Karlsruhe Institute of Technology

Reconstructing Z3 Proofs With KeY

Master's Thesis by

Wolfram Pfeifer

at the KIT Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Dr. Mattias Ulbrich
Second advisor:	M. Sc. Jonas Schiffl

16 July 2020 – 15 January 2021

Karlsruher Institut für Technologie
KIT-Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Acknowledgements

I want to thank my advisors Dr. Mattias Ulbrich and Jonas Schiffl for their support in conducting this work and for the time they have dedicated to discussing solutions and ideas in weekly video meetings. I would also like to express my gratitude to Prof. Dr. Bernhard Beckert for the long and fruitful discussions we had about this work.

I hereby declare that the work presented in this thesis is entirely my own. I confirm that I specified all employed auxiliary resources and clearly acknowledged anything taken verbatim or with changes from other sources. I further declare that I prepared this thesis in accordance with the rules for safeguarding good scientific practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, 15 January 2021

. . . .

(Wolfram Pfeifer)

Abstract

KeY is a tool for formal verification of specified properties of Java programs. It works by generating proof obligations from formal specification and source code, and transforming it into a set of first-order formulas afterwards. Due to the undecidability of first-order logic it is a challenging problem to find a proof for this set of formulas.

Modern SMT solvers, such as Z3, provide highly optimized algorithms for exactly this purpose. Therefore, translating (sub-)problems to SMT solvers has long been possible from KeY. This approach resulted in a partial proof in KeY complemented by (possibly multiple) SMT results. Since Z3 is able to produce proofs for its result, it is possible to improve upon this: This thesis presents a replay technique that allows for reconstruction of the proofs generated by Z3, such that the result is a closed proof in KeY and the SMT results can be dropped. Systematic as well as engineering challenges are identified and solutions for them presented.

A prototypical implementation of the replay technique in KeY is provided. In the evaluation part of this work, its capabilities as well as future potential is shown.

Zusammenfassung

KeY dient zur formalen Verifikation spezifizierter Eigenschaften von Java-Programmen. Dafür werden aus der formalen Spezifikation sowie dem Programm-Code Beweisverpflichtungen generiert. Diese werden dann Schritt für Schritt in eine Menge von Formeln der Prädikatenlogik erster Stufe überführt. Da diese allerdings unentscheidbar ist, ist es eine große Herausforderung, einen Beweis für diese Formelmenge zu finden.

Moderne SMT-Solver, wie zum Beispiel Z3, sind genau auf diesen Anwendungszweck hin optimiert. Daher ist schon lange in KeY die Möglichkeit eingebaut, (Teil-)Probleme für SMT-Solver zu übersetzen. Das Ergebnis bei diesem Vorgehen ist ein partieller Beweis in KeY, der von (möglicherweise mehreren) SMT-Antworten komplettiert wird. Da Z3 aber auch Beweise für seine Antworten liefern kann, gibt es hier Verbesserungspotential: In dieser Thesis wird eine Technik zum Nachspielen der Z3 Beweise in KeY vorgestellt, sodass man als Ergebnis einen geschlossenen Beweis in KeY erhält und die SMT-Antworten verworfen werden können. Herausforderungen sowohl systematischer als auch technischer Natur werden identifiziert and Lösungen dafür vogestellt.

Schließlich wird auch eine prototypische Implementierung der Technik zum Nachspielen der Beweise zur Verfügung gestellt. Im Evaluations-Teil der Arbeit wird die Leistungsfähigkeit dieser Implementierung sowie die zukünfigen Möglichkeiten erörtert.

Contents

List of Figures xi

1 Introduction 1
1.1 Motivation 1
1.2 Contributions 2
1.3 Related Work 3
1.4 Outline 3

2 Preliminaries 5
2.1 Introduction to the KeY System 5
2.2 Introduction to Z3 11
2.2.1 Proof Production 13

3 The Replay Technique 19
3.1 Overview 19
3.2 Clarifying Notions 19
3.3 Challenges 21
3.3.1 Proof Directions 21
3.3.2 Replacing Equisatisfiability 22
3.3.3 Skolemization 22
3.3.4 Translating the Type Hierarchy 25
3.3.5 Structurally Different Terms 28
3.4 Additional Rules for KeY as Abbreviations 29
3.5 Translating Z3's Proof Rules 30
3.5.1 Rules Replayed Manually 30
3.5.2 Rules Replayed With Automatic Proof Search in KeY 37
3.5.3 Other Rules 39
3.6 Properties of the Technique 39
3.6.1 Soundness 40
3.6.2 Totality 40

4 Implementation of the Replay Technique in KeY 41
4.1 Communication with Z3 41
4.2 Overview and Architecture 42
4.3 Implementation Challenges 43
4.4 Current Limitations of the Prototype 44

5 Evaluation **47**
5.1 Problems Automatically Provable by KeY 47
5.2 Problems Not Provable by KeY's Built-In Proof Seach 50

6 Conclusion **51**
6.1 Summary 51
6.2 Future Work 51

A Appendix **55**
A.1 Example for Incompleteness With Epsilon in Input 55
A.2 Comparison of the Type Hierarchy Axiom Count 55
A.3 File Paths of Evaluation Examples Inside the KeY Repository 56

