occurrence of ϕ. E.g.:

$$\dfrac{\dfrac{\Gamma \vdash [x/c]\phi', [x/d]\phi', \Delta \Downarrow [y/d][x/c]C}{\Gamma \vdash [x/c]\phi', \forall x.\phi', \Delta \Downarrow \forall y.[x/c]C}\ \text{ALL-RIGHT}}{\Gamma \vdash \forall x.\phi', \forall x.\phi', \Delta \Downarrow \forall x.\forall y.C}\ \text{ALL-RIGHT}$$

We can transform the subproof of $\Gamma \vdash [x/c]\phi', [x/d]\phi', \Delta \Downarrow [y/d][x/c]C$ into a proof of $\Gamma \vdash [x/c]\phi', [x/c]\phi', \Delta \Downarrow [y/c][x/c]C$ by replacing d everywhere in the proof with c. Subsequently, the subproof can be transformed into a proof of $\Gamma \vdash [x/c]\phi', \Delta \Downarrow D$ with $[y/c][x/c]C \Rightarrow D$ by the induction hypothesis. Simultaneously, one of the formulae $\forall x.\phi'$ and one of the ALL-RIGHT applications can be eliminated. Finally, it can be observed that the implication $\forall x.\forall y.C \Rightarrow \forall x.[y/x]C \Rightarrow \forall c.D$ holds.

– ϕ starts with $\forall$ and is in the antecedent, or with $\exists$ and is in the succedent, or is an equation, an inequality, a divisibility judgement or an atom $p(\bar{t})$. Because such formulae are not eliminated by any rule application, the two occurrences of ϕ can directly be replaced with only one occurrence everywhere in the proof.

Proof (Lem. 8). Suppose $\bar{a} = (a_1, \ldots, a_n)$ are the quantified variables, $c_1, \ldots, c_n$ are fresh constants and $\sigma = [a_1/c_1, \ldots, a_n/c_n]$. Let $\sigma_1, \ldots, \sigma_m$ be substitutions of $c_1, \ldots, c_n$ with integer literals that describe all solutions of $\sigma(\phi)$. By Lem. 7, there are $\mathrm{PresPred}^C_S$-proofs of the sequents $(\sigma_i(\sigma(\psi)) \vdash \quad \Downarrow C_i)_{i=1..m}$ for appropriate valid constraints $C_1, \ldots, C_m$. By Lem. 23, this implies that there are also m proofs of $(\sigma(\psi) \vdash \quad \Downarrow D_i)_{i=1..m}$ such that $\sigma_i(D_i)$ is valid for each i.

Then there is also a single $\mathrm{PresPred}^C_S$-proof $\sigma(\psi) \vdash \quad \Downarrow D$ such that $\sigma_i(D)$ is valid for each i: with the help of Lem. 24, we can first normalise each $\mathrm{PresPred}^C_S$-proof $(\sigma(\psi) \vdash \quad \Downarrow D_i)_{i=1..m}$ to a proof where the first steps are applications of

AND-$*$, OR-$*$, NOT-$*$:

$$\dfrac{\Gamma_1 \vdash \Delta_1 \Downarrow D_{i,1} \quad \cdots \quad \Gamma_k \vdash \Delta_k \Downarrow D_{i,k}}{\begin{array}{c}\vdots\\ \sigma(\psi) \vdash \quad \Downarrow D'_i\end{array}}$$

such that all formulae in $\Gamma_1, \ldots, \Gamma_k$ are atoms or start with $\forall$, all formulae in $\Delta_1, \ldots, \Delta_k$ are atoms or start with $\exists$, and $D_i \Rightarrow D'_i$ for all $i = 1, \ldots, m$. Thus, we can assume that each of the proofs $(\sigma(\psi) \vdash \ \Downarrow D'_i)_{i=1..m}$ contains a subproof for each sequent $(\Gamma_j \vdash \Delta_j)_{j=1..k}$, and that $D'_i = D_{i,1} \wedge \cdots \wedge D_{i,k}$ (recall that $\sigma(\psi)$ contains $\exists$ only in negative and $\forall$ only in positive positions). These subproofs can be combined to create one general proof for each of the sequents $(\Gamma_j \vdash \Delta_j)_{j=1..k}$, because every goal of a PresPred^C_S-proof of $\Gamma_j \vdash \Delta_j$ contains $\Gamma_j \vdash \Delta_j$ as a sub-sequent: the proofs are put together by starting with one proof, copying the second proof to all goals of the first proof, etc. If CLOSE is applied such that as many formulae as possible are selected in every goal, then for the constraint E_j of the big proof for $\Gamma_j \vdash \Delta_j \Downarrow E_j$ the implication $D_{1,j} \vee \cdots \vee D_{m,j} \Rightarrow E_j$ holds. Finally, the big proofs can be assembled further to obtain the anticipated proof of $\sigma(\psi) \vdash \ \Downarrow D$:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma_1 \vdash \Delta_1 \Downarrow E_1\end{array} \quad \cdots \quad \begin{array}{c}\vdots\\ \Gamma_k \vdash \Delta_k \Downarrow E_k\end{array}}{\begin{array}{c}\vdots\\ \sigma(\psi) \vdash \quad \Downarrow E_1 \wedge \cdots \wedge E_k\end{array}}$$

Because $\sigma_i(D'_i)$ is valid for each i, each of the constraints $\sigma_i(D_{i,1}), \ldots, \sigma_i(D_{i,k})$ is. This implies that $\sigma_i(E_j)$ is valid for all i, j, and thus also the formulae $\sigma_i(D) = \sigma_i(E_1 \wedge \cdots \wedge E_k)$ are valid.

Finally, we can prove $\exists \bar{a}.(\phi \wedge \psi)$ as follows:

$$\dfrac{\dfrac{\begin{array}{c}*\\ \vdots\\ \sigma(\phi), \sigma(\psi) \vdash \quad \Downarrow \neg\sigma(\phi) \vee D\end{array}}{\sigma(\phi \wedge \psi) \vdash \quad \Downarrow \neg\sigma(\phi) \vee D}\ \text{OR-LEFT}}{\exists \bar{a}.(\phi \wedge \psi) \vdash \quad \Downarrow \exists \bar{a}.(\neg\phi \vee [c_1/a_1, \ldots, c_n/a_n]D)}\ \text{EX-LEFT}^*$$

This proves the lemma, because $\exists \bar{a}.(\neg\phi \vee [c_1/a_1, \ldots, c_n/a_n]D)$ is a valid formula in Presburger Arithmetic.

Lemma 10 (Soundness of PresPred^C)

It is enough to check that all rules of the calculus are sound, which is trivial for most of the rules (also see Lem. 2). The interesting case is the rule OMEGA-ELIM. Assume that the lower sequent does not hold, i.e., C holds and the sequent

$$\Gamma, \{\alpha_i c - a_i \dot{\geq} 0\}_i, \{\beta_j c - b_j \dot{\leq} 0\}_j \vdash \Delta$$

is violated. This implies that the inequalities $\{\alpha_i c - a_i \dot{\geq} 0\}_i$, $\{\beta_j c - b_j \dot{\leq} 0\}_j$ hold. From the proof for Thm. 9 that is given in [10] we can conclude that then either the dark shadow conjunction $\bigwedge_{i,j} \alpha_i b_j - a_i\beta_j - (\alpha_i - 1)(\beta_j - 1) \dot{\geq} 0$ holds, or otherwise one of the splinters

$$\alpha_i c - a_i - k \doteq 0 \ \wedge \bigwedge_i \alpha_i c - a_i \dot{\geq} 0 \wedge \bigwedge_j \beta_j c - b_j \dot{\leq} 0$$

has to be satisfied (note, that this is more than what is guaranteed by the actual Thm. 9, where the splinters are existentially quantified).