List of Figures

2.1 The type hierarchy of KeY . . . 6
2.2 Closing rules of KeY . . . 7
2.3 Propositional calculus rules of KeY . . . 7
2.4 Quantifier rules of KeY . . . 8
2.5 Congruence rules of KeY . . . 8
2.6 Other rules available in KeY . . . 9
2.7 Propositional simplification rules in KeY . . . 9
2.8 Example proof obligation formula in the KeY GUI . . . 10
2.9 Overall system architecture of Z3 . . . 12
2.10 Architecture of Z3's SMT core solver . . . 12

3.1 Overview of the intended workflow of the technique . . . 20
3.2 Generic replay scheme for most rules . . . 22
3.3 The type hierarchy after translation to Z3 . . . 25
3.4 Axioms of the original type hierarchy translation . . . 26
3.5 Axioms of the modified type hierarchy . . . 28

4.1 Example of Z3 Input . . . 42
4.2 Overview of the implementation architecture . . . 43
4.3 Example proof output of Z3 . . . 44

1 Introduction

1.1 Motivation

Computing devices are ubiquitous nowadays. From smartphones to cash cards, from traffic lights to cars and planes, from fridges to cardiac pacemakers: In our everyday life, we rely heavily on them. Therefore, it is essential for our safety and security that they do not fail or behave erroneously. With increasing computing power, the software running on these devices also becomes more and more complex, raising the chance of bugs. Usually, developers try to increase confidence in the correctness of programs by extensively (and automatically) testing them.

However, this can not prove the absence of bugs. For safety and security critical software, it is therefore reasonable to apply formal methods to formally verify desired properties of the software. As one tool in this area, the program verifier KeY allows for formal verification of specified properties of Java programs. The current workflow in KeY is as follows: When loading a Java program and its formal specification, KeY generates a proof obligation formula in dynamic logic, that is containing a Java program. The Java program is then simplified step-by-step, the state changes are collected in so called *updates*. While KeY's automatic built-in proof search is able to reliably perform these program transformation steps, known as *symbolic execution* and *update simplification*, it is not equally good at finding a proof for the resulting first-order problem. A reason for this is certainly the much larger search space for such a proof. As a solution, the user can either guide the prover by interactively applying rules in the graphical user interface of KeY, or run the built-in translation of the problem to SMT solvers.

SMT (*satisfiability modulo theories*) solvers try to find a solution for the satisfiability of a given formula, where a *theory*, that is, a set of axioms for some of the symbols, is given. Examples of such theories are for example linear integer arithmetic or fixed-size bit-vectors. Modern SMT solvers provide highly optimized procedures for deciding satisfiability of quantifier-free formula sets. However, as consequence of the incompleteness of first-order logic, they can not contain decision procedures for formulas with quantifiers in general. Nonetheless, they very often succeed in finding a solution by using quantifier elimination and heuristic instantiation.

When running such an SMT solver with input from KeY, the SMT solver is often able to prove the given problem, even if KeY's automatic proof search fails. This SMT result would be applied to the proof, thereby closing it. While this approach is quite effective in practice, there are disadvantages: First of all, the user has to trust the result of the SMT solver, with no possibility to check it in KeY. Second, the information of how the problem was solved, that is for example, which quantifier instantiations or arithmetical reasoning steps were needed, is opaque for the user. Finally, the resulting proof is partially in KeY's

"language", interwoven with "closed by SMT" annotations. When saving and reloading such a proof, the SMT solver has to be run again to obtain a completely closed proof. Some modern SMT solvers, such as Z3, are able to produce certificates (proofs and models) for their result.

This thesis describes a replay technique for the proofs generated by Z3 within KeY. As a result, a proof only in the format of KeY is obtained, where no more annotations as described above are necessary.

1.2 Contributions

On the theoretical level, this work shows that it is possible to replay Z3 proofs in KeY. More precisely, the thesis describes how the proofs Z3 generates when given the translation of a proof obligation from KeY can be translated back into a closed proof in KeY. Various challenges are identified that are intrinsic for such a translation:

- Different proof directions: Z3 produces proofs in a constructive direction starting from leaves, while KeY deconstructs/simplifies the proof obligation formula until each branch can be closed.
- Equisatisfiability: It may occur in Z3 proofs, but no such notion is present in KeY.
- Skolemization: As consequence of the different proof directions, for representing skolem constants/functions the Hilbert choice operator ϵ has to be used.
- Different type hierarchies: In Z3, all types are top level, while KeY features a nested type hierarchy suited for verification of Java programs. A particular challenge here is how to axiomatize the type hierarchy in such a way that the resulting Z3 proofs are replayable.
- Structurally different terms: Some terms can not be translated in a one-to-one way, for example since "and" is polyadic in Z3, but binary in KeY.
- Granularity of rules: Some proof rules of Z3 represent very coarse reasoning steps.

For all of these systematic challenges, solutions are presented.

Complementing the theoretical insights, the replay technique has been implemented as prototype in KeY. Thereby, several implementation difficulties such as handling shared terms, finding the matching formulas during replay of the assertion rule, and using `ifEx` as a replacement for the Hilbert choice operator ϵ, have been solved.

The insights provided by the evaluation are two-fold: First, it is shown that the implemented technique actually works and can be useful for proving formulas where KeY fails to find a proof. Second, it can be seen that there is currently great optimization potential for making the implementation more efficient, most notably by implementing caching of terms and by exploiting sharing of sub-proofs in KeY.

1.3 Related Work

While there are many SMT solvers that at least partially support the current SMT-LIB standard 2.6 (Barrett et al., 2017), solvers with proof production capabilities are rare. In general, the SMT community does not seem to focus much on proof generation. This can be seen for example by the fact that, while there exist suggestions for a standard proof format for SMT solvers (Stump, 2009; Besson et al., 2011), still none is accepted. Therefore, at the moment each solver uses its own format. State of the art proof producing SMT solvers are:

- CVC3 (Barrett and Tinelli, 2007) and it's successor CVC4 (Barrett et al., 2011) (proof support only for quantifier free formulas)
- Z3 (de Moura and Bjørner, 2008a,b)
- PRINCESS (Rümmer, 2008a,b)
- veriT (Bouton et al., 2009)

For this work, Z3 is used, since among the presented solvers it seems to have the most mature proof production capabilities.

Some work exists in the area of proof replay: Probably the first approach to replay SMT proofs using a theorem prover is described in (Fontaine et al., 2006). Here, proofs from haRVey (the predecessor of veriT) are replayed using Isabelle/HOL. However, the technique is for quantifier-free formulas only. A different approach is used in (Reynolds et al., 2010) and (Stump et al., 2012), where a conversion method from CVC3 proofs to the "Logical Framework with Side Conditions" (LFSC) is described. Proofs given in this format are afterwards checked by a dedicated checker. As already pointed out above, the CVC family of SMT solvers provides proof generation only for quantifier-free formulas. There is also a tool called SMTCoq (Armand et al., 2011), which allows for checking proofs from veriT and CVC4 inside the theorem prover Coq.

The work most similar to ours is (Böhme, 2009; Böhme, 2012), which describes a technique to check Z3's proofs inside Isabelle/HOL. A significant difference to our work is that our proof obligations originate from KeY, are then translated to Z3, and the proofs afterwards back to KeY. This imposes challenges such as type hierarchy translation that were not present in their work. Nonetheless their existing formal description of a majority of Z3's proof rules made the work presented here much easier.

1.4 Outline

The thesis is structured as follows: In Chapter 2, introductions to the KeY system and Z3 are given. For the KeY part, this covers in particular its core logic JavaDL with the type hierarchy in use as well as the sequent calculus used for reasoning. When introducing Z3, special attention is drawn to its proof production features.

In Chapter 3, the theory of the replay technique is explained. After giving an overview of the intended workflow of the technique and clarifying notions, the theoretical challenges

for replay are discussed, presenting the solutions chosen for this work directly at hand. The following section shows in detail how every Z3 proof rule is mapped to rule applications in KeY. An explanation of the theoretical properties of the technique concludes the chapter.

Chapter 4 gives some implementation detail about the prototype and explains challenges arising from an engineering point of view.

In Chapter 5, some experimental results from applying the implemented technique are shown.

Chapter 6 concludes the thesis by summing up the contributions and lessons learned and describing possible future work and alternative approaches that showed up during the development of this thesis.

2 Preliminaries

In this section, we give introductions to the KeY system and Z3. We focus on features used for the work described later on in this thesis; mainly KeY's type hierarchy, calculus and Z3's proof production facilities.

2.1 Introduction to the KeY System

The KeY system (Ahrendt et al., 2016) is a tool for formal verification of Java programs developed since 1998. Just recently, version 2.8 was released, featuring several new features such as support for bounded sums, a flexible and configurable user interface, the ability to save proofs including all their resources into a single bundle, and much more.

While there are other use cases beyond functional verification that can be addressed with KeY, such as verification based test case generation or conducting information flow proofs, here we only describe functional specification and verification of Java programs. To restrict that even more, while there is an API and also a command line interface to some of the core functionality of KeY, we only consider verification using the graphical user interface, as it is clearly the most common use case for functional verification and the relevant feature of KeY for this thesis.

Type Hierarchy KeY's underlying logic is Java Dynamic Logic (JavaDL). We refrain from explaining syntax and semantics here and instead refer to (Ahrendt et al., 2016). Here we give only a rough overview of features which are needed for the scope of this thesis.

The first order subset of JavaDL is called Java First-Order Logic (JavaFOL), which is basically the well known first-order logic enriched with a type hierarchy suited for the needs of Java program verification. The mandatory type hierarchy of JavaFOL is shown in Figure 2.1. It strongly depends on the type hierarchy of Java. There is a top level type called *Any* that all other types are subtype of. The hierarchy consists of the following parts:

1. A top level type (*Any*).
2. JavaFOL counterparts of Java's primitive types (*boolean*, *int*). Note that the unbounded type *int* is used to capture Java's finite width integer types *int*, *byte*, *short*, etc. Also, Java's floating point types *float* and *double* have no counterpart in JavaFOL, since reasoning about them is not supported.
3. A type for the theory of sequences, which is included in JavaFOL (*Seq*).
4. Types for handling memory locations (*Heap*, *Field*, *LocSet*).