Lemma 11 (Constraint completeness in exhaustive proofs)

The conditions on page 183 ensure that (1) is preserved by all rule applications: if (1) holds for all premisses of a rule application, then it also holds for the conclusion. With the help of a simple induction, this entails that (1) holds for all proof nodes. In detail:

1. It is easy to see that AND-*, OR-*, NOT-*, PRED-UNIFY, RED, DIV-LEFT, DIV-RIGHT, and SIMP preserve (1). Observe that if $U' \subseteq U$, then:

$$\forall U'.\ (\mathit{\Gamma}_p \ \vdash\ \mathit{\Delta}_p) \ \Rightarrow\ \forall U'.\ C \quad \text{entails} \quad \forall U.\ (\mathit{\Gamma}_p \ \vdash\ \mathit{\Delta}_p) \ \Rightarrow\ \forall U.\ C$$

2. We only show the proof for EX-RIGHT-D. Let U be the annotation of the conclusion, ϕ a PA formula and assume $(\mathit{\Gamma}_p \ \vdash\ [x/c]\phi, \mathit{\Delta}_p) \ \Rightarrow\ [x/c]C$. This implies:

$$\forall U.\ (\mathit{\Gamma}_p \ \vdash\ \exists x.\phi, \mathit{\Delta}_p) \ \Leftrightarrow\ \forall U.\ \exists x.\ (\mathit{\Gamma}_p \ \vdash\ \phi, \mathit{\Delta}_p) \ \Rightarrow\ \forall U.\ \exists x.C$$

3. We show the proof for ALL-RIGHT. If ϕ is a PA formula, then:

$$\begin{aligned}\forall U.\ (\mathit{\Gamma}_p \ \vdash\ \forall x.\phi, \mathit{\Delta}_p) \ &\Leftrightarrow\ \forall(U \cup \{c\}).\ (\mathit{\Gamma}_p \ \vdash\ [x/c]\phi, \mathit{\Delta}_p)\\ &\Rightarrow\ \forall(U \cup \{c\}).\ [x/c]C \ \Leftrightarrow\ \forall U.\ \forall x.C\end{aligned}$$

   Similarly, if ϕ contains uninterpreted predicates:

$$\begin{aligned}\forall U.\ (\mathit{\Gamma}_p \ \vdash\ \mathit{\Delta}_p) \ &\Leftrightarrow\ \forall(U \cup \{c\}).\ (\mathit{\Gamma}_p \ \vdash\ \mathit{\Delta}_p)\\ &\Rightarrow\ \forall(U \cup \{c\}).\ [x/c]C \ \Leftrightarrow\ \forall U.\ \forall x.C\end{aligned}$$

4. For COL-RED:

$$\begin{aligned}&\forall U.\ \big(\mathit{\Gamma}_p, \alpha c + t \doteq 0 \ \vdash\ \mathit{\Delta}_p\big)\\ &\quad\Leftrightarrow\ \forall U.\ \big(\mathit{\Gamma}_p, \exists c'.(\alpha(u + c') + t \doteq 0 \wedge c - u - c' \doteq 0) \ \vdash\ \mathit{\Delta}_p\big)\\ &\quad\Leftrightarrow\ \forall(U \cup \{c'\}).\ \big(\mathit{\Gamma}_p, \alpha(u + c') + t \doteq 0, c - u - c' \doteq 0 \ \vdash\ \mathit{\Delta}_p\big)\\ &\quad\Rightarrow\ \forall(U \cup \{c'\}).\ [x/c']C\\ &\quad\Leftrightarrow\ \forall U.\ \forall x.C\end{aligned}$$

5. Let U be the annotation of the conclusion and assume:
$$\forall U.\ (\Gamma_p, \alpha(u+c') + t \doteq 0, c - u - c' \doteq 0 \ \vdash\ \Delta_p) \ \Rightarrow\ \forall U.\ [x/c']C$$
Because $c' \notin U$ and $c - u$ does not contain any constants from U, we can substitute $c - u$ for c':
$$\forall U.\ (\Gamma_p, \alpha(u + (c-u)) + t \doteq 0, c - u - (c-u) \doteq 0 \ \vdash\ \Delta_p) \ \Rightarrow \\ \forall U.\ [x/c-u]C$$
which entails:
$$\forall U.\ (\Gamma_p, \alpha c + t \doteq 0 \ \vdash\ \Delta_p) \ \Rightarrow\ \forall U.\ [x/c-u]C$$
6. Follows directly from Thm. 9 and the fact that $c \in U$ does not occur in Γ and Δ.
7. Let U be the annotation of the conclusion ($c \in U$) and assume:
$$\forall U.\ (\Gamma_p, \alpha c - t \doteq 0 \ \vdash\ \Delta_p) \ \Rightarrow\ \forall U.\ C, \\ \forall U.\ (\Gamma_p, \alpha c - t \doteq 0 \ \vdash\ \Delta_p)$$
which directly entails (because $c \in U$ does not occur in C' and $C \Leftrightarrow [x/\alpha c]C'$ holds):
$$\forall U.\ [x/\alpha c]C' \ \Leftrightarrow\ \forall U.\ \forall x.\ (C' \vee \alpha \nmid x)$$
The last formula also holds if t is substituted for x:
$$\forall U.\ ([x/t]C' \vee \alpha \nmid t)$$
8. Assume $val_{I,\delta}(\forall U.\ \neg\phi_1 \vee \cdots \vee \neg\phi_n \vee \psi_1 \vee \cdots \vee \psi_m) = \mathit{ff}$ for some interpretation I and constant assignment δ. Let $U_2 \subseteq U$ be those U-constants that only occur in equations of the succedent $\psi_1, \ldots, \psi_m, \Delta$. We modify δ to falsify all such equations, resulting in the assignment δ': this is possible because U_2-equations describe hyperplanes in $\mathbb{Z}^{|U_2|}$, and the intersection of the complements of (finitely many) hyperplanes is non-empty.
Then it is the case that $val_{I,\delta'}(\Gamma_p, \phi_1, \ldots, \phi_n \ \vdash\ \psi_1, \ldots, \psi_m, \Delta_p) = \mathit{ff}$: formulae that do not contain U_2-constants evaluate to *tt* (antecedent) or *ff* (succedent) by assumption, and equations with U_2-constants by construction.

Lemma 14 (Termination and exhaustiveness)

Termination has to be proven on different levels:

Loop 1–2. Terminates because $<_r$ is well-founded.

As a special case, note that if any rule application derives the formula *false* (which is abbreviation for $1 \doteq 0$) in the antecedent, subsequent applications of RED will immediately replace *all* terms in the sequent with 0 and thereby cause the algorithm to terminate.

Loop 1–4. We prove termination similarly as in the appendix of [8], by considering the following mapping of sequents to triples $(c_U, c_E, |U|)$ of two multisets over $\mathbb{N} \cup \{\infty\}$ and one natural number. We call a constant c *dependent* if it occurs as the leading term of an equation $c + t \doteq 0$ in the antecedent, and *independent* otherwise. Once a constant c is dependent, the exhaustive application of RED will eliminate all occurrences of c in a sequent but the one in the defining equation $c + t \doteq 0$.

- c_U is the multiset of greatest common divisors of leading coefficients for independent U-constants:

$$\left\{\!\!\left\{ \gcd(\alpha_1, \ldots, \alpha_n) \in \mathbb{N} \cup \{\infty\} \;\middle|\; \begin{array}{c} c \in U \text{ an independent constant,} \\ \alpha_1 c + t_1 \doteq 0, \ldots, \alpha_n c + t_n \doteq 0 \\ \text{all equations in the antecedent} \\ \text{whose leading term is } c \end{array} \right\}\!\!\right\}$$

  in which we define $\gcd() = \infty$.
- c_E is the corresponding multiset for non-U-constants:

$$\left\{\!\!\left\{ \gcd(\alpha_1, \ldots, \alpha_n) \in \mathbb{N} \cup \{\infty\} \;\middle|\; \begin{array}{c} c \notin U \text{ an independent constant} \\ \text{in the sequent,} \\ \alpha_1 c + t_1 \doteq 0, \ldots, \alpha_n c + t_n \doteq 0 \\ \text{all equations in the antecedent} \\ \text{whose leading term is } c \end{array} \right\}\!\!\right\}$$

- $|U|$ is the number of U-constants.