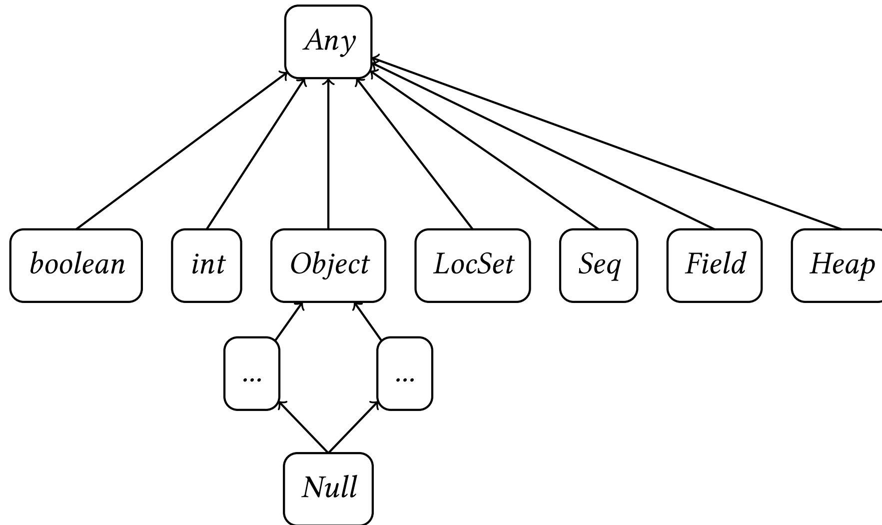

Figure 2.1: The type hierarchy of KeY

5. Java's object hierarchy (*Object* and subtypes). The dots indicate all object (and interface) types present in Java, they are included in JavaFOL's type hierarchy with subtype relations as declared in Java. The type *Null* is subtype of every object type, it has only a single element (`null`).

From a type theoretical perspective, the type hierarchy includes the universal type $\top$ at the top of the hierarchy, as well as the empty type $\bot$ at the bottom. We omit both here for easier readability, and also for easier comparability with Figure 3.3.

Sequent Calculus The calculus used for reasoning in KeY is the sequent calculus. A sequent is of the form[1] $\phi_1, \dots, \phi_n \vdash \psi_1, \dots, \psi_m$, the left hand side of the turnstile is called *antecedent* and the right hand side *succedent.* This intuitively has the following semantics: If all the formulas $\phi_1, \dots, \phi_n$ hold, it is to show that at least one ψ_i holds for any $i \in \{1, \dots, m\}$ to prove the sequent valid. The sequent calculus decomposes complex formulas into simpler ones by applying inference rules. More than 1500 rules are available, which can be grouped into four different categories:

1. *Nonprogram rules* are used to handle statements about a single state without modalities.

2. *Symbolic execution rules* transform a program inside a modality into updates and case distinctions.

[1]Note that in KeY, instead of the turnstile $\vdash$, the sequent arrow $\Longrightarrow$ is used.

3. *Update simplification rules* are used for applying updates to formulas and simplifying them, thus effectively transforming formulas about multiple states into formulas about a single state.

4. *Rules for program abstraction and modularization*, for example rules for handling loop invariants and method calls.

Calculus rules are of the following form:

$$\frac{P_1 \quad \dots \quad P_n}{C} \; ruleName$$

The P_i are called *premises*, C is called *conclusion* of the rule. Note that in this thesis, all names of KeY rules are displayed in italic font. Rules are applied bottom up, which means that rules with more than one premise split the proof into multiple branches. A proof is closed if all branches are closed, a branch can be closed by applying one of the following closing rules:

$$\frac{}{\Gamma \vdash \top} \; closeTrue \qquad \frac{}{\Gamma, \bot \vdash} \; closeFalse \qquad \frac{}{\Gamma, \phi \vdash \phi} \; close$$

Figure 2.2: Closing rules of KeY

Γ stands for any set of formulas here. Note that while there could be formulas on the right hand side of the turnstile, their negation could be included in Γ. Therefore, we only show Γ here to keep the presentation as clean as possible. In addition, instead of $\Gamma \cup \{\phi\}$ we write Γ, ϕ.

All rules presented here and used later during the thesis are nonprogram rules (category 1), since symbolic execution and update simplification typically are handled by the automatic proof search algorithm of KeY without any problems or required user interactions. First of all, Figure 2.3 shows propositional rules of KeY. The most notable rule here is the *cut* rule, which allows for arbitrary case distinctions. This rule will play an important role later.

$$\frac{\Gamma, \phi, \psi \vdash}{\Gamma, \phi \wedge \psi \vdash} \; andLeft \qquad \frac{\Gamma, \phi \vdash \quad \Gamma, \psi \vdash}{\Gamma, \phi \vee \psi \vdash} \; orLeft \qquad \frac{\Gamma \vdash \phi}{\Gamma, \neg\phi \vdash} \; notLeft$$