Such triples $(c_U, c_E, |U|)$ are compared lexicographically. Multisets over $\mathbb{N} \cup \{\infty\}$ are compared using the well-founded ordering $<_m$: for elements $a_1 \leq \cdots \leq a_n$, $b_1 \leq \cdots \leq b_m$, we define:

$$\begin{aligned} &\{\!\{a_1, \ldots, a_n\}\!\} <_m \{\!\{b_1, \ldots, b_m\}\!\} \quad \text{iff} \\ &\qquad\qquad n < m \text{ or } (n = m \text{ and } (a_1, \ldots, a_n) <_{\text{lex}} (b_1, \ldots, b_m)) \end{aligned}$$

We prove termination of the loop 1–4 by showing that the triple $(c_U, c_E, |U|)$ for a sequent becomes strictly smaller each time step 3 or 4 is carried out, and does not become bigger if step 2 occurs. We only consider sequents in which the rule SIMP has been fully applied to all formulae (in other words, trailing applications of SIMP are conceptually considered as part of the other rule applications).

- RED (step 2): U does not change in this step.
  If the target formula is not an equation in the antecedent, the only relevant effect might be that constants disappear from a sequent, which does not increase the measure.
  Thus, assume that the application turns the left-hand side of an antecedent equation $s \doteq 0$ into $s + \alpha \cdot t <_r s$; after a possibly following application of SIMP, the new equation is $s' \doteq 0$:

- if $s' = 0$, then it must be the case that $s = t$, which contradicts the assumption that $\phi[s]$ is not an equation in the antecedent.
- if $s' = 1$, *false* has been derived and the strategy terminates abruptly.
- otherwise, if s and s' have the same leading term c and leading coefficients β, β', then $\beta' = \beta$ or $\beta' <_r \beta$. This implies that neither c_U nor c_E have become $<_m$-bigger.
- otherwise, if s and s' have different leading terms c, c' and leading coefficients β, β', then $c' <_r c$ and $t \doteq 0$ has the form $\gamma c + w \doteq 0$, where β is a multiple of γ.
  This implies that the c_U- or c_E-element for c has not changed (if c is dependent, there is no element at all). The c_U- or c_E-element for c' has at least not become bigger (again, it is possible that c' is dependent and there is not element).

– COL-RED-SUBST (step 3): first, note that t contains constants (otherwise, SIMP would be applicable), but does not contain U-constants (because $c \notin U$ and U-constants are $<_r$-maximal). Because no U-constants are involved, c_U stays the same. Further, the leading term of the equation $\alpha(u + c') + t \doteq 0$, after a potential application of SIMP, is not c': this could only be the case if all coefficients in t were multiples of α, which means that SIMP would have been applicable to $\alpha c + t \doteq 0$.
If the leading coefficient of the new equation $\alpha(u + c') + t \doteq 0$ is 1, then the cardinality of c_E decreases (because an independent constant disappears) and c_E becomes $<_m$-smaller.
Otherwise, there are three changes affecting c_E:
  - The constant c is independent before the rule application and dependent afterwards, which means that one element of c_E disappears. Because RED has been applied exhaustively before step 3, $\alpha c + t \doteq 0$ is the only equation in the antecedent whose leading term is c and the removed element is α.
  - The new constant c' is independent and does not occur as leading term of any equation, which means that ∞ is added as a new element to c_E.
  - The c_E-element belonging to the leading term d of the new equation $\alpha(u + c') + t \doteq 0$ (which was an independent constant before applying COL-RED-SUBST because RED was applied exhaustively) changes: suppose γ is the c_E-element belonging to d before the application of COL-RED-SUBST. Because u is chosen such that:

$$(\alpha u + t) \;=\; \min_{<_r} \{\alpha u' + t \mid u' \ \text{ a term}\}$$

    the leading coefficient γ' of $\alpha(u + c') + t \doteq 0$, after a potential application of SIMP, has to be greater than 1 (by assumption) but strictly smaller than α. Besides, γ' is also strictly smaller than γ, because RED was applied exhaustively: if $\gamma < \infty$, there has to be an equation $\gamma d + w \doteq 0$ in the antecedent that can be applied to reduce t.

Altogether, the new value of c_E is:

$$c'_E \;=\; c_E \backslash \{\!\{\alpha, \gamma\}\!\} \cup \{\!\{\infty, \gamma'\}\!\}$$

and $c'_E <_m c_E$ because of $0 < \gamma' < \alpha$ and $0 < \gamma' < \gamma$.

- COL-RED (step 3): shown in the same way as for COL-RED-SUBST, with the difference that c_U is considered instead of c_E. The condition that t contains further U-constants whose coefficient is not a multiple of α is needed to ensure that c' is not the leading term of the new equation $\alpha(u + c') + t \doteq 0$.
- DIV-CLOSE (step 4):
  - If $\alpha > 1$, then c was a independent U-constant before applying DIV-CLOSE, i.e., the cardinality of c_U becomes smaller because c is removed from U.
  - If $\alpha = 1$ and t contains further constants, then after an application of SIMP to the equation $\alpha c + t \doteq 0$ the constant c is no longer the leading term. Call the new leading term c' and its coefficient α'. The constant c' was independent before applying DIV-CLOSE. Because RED was applied exhaustively, the old c_U- or c_E-element for c' is bigger than α'. This implies that either c_U becomes $<_m$-smaller, or c_U stays the same and c_E becomes $<_m$-smaller.
  - If $\alpha = 1$ and t does not contain further constants, then neither c_U nor c_E changes, but the cardinality of U decreases.

Loop 1–5. It can first be observed that there is little interaction between the rule FM-ELIM and the rules of step 1–4 that treat equalities: step 5 is only reached once the leading coefficient of all equations in the antecedent is 1, and once no leading term of such an equation occurs in more than place in the sequent. This implies that an application of FM-ELIM never enables further applications of the rules in step 1–4. The application of FM-ELIM alone has to terminate because for any finite set of inequalities there is only a finite number of Fourier-Motzkin-inferences (respecting the ordering $<_r$).

In order to show that the application of ANTI-SYMM, AND-*, OR-*, NOT-* terminates, we use a pair of natural numbers as a measure for the complexity of a sequent. Two such pairs are compared lexicographically:

- The number of propositional connectives $\wedge$, $\vee$, $\neg$ in the sequent.
- The dimension d of the smallest affine space in $\mathbb{R}^C$ that contains all (integer) solutions of the equations in the antecedent, where C is the set of all constants occurring in a sequent.

Both FM-ELIM and the rules in step 1–4 do not apply to propositional connectives and preserve the dimension d, while the other rules of step 5 decrease the measure:

- ANTI-SYMM (step 5): again, this rule is only applied once the leading coefficient of all equations in the antecedent is 1, and once no leading term of such an equation occurs in more than place in the sequent (in particular not in the equation that is added by ANTI-SYMM). This implies that the dimension d is decreased by 1 by the new equation.
- AND-*, OR-*, NOT-* (step 5): eliminates one propositional connective.

Loop 1–6. Again, we use vectors of natural numbers that are compared lexicographically as a measure for sequents:

– The number of divisibility judgements $\alpha \mid t$ in positive positions.
– The number of divisibility judgements $\alpha \mid t$ in negative positions.
– The number of quantifiers $\forall$, $\exists$ in the sequent.
– The dimension d of the smallest affine space in $\mathbb{R}^C$ that contains all (integer) solutions of the equations in the antecedent, where C is the set of all constants occurring in a sequent.
– The number of equations in positive positions.

None of the rules in step 1–5 increase any of these features (but possibly decrease some of them), while the other rules of step 6 decrease the measure:

– SPLIT-EQ (step 6): eliminates an equation in a positive position.
– OMEGA-ELIM (step 6): an application of this rule will at first not have any influence on the complexity of a sequent (but it introduces new propositional connectives). The next rules applicable after OMEGA-ELIM, however, are the rules AND-*, OR-*, NOT-* of step 5 that split the introduced formula into its disjuncts. For each of the disjuncts, the dimension d is reduced by 1: the first of the disjuncts does no longer contain the constant c at all (the set C becomes smaller), while the other disjuncts introduce a new equation in a negative position so that the same argument as for the rule ANTI-SYMM applies.
– ALL-*, EX-* (step 6): eliminates one quantifier.
– DIV-RIGHT (step 6): eliminates a divisibility judgement in a positive position (and introduces new judgements in negative positions).
– DIV-LEFT (step 6): eliminates a divisibility judgement in a negative position.

Exhaustiveness. To prove that the resulting proof is exhaustive, annotate the proof tree with the sets U that are maintained by the strategy. The most involved point is to see that CLOSE is applied in the right way. To this end, observe that if CLOSE must not be applied according to the conditions in Sect. 5.1, then some other rule with higher priority can be applied.