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \; andRight \qquad \frac{\Gamma \vdash \phi, \psi}{\Gamma \vdash \phi \vee \psi} \; orRight \qquad \frac{\Gamma, \phi \vdash}{\Gamma \vdash \neg\phi} \; notRight$$

$$\frac{\Gamma, \phi \vdash \quad \Gamma \vdash \phi}{\Gamma \vdash} \; cut$$

for a ground formula ϕ

Figure 2.3: Propositional calculus rules of KeY

KeY allows for skolemization of quantifiers with two rules: *allLeft* and *exRight* as shown in Figure 2.4. The term $[x/c]\phi$ denotes a substitution of all free occurrences of x in ϕ by c. In the skolemization rules, c denotes a fresh constant. Note that using this technique, KeY only uses skolem *constants*, since the freshly introduced symbol does not depend on anything. Instantiation of quantifiers can be done via *allLeft* and *exRight*,

$$\frac{\Gamma, \forall x.\ \phi, [x/t]\phi \vdash}{\Gamma, \forall x.\ \phi \vdash}\ \mathit{allLeft}$$

for any ground term t

$$\frac{\Gamma, [x/c]\phi \vdash}{\Gamma, \exists x.\ \phi \vdash}\ \mathit{exLeft}$$

where c is a fresh constant

$$\frac{\Gamma \vdash [x/c]\phi}{\Gamma \vdash \forall x.\ \phi}\ \mathit{allRight}$$

where c is a fresh constant

$$\frac{\Gamma \vdash \exists x.\ \phi, [x/t]\phi}{\Gamma \vdash \exists x.\ \phi}\ \mathit{exRight}$$

for any ground term t

$$\forall x.\ \phi \rightsquigarrow \phi \quad \mathit{all_unused}$$

if x does not occur free in ϕ

$$\neg\forall x.\ \phi \rightsquigarrow \exists x.\ \neg\phi \quad \mathit{nnf_NotAll}$$

Figure 2.4: Quantifier rules of KeY

under the condition that the instantiation term t does not contain free variables. Unused universal quantifiers can be eliminated via *all_unused* (of course, there is also the dual rule for existential quantifiers). In addition, many rules are available to bring formulas into negation normal form (NNF). We just want to show *nnf_NotAll*, which allows to shift a negation to the inside of a quantifier (DeMorgan's Law). Note that *all_unused* and *nnf_NotAll* are not classical calculus rules, but *rewrite rules* that allow for replacing the term on the left by that on the right anywhere on the sequent. In KeY, calculus rules and rewrite rules (possibly with conditions) can be expressed as so called *taclets*. With this, reasoning rules can be formulated in a concise way. In addition, it allows the user to provide hints (*heuristics*) to the automatic proof search strategy in which proof situations a rule should be used.

For reasoning about equality = and equivalence $\leftrightarrow$, the rules shown in Figure 2.5 are available. Note that there is currently no rule corresponding to *eqSymm* for symmetry of equivalence.

$$\phi \leftrightarrow \psi \rightsquigarrow \top \qquad \mathit{eq_eq}$$

$$t = t \rightsquigarrow \top \qquad \mathit{eqClose}$$

$$s = t \rightsquigarrow t = s \qquad \mathit{eqSymm}$$

Figure 2.5: Congruence rules of KeY

For a known equality $s = t$ of two terms s and t, one can replace an occurrence of s by t in any formula by using the rule *applyEq*. One could want to do the same for equivalence: Replace ϕ by ψ if $\phi \leftrightarrow \psi$ is present on the left hand side of the sequent. However, there is no such rule as *applyEquiv* in KeY. Instead, there is the rule *insert_eqv_lr* (or *insert_eqv_rl* for the reverse direction), which creates a new rule *insert_eqv*[2]. Afterwards, this newly added rule can be used to replace occurrences of ϕ. In this thesis, we abbreviate these two steps and use the hypothetical rule *applyEquiv*.

In addition to all the rules presented above, there are rules that can help the user as well as the automatic proof search by hiding formulas considered superfluous for the proof (*hideLeft* and *hideRight* in Figure 2.6). To leave the completeness of the calculus intact, applying one of these rules introduces a new one to the calculus to re-insert the hidden formula. To shorten proofs, the rules *replace_known_left* and *replace_known_right* serve

$$\frac{\Gamma, \vdash}{\Gamma, \phi \vdash}\ \textit{hideLeft}$$

This rule introduces a new taclet that re-introduces the hidden formula ϕ.

$$\frac{\Gamma \vdash}{\Gamma \vdash \phi}\ \textit{hideRight}$$

This rule introduces a new taclet that re-introduces the hidden formula ϕ.

$$\phi \rightsquigarrow \top \qquad \textit{replace_known_left}$$

under the condition that ϕ occurs top level on the left side of the sequent

$$\phi \rightsquigarrow \bot \qquad \textit{replace_known_right}$$

under the condition that ϕ occurs top level on the right side of the sequent

Figure 2.6: Other rules available in KeY

to replace formulas that appear top level on the sequent by true or false, depending on the side of the sequent they appear. These two rules are rewrite rules with conditions: ϕ has to occur top level on the sequent. Finally, there is a family of boolean simplification rules for all boolean connectives, where one parameter is true or false. While these are all different rules in KeY, for brevity we want to refer to all of them as *simplify*.

$$\begin{array}{lll}
\top \land \phi \rightsquigarrow \phi & \top \lor \phi \rightsquigarrow \top & \top \leftrightarrow \phi \rightsquigarrow \phi \\
\bot \land \phi \rightsquigarrow \bot & \bot \lor \phi \rightsquigarrow \phi & \bot \leftrightarrow \phi \rightsquigarrow \neg\phi \\
\phi \land \top \rightsquigarrow \phi & \phi \lor \top \rightsquigarrow \top & \phi \leftrightarrow \top \rightsquigarrow \phi \\
\phi \land \bot \rightsquigarrow \bot & \phi \lor \bot \rightsquigarrow \phi & \phi \leftrightarrow \bot \rightsquigarrow \neg\phi
\end{array}$$

Figure 2.7: Propositional simplification rules in KeY (all called *simplify* in this thesis)

[2]The insert_eqv_lr/rl rules are only manually applicable, since KeY typically handles equivalence by splitting it into two implications.

```
==>
     wellFormed(heap)
   & !self_25 = null
   & self_25.<created> = TRUE
   & SumAndMax::exactInstance(self_25) = TRUE
   & (a = null | a.<created> = TRUE)
   & measuredByEmpty
   & (\forall int i; (0 <= i & i < a.length & inInt(i) -> 0 <= a[i]) & (self_25.<inv> & !a = null))
-> {heapAtPre_0:=heap || _a:=a}
     \<{
         exc_25=null;try {
           self_25.sumAndMax(_a)@SumAndMax;
         }   catch (java.lang.Throwable e) {
           exc_25=e;
         }
       }\> (  \forall int i; (0 <= i & i < a.length & inInt(i) -> a[i] <= self_25.max)
            & (  (a.length >  0 -> \exists int i; (0 <= i & i < a.length & inInt(i) & self_25.max = a[i]))
               & (  self_25.sum = javaCastInt(bsum{int i;}(0, a.length, a[i]))
                  & (self_25.sum <= javaMulInt(a.length, self_25.max) & self_25.<inv>)))
            & exc_25 = null
            & \forall Field f;
                \forall java.lang.Object o;
                  (  (o, f) \in {(self_25, SumAndMax::$sum)} \cup {(self_25, SumAndMax::$max)}
                   | !o = null
                   & !o.<created>@heapAtPre_0 = TRUE
                   | o.f = o.f@heapAtPre_0))
```

Figure 2.8: The presented proof obligation formula after loading the SumAndMax example in the KeY GUI

Workflow for Functional Verification using the GUI For functional verification of Java programs, the workflow is as follows: The Java source code is annotated with a formal specification in the Java Modeling Language (JML). Code and specification are then given to KeY as input. From that, KeY generates a proof obligation, which is a formula of dynamic logic:

$$\phi \rightarrow \langle p \rangle \, \psi$$

ϕ is the precondition (translated from JML), p is the program, and ψ is the postcondition. Intuitively, the above formula states that if the precondition holds, then after (terminating) execution of the program the postcondition holds. Figure 2.8 shows the graphical user interface of KeY after loading the SumAndMax example shipped with KeY. The goal is now to show the validity of this formula in KeY. On an abstract level, the proof consist of the following steps: The proof obligation formula is a real dynamic logic formula with usually one modality. First of all, the program inside the modality is reduced step by step, a process called *symbolic execution* (rules of category 2 and 4 from above). During this, usually case distinctions in the proof are made, splitting it up into multiple sub-goals. The state changes which result from assignments to variables are captured using so called *updates*, which describe transitions between states. In comparison to the standard weakest precondition calculus, this technique with updates has the advantage of allowing for symbolic execution in a forward way, making the proof much easier to read and understand for humans. After the symbolic execution is finished, all branches of the proof contain no more modalities. Next, the updates are simplified and finally removed, which results in a pure first order problem without updates or modalities (rules of category 3). While symbolic execution

and update simplification usually are straight forward, this first order part of the proof is a real challenge and the main focus of this thesis.

SMT Integration KeY has a built-in proof search technique that is often able to find the necessary rule applications from the rules presented above (and much more rules which are not shown here), and thus to close the proof. However, since the search space is large, often it does not find a proof, which may happen for example when quantifier instantiations are needed. One solution is to apply proof rules interactively, provided that the user has enough understanding to find useful rule applications. Another approach is to use modern SMT solvers which are often able to find a proof even if the built-in proof search algorithm of KeY fails. Therefore, a translation from KeY sequents to SMT solvers has long been part of KeY, currently there is ongoing work in modernizing and modularizing it. For generality, the SMT translation uses the SMTLIB format, which is the standard interchange format for SMT input. Currently, the solvers Z3, CVC3, Simplify, and Yices are supported.

If the SMT solver could prove the problem valid, the goal in KeY would be marked as closed. Note that the SMT translation is under-approximating in the sense that formulas containing updates or modalities are not translated at all. Therefore calling the SMT solver when such are present very likely does not lead to a successful verification attempt. In the graphical user interface, the user gets a warning in this situation.

2.2 Introduction to Z3

Z3 is a state of the art SMT solver developed by Microsoft Research. Since 2015, the code is available at GitHub[3] under the MIT license. Z3 incorporates a decision procedure for quantifier free formulas combined from theories such as arrays, uninterpreted functions, bit-vectors, arithmetic, and more. While the general decision problem for quantifier-free formulas is undecidable, Z3 manages to find results for many formulas arising in practice.