Lemma 15 (Quantifier elimination)

By induction on the size of a proof, show that if a sequent $\Gamma \vdash \Delta \Downarrow C$ of the resulting exhaustive proof is annotated with U, then C does not contain any U-constants. Note, in particular, that divisibility judgements $\alpha \mid t$ (which can equivalently be expressed using existentially quantified equations) never reduce the set U. Because the introduced constant is added to U when the rules ALL-RIGHT, EX-LEFT or COL-RED are applied, this proves the lemma.

Lemma 16 (PresPred$_S^C$ subsumes PresPred^C)

We first need two further lemmas:

Lemma 25. *Suppose a PresPred$_S^C$-proof exists for the sequent $\Gamma \vdash \Delta \Downarrow C$. For some D with $C \Rightarrow D$ there is a proof of the sequent $\Gamma \vdash \Delta \Downarrow D$ in which no rule apart from* CLOSE *is applied to formulae that do not contain uninterpreted predicates (i.e., to PA formulae).*

Proof. The proof is done by induction on the size of the original proof P of $\Gamma \vdash \Delta \Downarrow C$. We can assume that the first rule application in P is different from CLOSE and is performed on a PA formula ϕ, and that no other rule application in P (apart from CLOSE) involves PA formulae. There are the following cases:

- ϕ starts with $\neg$, with $\wedge$ and is in the antecedent, or with $\vee$ and is in the succedent. Because no rules apart from CLOSE are applied to the formulae resulting from the first rule application in P, the application can simply be left out without changing the overall constraint.
- ϕ starts with $\vee$ and is in the antecedent, or with $\wedge$ and is in the succedent, i.e., the first rule application splits P into two subproofs. E.g.:

$$\dfrac{\begin{matrix}\vdots \\ \Gamma \vdash \phi_1, \Delta \Downarrow C\end{matrix} \qquad \begin{matrix}\vdots \\ \Gamma \vdash \phi_2, \Delta \Downarrow D\end{matrix}}{\Gamma \vdash \phi_1 \wedge \phi_2, \Delta \Downarrow C \wedge D} \text{ AND-RIGHT}$$

  By Lem. 24 and as in the proof of Lem. 8, we can assume that uninterpreted predicates occur in Γ, Δ only in formulae in the antecedent that start with $\forall$, in formulae in the succedent that start with $\exists$, or in literals $p(\bar{t})$: otherwise, we can turn P into a proof in which the first rule application is performed on a formula with uninterpreted predicates of different shape and consider the direct subproofs of this proof.
  Because no further rules (apart from CLOSE) are applied to ϕ_1 and ϕ_2 (or to any PA formulae), this means that there are proofs of $\Gamma \vdash \Delta \Downarrow C'$ and $\Gamma \vdash \Delta \Downarrow D'$ such that $C \Rightarrow C' \vee \phi_1$ and $D \Rightarrow D' \vee \phi_2$. Further, because of the assumption about the formulae in Γ, Δ, we know that $\Gamma \vdash \Delta$ is a subsequent of each goal in both proofs. This means that we can copy the second proof to each goal of the first proof (possibly renaming constants so that no name clashes occur). If CLOSE is in each goal applied as liberally as possible, the constraint of the resulting proof is at least as weak as $C' \vee D'$. Finally, by adding $\phi_1 \wedge \phi_2$ to all succedents in the proof, the constraint can be made as weak as $C' \vee D' \vee \phi_1 \wedge \phi_2$. Because of $C \wedge D \Rightarrow (C' \vee \phi_1) \wedge (D' \vee \phi_2) \Rightarrow C' \vee D' \vee \phi_1 \wedge \phi_2$, this concludes the case.
- ϕ starts with $\exists$ and is in the antecedent, or with $\forall$ and is in the succedent. E.g.:

$$\dfrac{\begin{matrix}\vdots \\ \Gamma \vdash [x/c]\phi', \Delta \Downarrow [x/c]C\end{matrix}}{\Gamma \vdash \forall x. \phi', \Delta \Downarrow \forall x. C} \text{ ALL-RIGHT}$$

As before, this means that there is a proof of a sequent $\Gamma \vdash \Delta \Downarrow D$ such that $[x/c]C \Rightarrow D \vee [x/c]\phi'$, whereby we can assume that c does not occur in D. By adding $\forall x.\phi'$ to all succedents of this proof, we obtain a proof of $\Gamma \vdash \forall x.\phi', \Delta \Downarrow E$ such that $D \vee \forall x.\phi' \Rightarrow E$. Altogether, this means that $\forall x.C \Rightarrow \forall x.(D \vee \phi') \Rightarrow D \vee \forall x.\phi' \Rightarrow E$.

– ϕ starts with $\forall$ and is in the antecedent, or with $\exists$ and is in the succedent. E.g.

$$\dfrac{\begin{matrix}\vdots\\ \Gamma \vdash [x/c]\phi', \exists x.\phi', \Delta \Downarrow [x/c]C\end{matrix}}{\Gamma \vdash \exists x.\phi', \Delta \Downarrow \exists x.C}\ \text{EX-RIGHT}$$

By leaving out $[x/c]\phi'$ everywhere, we obtain a proof of $\Gamma \vdash \exists x.\phi', \Delta \Downarrow D$ such that $[x/c]C \Rightarrow D \vee [x/c]\phi'$, whereby we can assume that c does not occur in D. If CLOSE is applied as liberally as possible in each goal, the implication $\exists x.\phi' \Rightarrow D$ holds, i.e., $D \Leftrightarrow D \vee \exists x.\phi'$. Altogether, this means $\exists x.C \Rightarrow \exists x.(D \vee \phi') \Rightarrow D \vee \exists x.\phi' \Rightarrow D$.

The following lemma will be used to justify application of the rules RED and SIMP:

Lemma 26. *Suppose that Γ_p, Δ_p are sets of PA formulae and s_1, s_2 are two terms or two PA atoms (equations, inequalities, or divisibility judgements) such that:*

$$\bigwedge \Gamma_p \to \bigvee \Delta_p \quad \Rightarrow \quad s_1 - s_2 \doteq 0 \qquad \textit{(in case of terms)}$$

$$\bigwedge \Gamma_p \to \bigvee \Delta_p \quad \Rightarrow \quad (s_1 \wedge s_2) \vee (\neg s_1 \wedge \neg s_2) \qquad \textit{(in case of atoms)}$$

Further, suppose a $PresPred^C_S$-proof exists for $\Gamma, \Gamma_p \vdash \phi[s_1], \Delta_p, \Delta \Downarrow C$ (we write $\phi[s_1]$ in the succedent to denote that the term or atom s_1 can occur in an arbitrary position in the sequent, in particular also in the antecedent) in which no rule apart from CLOSE is applied to formulae that do not contain uninterpreted predicates. For some D with $C \Rightarrow D$ there is a proof of $\Gamma, \Gamma_p \vdash \phi[s_2], \Delta_p, \Delta \Downarrow D$ that has the same depth as the original proof and that starts with the same rule application, and in which no rule apart from CLOSE is applied to formulae that do not contain uninterpreted predicates.

Proof. The proof is done by induction on the size of the original proof P of $\Gamma, \Gamma_p \vdash \phi[s_1], \Delta_p, \Delta \Downarrow C$. As in the proof of Lem. 24, we first show the main conjecture using proof trees with multiset sequents, and then refer to *Step 2* of the proof of Lem. 24 to carry over the result to proofs with normal sequents. Observe that none of the following transformation steps increases the depth of a proof or introduces new rule applications (other than CLOSE) to PA formulae.

We can assume that the first rule application in P involves the formula $\phi[s_1]$. Otherwise, consider the maximal subproofs of P with this property. Outside of the subproofs, $\phi[s_1]$ can simply be replaced with $\phi[s_2]$ and is unaffected by other rule applications due to the usage of multiset sequents.

If the first (and only) rule application in P is CLOSE and includes $\phi[s_1]$, the formula cannot contain uninterpreted predicates, and then neither does $\phi[s_2]$. This means that $\phi[s_1]$ can be replaced with $\phi[s_2]$ in this goal. CLOSE can be applied such that both $\phi[s_2]$ and the formulae Γ_p, Δ_p are included. Because of $\phi[s_1] \Rightarrow (\bigwedge \Gamma_p \to \bigvee \Delta_p) \to \phi[s_2]$, the resulting constraint does not get stronger.

If the first application in P is done with a rule other than CLOSE, it is always possible to do this application to $\phi[s_2]$ instead of $\phi[s_1]$ and to handle the direct subproofs using the induction hypothesis (note, that $\phi[s_1]$ and $\phi[s_2]$ have the same structure).

Proof (Lem. 16). By an induction on the size of the PresPred^C-proof P. In each step, it can be assumed that the first rule application in P is done using a rule that is not present in PresPred^C_S, and that all other rules applied in P are PresPred^C_S-rules. By Lem. 25, we can furthermore assume that no inner rule application (other than CLOSE) is done on PA formulae. The following cases are possible, depending on the first rule applied:

– ALL-LEFT-D, EX-RIGHT-D: replace the application with ALL-LEFT or EX-RIGHT, which does not make the resulting constraint stronger.
– RED, SIMP: the rule application can be left out with the help of Lem. 26.
– DIV-LEFT, DIV-RIGHT, ANTI-SYMM, FM-ELIM: because these rules replace PA formulae with equivalent formulae, they can directly be left out without strengthening constraints.
– COL-RED, e.g.:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, \alpha(u+c')+t \doteq 0, c-u-c' \doteq 0 \ \vdash\ \Delta \Downarrow [x/c']C'\end{array}}{\Gamma, \alpha c + t \doteq 0 \ \vdash\ \Delta \Downarrow \forall x.C'}\ \text{COL-RED}$$

By leaving out the two equations in the antecedent, we can create a proof of a sequent $\Gamma \ \vdash\ \Delta \Downarrow C''$ with:

$$[x/c']C' \quad \Rightarrow \quad C'' \vee \alpha(u+c')+t \not\doteq 0 \vee c-u-c' \not\doteq 0$$

whereby it can be assumed that c' does not occur in C''. Then, by adding $\alpha c + t \doteq 0$ to all antecedents, a proof of $\Gamma, \alpha c + t \doteq 0 \ \vdash\ \Delta \Downarrow D$ can be derived such that $C'' \vee \alpha c + t \not\doteq 0 \Rightarrow D$. Altogether, this means:

$$\begin{aligned}\forall x.C' \quad \Rightarrow \quad \forall c'.[x/c']C' \quad \Rightarrow \quad & \forall c'.(C'' \vee \alpha(u+c')+t \not\doteq 0 \vee c-u-c' \not\doteq 0)\\ \Rightarrow \quad & C'' \vee \alpha c + t \not\doteq 0 \quad \Rightarrow \quad D\end{aligned}$$

– COL-RED-SUBST: analogously.
– DIV-CLOSE, e.g.:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, \alpha c - t \doteq 0 \ \vdash\ \Delta \Downarrow C'\end{array}}{\Gamma, \alpha c - t \doteq 0 \ \vdash\ \Delta \Downarrow [x/t]C'' \vee \alpha \nmid t}\ \text{DIV-CLOSE}$$

If CLOSE is in the proof always applied as liberally as possible, such that also the equation $\alpha c - t \doteq 0$ is selected, then $\alpha c - t \not\doteq 0 \Rightarrow C'$, i.e., the equivalence $C' \Leftrightarrow C' \vee \alpha c - t \not\doteq 0$ holds. Because of $C' \Leftrightarrow [x/\alpha c]C''$, this means $C' \Leftrightarrow [x/t]C'' \vee \alpha c - t \not\doteq 0$. Finally, because of $\alpha c - t \doteq 0 \Rightarrow \alpha \mid t$:

$$[x/t]C'' \vee \alpha \nmid t \quad \Rightarrow \quad [x/t]C'' \vee \alpha c - t \not\doteq 0 \quad \Rightarrow \quad C'$$

This means that the constraint of the proof does not become stronger if the application of DIV-CLOSE is left out.

– SPLIT-EQ, e.g.:

$$\dfrac{\begin{matrix}\vdots & & \vdots \\ \Gamma \vdash t \dot{\leq} 0, \Delta \Downarrow C' & & \Gamma \vdash t \dot{\geq} 0, \Delta \Downarrow D'\end{matrix}}{\Gamma \vdash t \doteq 0, \Delta \Downarrow C' \wedge D'} \text{ SPLIT-EQ}$$

We can modify the proof to get a similar one without the application of SPLIT-EQ:

$$\dfrac{\begin{matrix}\vdots & & \vdots \\ \Gamma \vdash t \dot{\leq} 0, \Delta \Downarrow C' & & \Gamma \vdash t \dot{\geq} 0, \Delta \Downarrow D'\end{matrix}}{\Gamma \vdash t \dot{\leq} 0 \wedge t \dot{\geq} 0, \Delta \Downarrow C' \wedge D'} \text{ AND-RIGHT}$$

By Lem. 25, this can be turned into a proof of $\Gamma \vdash t \dot{\leq} 0 \wedge t \dot{\geq} 0, \Delta \Downarrow E$ in which no rules other than CLOSE are applied to PA formulae, such that the implication $C' \wedge D' \Rightarrow E$ holds. Finally, $t \dot{\leq} 0 \wedge t \dot{\geq} 0$ can be replaced with the (equivalent) equation $t \doteq 0$ everywhere in the proof, which leads to a proof of $\Gamma \vdash t \doteq 0, \Delta \Downarrow E'$ with $E' \Leftrightarrow E$.

– OMEGA-ELIM, e.g.:

$$\dfrac{\begin{matrix}\vdots \\ \Gamma, \phi(c) \vdash \Delta \Downarrow C\end{matrix}}{\Gamma, \{\alpha_i c - a_i \dot{\geq} 0\}_i, \{\beta_j c - b_j \dot{\leq} 0\}_j \vdash \Delta \Downarrow C} \text{ OMEGA-ELIM}$$

The rule application can simply be left out because of:

$$\bigwedge_i \alpha_i c - a_i \dot{\geq} 0 \wedge \bigwedge_j \beta_j c - b_j \dot{\leq} 0 \quad \Rightarrow \quad \phi(c)$$

This implication follows from the proof for Thm. 9 that is given in [10] (note, that this is more than what is guaranteed by the actual Thm. 9, where the splinters are existentially quantified).

Lemma 17 (Fair proof construction)

We call the PresPred^C_S-proof P and the fair PresPred^C-proof Q. By Lem. 25, we can assume that the only rule that is applied to PA formulae in P is CLOSE.

For the following induction, we also make the assumption that the rule PRED-UNIFY is in Q applied in the same fair manner as the rules in Fig. 1, i.e., it is eventually applied infinitely often to all complementary predicate literals (or their successors). Given any fair PresPred^C-proof, it is possible to insert further applications of PRED-UNIFY to achieve this property without changing the constraints generated by the proof (the constraints stay equivalent). Namely, assume that PRED-UNIFY is applied at some point in the proof:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, p(s_1,\ldots,s_n) \ \vdash\ p(t_1,\ldots,t_n), \bigwedge_i s_i - t_i \doteq 0, \Delta \ \Downarrow C\end{array}}{\Gamma, p(s_1,\ldots,s_n) \ \vdash\ p(t_1,\ldots,t_n), \Delta \ \Downarrow C}\ \text{PRED-UNIFY}$$

Let $\Gamma_p \subseteq \Gamma$ and $\Delta_p \subseteq \Delta \cup \{\bigwedge_i s_i - t_i \doteq 0\}$ be the sets of PA formulae in the premiss that do not contain $\exists$ in positive positions or $\forall$ in negative positions. It is obviously the case that an immediate second application of PRED-UNIFY is unnecessary because of:

$$\bigwedge_i s_i - t_i \doteq 0 \quad\Rightarrow\quad \bigwedge \Gamma_p \rightarrow \bigvee \Delta_p \tag{3}$$

i.e., the conjunction introduced by a second application is subsumed by the formulae already present in the sequent. By a simple induction on the size of a PresPred^C-proof, it can be shown that property (3) is preserved when applying arbitrary PresPred^C-rules (including RED or SIMP to the complementary literals $p(s_1,\ldots,s_n)$, $p(t_1,\ldots,t_n)$).