Input can be given to Z3 in the SMT-LIB standard format or by using one of the APIs available for various programming languages, such as C, C++, Java, Python, and OCamL.

Alongside the general SMT solver, various specialized solvers are included, for example for purely propositional formulas, or NLSat for nonlinear-arithmetic (as shown in Figure 2.9). Users can fine-tune the solver to their needs by using tactics.

The core of Z3's generic SMT solver consists of a theory solver for uninterpreted functions with equality, a DPLL based SAT solver with clause learning (CDCL), and separate specialized theory solvers. It works by applying a DPLL(T) style "ping pong game" between SAT solver and theory solvers. The theory solver for equality and uninterpreted functions (EUF) is integrated with the SAT solver. An overview of the architecture of Z3's SMT solver is shown in Figure 2.10.

As stated above, Z3 does not contain a decision procedure for arbitrary first-order formulas with quantifiers, as first-order logic is not decidable in theory. However, it is still able to decide satisfiability for many formulas containing quantifiers using the following

[3]https://github.com/Z3Prover/z3/

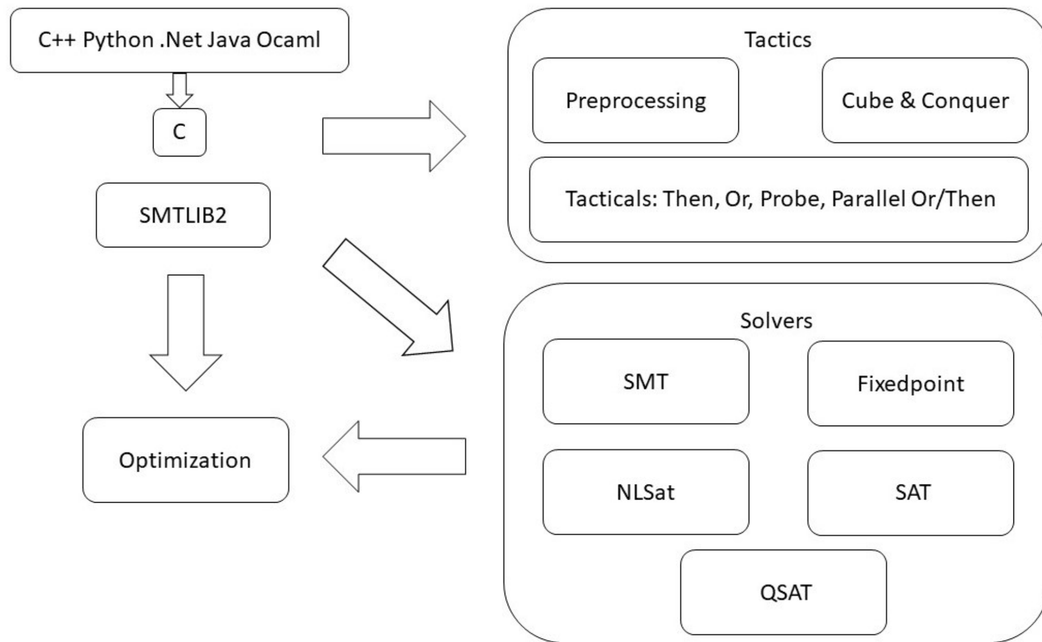

Figure 2.9: Overall system architecture of Z3 (from (Bjørner et al., 2019))

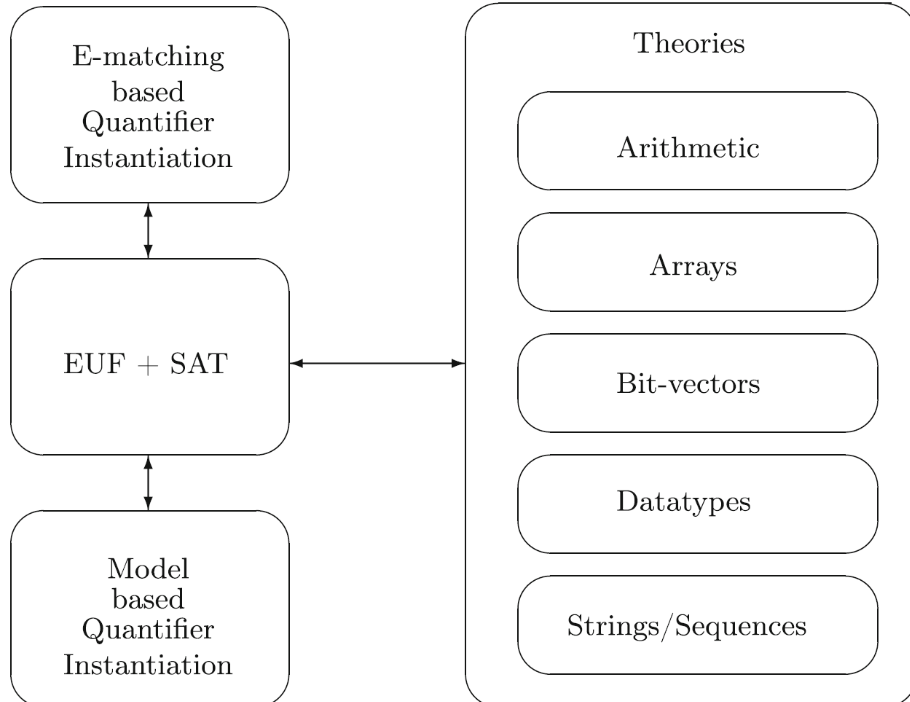

Figure 2.10: Architecture of Z3's SMT core solver (from (Bjørner et al., 2019)