We perform Noetherian induction on the set of all possible pairs (P, Q), where P is a PresPred^C_S-proof for the sequent $\Gamma \ \vdash\ \Delta \ \Downarrow C$ in which the only rule that is applied to PA formulae is CLOSE, and Q is a fair PresPred^C-proof of $\Gamma \ \vdash\ \Delta \ \Downarrow ?$ (fair also concerning PRED-UNIFY in the way described above). The ordering is the lexicographic order on the pair (d_P, n), where

- d_P is the length of the longest branch in P (the depth of P), and
- n is the maximum number of rule applications that happen on a branch of Q before the first rule application in P is done on the branch. Because of fairness, the first rule application in P is eventually performed on all Q-branches, although possibly on successors of the involved formulae. By König's lemma, the maximum number of other rule applications before this happens is finite. In case the first rule application in P is CLOSE, we define $n = 0$.

The induction formula is:

> Suppose that the root of Q is annotated with U. Then Q generates a constraint D with $\forall U.C \Rightarrow \forall U.D$.

There are a number of induction step cases. In all of them, we assume that the constants introduced by EX-*, ALL-* are renamed when necessary to avoid collisions. Further, we make use of the fact that also all subproofs of Q are fair proofs (also concerning PRED-UNIFY).

- The first rule application in P is CLOSE. We can then simply prune Q and apply CLOSE to the same formulae as in P. In all of the following cases, it is therefore assumed that P does not start with CLOSE (which implies, because of fairness, that also Q does not start with CLOSE).
- P and Q start with the same rule application to the same formula(e). In case the rule is EX-* or ALL-*, we can ensure through renaming that the same constant is introduced. Then, we can apply the induction hypothesis to the direct subtrees of P and Q. There are the following cases, depending on the first rule applied:
  - AND-LEFT, OR-RIGHT, NOT-*, PRED-UNIFY: by the induction hypothesis, we know $\forall U'.C \Rightarrow \forall U'.D$ for the constraints C, D and annotation U' of the subtree roots. Because of $U' \subseteq U$, this entails $\forall U.\ C \Rightarrow \forall U.\ D$.
  - AND-RIGHT, OR-LEFT: by the induction hypothesis, $\forall U'.C' \Rightarrow \forall U'.D'$ and $\forall U''.C'' \Rightarrow \forall U''.D''$ for the constraints and annotations of the subtree roots. Because of $U' \subseteq U$ and $U'' \subseteq U$, this entails:
  $$\forall U.\ (C' \wedge C'') \quad \Rightarrow \quad \forall U.\ (D' \wedge D'')$$
  - ALL-LEFT, EX-RIGHT: by the induction hypothesis, $[x/c]C' \Rightarrow [x/c]D'$ for the constraints of the subtrees (which are annotated with the empty set), which entails $\forall U.\exists x.\ C' \Rightarrow \forall U.\exists x.\ D'$.
  - ALL-RIGHT, EX-LEFT: we know that $\forall U'.[x/c]C' \Rightarrow \forall U'.[x/c]D'$ for the constraints and annotations of the subtrees. Because of $U' \subseteq U \cup \{c\}$, this entails: $\forall U.\forall x.\ C' \Rightarrow \forall U.\forall x.\ D'$.

  In all of the following cases, we therefore assume that P and Q start with different rule applications.
- The first rule application in Q is AND-*, OR-*, NOT-*, EX-LEFT, ALL-RIGHT to a formula ϕ. By Lem. 24, we can transform P into a proof P' of some sequent $\Gamma \vdash \Delta \Downarrow D$ with $C \Rightarrow D$ that starts with the same rule application as Q. The depth of P' is at most one bigger than the depth of P, and the first rule application of P is the second rule application on all branches in P'. Furthermore, the only rule in P' that is applied to PA formulae is CLOSE, possibly apart from the first rule application in P'. We can then apply the induction hypothesis to the direct subtrees of P' and Q.
- The first rule application in Q is PRED-UNIFY, ALL-LEFT, or EX-RIGHT. This rule application can be inserted as first rule application in P, adding the resulting formula to all sequents, which leads to a proof P' whose depth is one bigger than that of P and that has the same or a weaker constraint as P. The first rule application of P is the second rule application in P'. Then, the induction hypothesis can be applied to the direct subtrees of P' and Q.
- The first rule application in Q is EX-RIGHT-D or ALL-LEFT-D. E.g.:

$$\dfrac{\begin{matrix}\vdots \\ \Gamma \vdash [x/c]\phi, \Delta \Downarrow ?\end{matrix}}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow ?}\ \text{EX-RIGHT-D}$$

Because the proof P (of the sequent $\Gamma \vdash \exists x.\phi, \Delta \Downarrow C$) does not contain any rule applications to $\exists x.\phi$ apart from CLOSE (the formula does not contain uninterpreted predicates), this means that $\exists x.\phi$ can be left out everywhere in P, leading to a similar proof P' of a sequent $\Gamma \vdash \Delta \Downarrow C'$ with $C \Rightarrow C' \vee \exists x.\phi$ (as in the proof of Lem. 25). It is then possible to add the formula $[x/c]\phi$ to all succedents in P', resulting in a proof P'' of a sequent $\Gamma \vdash [x/c]\phi, \Delta \Downarrow C''$ (if necessary, one has to ensure by renaming that c does not occur in P'). If CLOSE is applied as liberally as possible in P'', the implication $C' \vee [x/c]\phi \Rightarrow C''$ holds. Finally, a proof P''' can be obtained from P'' by inserting EX-RIGHT-D as first rule application:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash [x/c]\phi, \Delta \Downarrow C''\end{array}}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists c.C''}\ \text{EX-RIGHT-D}$$

The depth of P''' is one bigger than the depth of P, and the first rule application in P is the second rule application in P'''. Thus, applying the induction hypothesis to the direct subproofs of P''' and Q, we know that $C'' \Rightarrow D'$ (the annotation of the root of the direct subproof of Q is the empty set). This entails that:

$$C \;\Rightarrow\; C' \vee \exists x.\phi \;\Rightarrow\; \exists x.(C' \vee \phi) \;\Rightarrow\; \exists c.(C' \vee [x/c]\phi) \;\Rightarrow\; \exists c.C'' \;\Rightarrow\; \exists c.D'$$

and therefore $\forall U.C \Rightarrow \forall U.\exists c.D'$.

- If the first rule application in Q is COL-RED, DIV-LEFT, DIV-RIGHT, SPLIT-EQ, ANTI-SYMM, or FM-ELIM, the same technique as in the previous case can be used.
- If the first rule application in Q is RED or SIMP, we can insert the same application as first step in P with the help of Lem. 26. Then, the induction hypothesis can be applied to the direct subproofs of the proofs.
- If the first rule application in Q is COL-RED-SUBST, we can first turn P into a proof P' of a sequent $\Gamma, \alpha(u + c') + t \doteq 0, c - u - c' \doteq 0 \vdash \Delta \Downarrow C'$ by replacing the original equation $\alpha c + t \doteq 0$ (if necessary, it has to be ensured by bound renaming that c' does not occur in P). If CLOSE is applied as liberally as possible in P', it holds that $C \Rightarrow C' \vee \alpha c + t \not\doteq 0$ and $\alpha(u + c') + t \not\doteq 0 \vee c - u - c' \not\doteq 0 \Rightarrow C'$. We can then obtain a proof P'' of $\Gamma, \alpha c + t \doteq 0 \vdash \Delta \Downarrow [c'/c - u]C'$ by adding COL-RED as first rule application in P'. Considering the constraints, we have:

$$\begin{aligned} [c'/c - u](\alpha(u + c') + t \not\doteq 0 \vee c - u - c' \not\doteq 0) \;&\Leftrightarrow\; \alpha c + t \not\doteq 0 \\ &\Rightarrow\; [c'/c - u]C' \end{aligned}$$

Because C and t do not contain c', this altogether means that the implication $C \Rightarrow [c'/c - u](C' \vee \alpha c + t \not\doteq 0) \Rightarrow [c'/c - u]C'$ holds. Furthermore, applying the induction hypothesis to the direct subproofs of P'' and Q, we know that $\forall U'.C' \Rightarrow \forall U'.D'$ holds for the constraints and annotations of the subproofs. Because $c, c' \notin U'$ and u does not contain any constants from U, then also:

$$\forall U'.\ [c'/c - u]C' \quad \Rightarrow \quad \forall U'.\ [c'/c - u]D'$$

Finally, because of $U' \subseteq U$:

$$\forall U.C \quad \Rightarrow \quad \forall U.\ [c'/c - u]C' \quad \Rightarrow \quad \forall U.\ [c'/c - u]D'$$

– If the first rule application in Q is DIV-CLOSE, it is the case that $c \in U$ and we can simple insert DIV-CLOSE as first rule application in P, resulting in a proof P'. By the induction hypothesis, $\forall U'.C \Rightarrow \forall U'.D$ for the constraints and annotation of the direct subproofs, and because of $U' \subseteq U$ also $\forall U.C \Rightarrow \forall U.D$. Let D' be a formula with $D \Leftrightarrow [x/\alpha c]D'$ that does not contain c. Then:

$$\begin{aligned} \forall U.C \quad \Rightarrow \quad \forall U.D \quad \Rightarrow \quad \forall U.[x/\alpha c]D' \quad &\Rightarrow \quad \forall U.\forall x.(D' \vee \alpha \nmid x) \\ &\Rightarrow \quad \forall U.([x/t]D' \vee \alpha \nmid t) \end{aligned}$$

– The first rule application in Q is OMEGA-ELIM, which means that $c \in U$:

$$\frac{\begin{array}{c}\vdots \\ \Gamma, \phi(c) \ \vdash \ \Delta \Downarrow C\end{array}}{\Gamma, \{\alpha_i c - a_i \dot{\geq} 0\}_i, \{\beta_j c - b_j \dot{\leq} 0\}_j \ \vdash \ \Delta \Downarrow C} \text{ OMEGA-ELIM}$$

Because P does not contain any rule applications to the eliminated inequalities (other than CLOSE), these formulae can be left out everywhere, leading to a proof P' of the sequent $\Gamma \ \vdash \ \Delta \Downarrow C'$ with:

$$C \quad \Rightarrow \quad C' \vee \neg\Big(\bigwedge_i \alpha_i c - a_i \dot{\geq} 0 \wedge \bigwedge_j \beta_j c - b_j \dot{\leq} 0\Big)$$

Because Γ, Δ do not contain c, we can also assume that c does not occur in C'. Next, we can add the formula $\phi(c)$ to all antecedents, which yields a proof P'' of $\Gamma, \phi(c) \ \vdash \ \Delta \Downarrow C''$. If CLOSE is applied as liberally as possible in P'', the implication $C' \vee \neg\phi(c) \Rightarrow C''$ holds. Finally, OMEGA-ELIM can be inserted as first rule application in P'', which results in the proof P'''. The induction hypothesis can be applied to the direct subproofs of P''' and Q, which means that $\forall U'.C'' \Rightarrow \forall U'.D$ for the constraints and annotation of the subproofs. Because of $U' \subseteq U$, then also $\forall U.C'' \Rightarrow \forall U.D$. Furthermore:

$$\begin{aligned} \forall c.C \quad &\Rightarrow \quad \forall c.\Big(C' \vee \neg\Big(\bigwedge_i \alpha_i c - a_i \dot{\geq} 0 \wedge \bigwedge_j \beta_j c - b_j \dot{\leq} 0\Big)\Big) \\ &\Rightarrow \quad C' \vee \neg\exists c.\Big(\bigwedge_i \alpha_i c - a_i \dot{\geq} 0 \wedge \bigwedge_j \beta_j c - b_j \dot{\leq} 0\Big) \\ &\overset{(*)}{\Rightarrow} \quad C' \vee \neg\exists c.\phi(c) \\ &\Rightarrow \quad \forall c.(C' \vee \neg\phi(c)) \\ &\Rightarrow \quad \forall c.C'' \end{aligned}$$

where $(*)$ makes use of Thm. 9. Altogether, this entails $\forall U.C \Rightarrow \forall U.D$.

Lemma 19 (Shielded Constraints)

We first need a further lemma:

Lemma 27. *If x is a variable, $\phi_1, \dots, \phi_m$ are formulae in which x does not occur, and $\psi_0[x], \dots, \psi_m[x]$ are arbitrary formulae, then the following equivalence holds:*

$$\forall x.\Big(\psi_0[x] \vee \bigvee_{i=1}^{m} (\psi_i[x] \wedge \phi_i)\Big) \quad \Leftrightarrow \quad \bigvee_{S \subseteq \{1,\dots,m\}} \Big(\bigwedge_{i \in S} \phi_i \wedge \forall x.\Big(\psi_0[x] \vee \bigvee_{i \in S} \psi_i[x]\Big)\Big)$$

Proof. By induction on m. The case $m = 0$ is clear, and the step case $m \to m+1$ as follows:

$$\begin{aligned}
& \forall x.\Big(\psi_0[x] \vee \bigvee_{i=1}^{m+1} (\psi_i[x] \wedge \phi_i)\Big) \\
\Leftrightarrow\ & \forall x.\Big((\psi_0[x] \vee \psi_{m+1}[x] \wedge \phi_{m+1}) \vee \bigvee_{i=1}^{m} (\psi_i[x] \wedge \phi_i)\Big) \\
\overset{\text{(IH)}}{\Leftrightarrow}\ & \bigvee_{S \subseteq \{1,\dots,m\}} \Big(\bigwedge_{i \in S} \phi_i \wedge \forall x.\Big(\psi_0[x] \vee \psi_{m+1}[x] \wedge \phi_{m+1} \vee \bigvee_{i \in S} \psi_i[x]\Big)\Big) \\
\Leftrightarrow\ & \bigvee_{S \subseteq \{1,\dots,m\}} \left(\bigwedge_{i \in S} \phi_i \wedge \begin{pmatrix} \forall x.(\psi_0[x] \vee \psi_{m+1}[x] \vee \bigvee_{i \in S} \psi_i[x]) \\ \wedge\ \forall x.(\psi_0[x] \vee \phi_{m+1} \vee \bigvee_{i \in S} \psi_i[x]) \end{pmatrix}\right) \\
\overset{(*)}{\Leftrightarrow}\ & \bigvee_{S \subseteq \{1,\dots,m\}} \left(\bigwedge_{i \in S} \phi_i \wedge \begin{pmatrix} \phi_{m+1} \wedge \forall x.(\psi_0[x] \vee \psi_{m+1}[x] \vee \bigvee_{i \in S} \psi_i[x]) \\ \vee\ \forall x.(\psi_0[x] \vee \bigvee_{i \in S} \psi_i[x]) \end{pmatrix}\right) \\
\Leftrightarrow\ & \bigvee_{S \subseteq \{1,\dots,m+1\}} \Big(\bigwedge_{i \in S} \phi_i \wedge \forall x.\Big(\psi_0[x] \vee \bigvee_{i \in S} \psi_i[x]\Big)\Big)
\end{aligned}$$

(*) holds because of:

$$\begin{aligned}
& \forall x.(a[x] \vee b[x]) \wedge \forall x.(a[x] \vee c) \\
\Leftrightarrow\ & \forall x.(a[x] \vee b[x]) \wedge (c \vee \forall x.a[x]) \\
\Leftrightarrow\ & (\forall x.(a[x] \vee b[x]) \wedge c) \vee (\forall x.(a[x] \vee b[x]) \wedge \forall x.a[x]) \\
\Leftrightarrow\ & (\forall x.(a[x] \vee b[x]) \wedge c) \vee \forall x.a[x]
\end{aligned}$$

Proof (Lem. 19). We show the conjecture by an induction over the subtrees of P. In the proof leaves, the conjecture coincides with the assumption how CLOSE is applied in P_1, P_2. Otherwise, pick a subproof R (and the corresponding subproofs R_1, R_2 of P_1, P_2) and assume that the conjecture holds for the direct subproofs of R. There are the following cases, depending on the constraint transformation that is performed by the rule applied in the root of R:

- The constraint is not changed (rules AND-LEFT, etc.): trivial
- Conjunction of constraints (rules AND-RIGHT, etc.): Let D^1, D^2 be the constraints of the direct subproofs of R_1. The constraints of the direct subproofs of R_2 are equivalent to $D^1 \vee \bigvee_{i=1}^{n_1} \phi_i^1$ and $D^2 \vee \bigvee_{i=1}^{n_2} \phi_i^2$, where each ϕ_i^1, ϕ_i^2 is shielded by Q. Then the constraint of R_1 is $D^1 \wedge D^2$ and the constraint of R_2 is equivalent to:
$$\begin{aligned} &\big(D^1 \vee \bigvee_{i=1}^{n_1} \phi_i^1\big) \wedge \big(D^2 \vee \bigvee_{i=1}^{n_2} \phi_i^2\big) \\ \Leftrightarrow\ &(D^1 \wedge D^2) \vee \bigvee_{i=1}^{n_1} (\phi_i^1 \wedge D^2) \vee \bigvee_{i=1}^{n_2} (\phi_i^2 \wedge D^1) \vee \bigvee_{i=1}^{n_1} \bigvee_{j=1}^{n_2} (\phi_i^1 \wedge \phi_j^2) \end{aligned}$$
- The rule COL-RED-SUBST applies a substitution to a constraint: let $[x/c']D_1$ be the constraint of the direct subproof of R_1 and $[x/c-u]D_1$ the constraint of R_1. The constraint $[x/c']D_2$ of the direct subproof of R_2 is then equivalent to $[x/c']D_1 \vee \bigvee_{i=1}^{n} \phi_i$, where each ϕ_i is shielded by Q, and the constraint of R_2 is equivalent to $[x/c-u]D_1 \vee \bigvee_{i=1}^{n} [c'/c-u]\phi_i$.
  (i) $[x/c']D_1$ does not contain Q_c-constants. If $[x/c-u]D_1$ contains Q_c-constants, then also $c-u$ does and x occurs free in D_1. Because of the definition of free constant sets, then also $c' \in Q_c$ and then $[x/c']D_1$ contains Q_c-constants, which is a contradiction.
  (ii) We can assume that each ϕ_i has the form $\beta_i e_i + t_i \doteq 0 \wedge \psi_i$ with $e_i \in Q$, such that $d \prec_P e_i$ for all constants d in t_i. Due to the definition of $\prec_P$ we have $e_i \prec_P c'$ and thus $e_i \neq c'$ and c' does not occur in t_i. This implies that $[c'/c-u]\phi_i$ is shielded by Q:
$$[c'/c-u]\phi_i \quad \Leftrightarrow \quad \beta_i e_i + t_i \doteq 0 \wedge [c'/c-u]\psi_i$$
- The rule DIV-CLOSE' is applied: let $D_1 \Leftrightarrow [x/\alpha c']D_1'$ be the constraint of the direct subproof of R_1 and $[x/t]D_1' \vee \alpha \nmid t$ the constraint of R_1. Because D_1 does not contain any Q_c-constants, we can assume that D_1' does neither. The constraint of the direct subproof of R_2 is equivalent to $[x/\alpha c']D_1' \vee \bigvee_{i=1}^{n} \phi_i$, where each ϕ_i is shielded by Q.
  (i) Because t does not contain Q_c-constants, neither does $[x/t]D_1' \vee \alpha \nmid t$.
  (ii) We can assume that each ϕ_i has the form $\beta_i e_i + t_i \doteq 0 \wedge \psi_i$ with $e_i \in Q$. As for COL-RED-SUBST, it follows that c' does not occur in $\beta_i e_i + t_i$. Assume that $\psi_i \Leftrightarrow [x/\alpha c']\psi_i'$, then the formula $\beta_i e_i + t_i \doteq 0 \wedge [x/t]\psi_i'$ is shielded by Q. Altogether, the constraint of R_2 is equivalent to:
$$[x/t]D_1' \vee \alpha \nmid t \vee \bigvee_{i=1}^{n} (\beta_i e_i + t_i \doteq 0 \wedge [x/t]\psi_i')$$
- Existential quantification of constraints (rules EX-RIGHT, etc.): let $[x/c]D_1$ be the constraint of the direct subproof of R_1 and $\exists x.D_1$ the constraint of R_1. The constraint of the direct subproof of R_2 is equivalent to $[x/c]D_1 \vee \bigvee_{i=1}^{n} \phi_i$, where each ϕ_i is shielded by Q.

(i) Because $[x/c]D_1$ does not contain Q_c-constants, neither does $\exists x.D_1$.

(ii) The constraint of R_2 is equivalent to:

$$\exists x.\Big(D_1 \vee \bigvee_{i=1}^{n} [c/x]\phi_i\Big) \;\Leftrightarrow\; \exists x.D_1 \vee \bigvee_{i=1}^{n} \exists c.\phi_i$$

We can assume that each ϕ_i has the form $\beta_i e_i + t_i \doteq 0 \wedge \psi_i$ with $e_i \in Q$. As for COL-RED-SUBST, it follows that c does not occur in $\beta_i e_i + t_i$, and thus $\exists c.\phi_i \Leftrightarrow \beta_i e_i + t_i \doteq 0 \wedge \exists c.\psi_i$ is shielded by Q. By renaming it can be achieved that no illegal constants occur in $\exists c.\psi_i$.

– Universal quantification of constraints (rules ALL-RIGHT, etc.): let $[x/c]D_1$ be the constraint of the direct subproof of R_1 and $\forall x.D_1$ the constraint of R_1. The constraint of the direct subproof of R_2 is equivalent to $[x/c]D_1 \vee \bigvee_{i=1}^{n} \phi_i$, where each ϕ_i is shielded by Q.

(i) As for existential quantification.

(ii) We can assume that each ϕ_i has the form $t_i \doteq 0 \wedge \psi_i$, where $t_i \doteq 0$ is the shielding equation. Wlog., assume that $t_1, \ldots, t_k$ are the terms that contain c with a non-zero coefficient, while c does not occur in $t_{k+1}, \ldots, t_n$. This implies that c shields the formulae $\phi_1, \ldots, \phi_k$.

- If $c \notin Q$, it has to be the case that $k = 0$, as for COL-RED-SUBST. With the help of Lem. 27, we can rewrite the constraint of R_2 as follows:

$$\begin{aligned}
&\forall c.\Big([x/c]D_1 \vee \bigvee_{i=1}^{n} (t_i \doteq 0 \wedge \psi_i)\Big)\\
\Leftrightarrow\;& \bigvee_{S \subseteq \{1,\ldots,n\}} \Big(\bigwedge_{i\in S} t_i \doteq 0 \wedge \forall c.\Big([x/c]D_1 \vee \bigvee_{i\in S} \psi_u\Big)\Big)\\
\Leftrightarrow\;& \forall x.D_1 \vee \bigvee_{\substack{S \subseteq \{1,\ldots,n\}\\ S\neq\emptyset}} \Big(\bigwedge_{i\in S} t_i \doteq 0 \wedge \forall c.\Big([x/c]D_1 \vee \bigvee_{i\in S} \psi_u\Big)\Big)
\end{aligned}$$

- If $c \in Q$, then D_1 does not contain x by the induction hypothesis. We can again use Lem. 27 as follows:

$$\begin{aligned}
&\forall c.\Big(\underbrace{D_1 \vee \bigvee_{i=1}^{k} (t_i \doteq 0 \wedge \psi_i)}_{\psi_0[c]} \vee \bigvee_{i=k+1}^{n} (t_i \doteq 0 \wedge \psi_i)\Big)\\
\Leftrightarrow\;& \bigvee_{S \subseteq \{k+1,\ldots,n\}} \Big(\bigwedge_{i\in S} t_i \doteq 0 \wedge \forall c.\Big(\psi_0[c] \vee \bigvee_{i\in S} \psi_u\Big)\Big)\\
\Leftrightarrow\;& \forall c.\psi_0[c] \vee \bigvee_{\substack{S \subseteq \{k+1,\ldots,n\}\\ S\neq\emptyset}} \Big(\bigwedge_{i\in S} t_i \doteq 0 \wedge \forall c.\Big(\psi_0[c] \vee \bigvee_{i\in S} \psi_u\Big)\Big)
\end{aligned}$$

The formula $\forall c.\psi_0[c]$ can be simplified because all but the first disjunct are shielded by c:

$$\begin{aligned}\forall c.\Big(D_1 \vee \bigvee_{i=1}^{k} (t_i \doteq 0 \wedge \psi_i)\Big) &\Leftrightarrow D_1 \vee \forall c. \bigvee_{i=1}^{k} (t_i \doteq 0 \wedge \psi_i) \\ &\Leftrightarrow D_1 \Leftrightarrow \forall x.D_1\end{aligned}$$

In both cases, renaming can be used afterwards to eliminate c.