techniques: First of all, Z3 provides a heuristic mechanism called *E-matching*. It is based on *patterns*, which are annotations given by the user in the input file. These patterns provide hints to the solver what could be useful instantiations of the quantified variables. While this approach is quite effective in practice, it is incomplete. Therefore, there is a complementing approach called *Model-based Quantifier Instantiation (MBQI)*. MBQI works by producing a candidate model from the quantifier free part of the given problem and checking it. If it does not satisfy the problem with quantifiers, it is iteratively refined.

When queried about the satisfiability of a set of formulas, Z3 may give one of three possible results:

- If Z3 found a model for the formula set, it prints `sat`. The model can be retrieved by `get-model`.
- If Z3 is able to prove the formula set unsatisfiable, it prints `unsat`. If proof production had been enabled by `(set-option :produce-proofs true)`, the proof can be retrieved via `get-proof`.
- It may be the case that Z3 is not able to find a solution for (un-)satisfiability of the input formulas. In this case, it prints `unknown`.

2.2.1 Proof Production

Note that Z3's proof production in general is not documented very well, for example, for many rules there is only a single sentence of text roughly describing what the proof rule is intended to mean. The descriptions presented here have partly been taken from the work of Sascha Böhme (Böhme, 2009; Böhme, 2012), and the rest carefully reverse-engineered from the papers describing Z3 (de Moura and Bjørner, 2008b; Bjørner et al., 2019), its sparse documentation, Z3's source code, and experimenting with Z3 and studying the produced proofs.

Z3 produces proofs of unsatisfiability, deducing $\bot$ from a set of assertions. Due to the CDCL(T) nature of Z3's solver, proofs are natural deduction style proofs: The SAT core uses unit resolution and a few other propositional rules, while for conflict clauses from clause learning are handled via `lemma` and `hypothesis`. The congruence closure implementation (for solving EUF) may introduce the rules `refl`, `symm`, `trans`, `monotonicity`, and `commutativity`. The theory solvers only give very coarse proofs, resulting only in a single rule application (`th-lemma` or `rewrite`). For quantifier instantiation, a single rule `quant-inst` exists, skolemization steps are justified by the `sk` rule. In addition, there are some rules used to justify simplification and normalization steps, for example `push-quant`, `pull-quant`, `der`, `nnf-pos`, `nnf-neg`, and `def-axiom`. Finally, some rules compress multiple steps, such as `rewrite*` or `trans*`. In total, there are 42 rules that may occur in a Z3 proof.

Z3 prints proofs is terms of sort `Proof` in the usual SMT-LIB prefix notation. Proof rules are presented as functions, taking proofs for premises of the rule and producing a proof for the formula given as last parameter:

$$\texttt{rulename} : \texttt{Proof} \times \ldots \times \texttt{Proof} \times \text{Formula} \rightarrow \texttt{Proof}$$

For example, $mp(p, q, \phi)$ denotes a proof for ϕ, where p is a proof term for $\psi \rightarrow \phi$ and q is a proof for ψ. For readability and easy comparison to the rules of KeY from the previous section, we present the rules using Gentzen style sequents. To facilitate distinction between KeY and Z3 rules and proof trees, all Z3 rule names in this thesis are displayed in typewriter font.

Propositional Rules The rules in this paragraph are produced by the SAT part of the CDCL(T) solver. The most simple rule produces a proof for true (`true-axiom`). The rules `asserted` and `goal` are used as leaf rules by Z3 to create a proof for an assertion given as input by the user. While they are semantically equal, `goal` is intended to retain "the distinction between goals and assumptions in proof objects" (de Moura and Bjørner, 2008b). Note that, however, it is not clear to the us what this should mean. The `goal` rule did neither occur in any of the proofs produced while experimenting with Z3 nor could any example be found in any of the sources describing proof production.

$$\frac{}{\Gamma \vdash \top}\ \texttt{true-axiom} \qquad \frac{}{\Gamma, \phi \vdash \phi}\ \texttt{asserted} \qquad \frac{}{\Gamma, \phi \vdash \phi}\ \texttt{goal}$$

The rules `hypothesis` and `lemma` are closely connected: While the former introduces any formula as hypothesis to the proof, the latter is used to justify this assumption: The `lemma` rule states that a refutation can be constructed from the literals $L_1, \ldots, L_n$, therefore, it can be concluded that at least one of them does not hold.

$$\frac{}{\Gamma, \phi \vdash \phi}\ \texttt{hypothesis} \qquad \frac{\Gamma, L_1, \ldots, L_n \vdash \bot}{\Gamma \vdash \neg L_1 \vee \ldots \vee \neg L_n}\ \texttt{lemma} \qquad \text{where } \Gamma \text{ does not contain any } L_i \text{ for } i \in \{1, \ldots, n\}$$

The core rule produced by the SAT solver is `unit-resolution`, which justifies multiple unit resolution steps at once:

$$\frac{\Gamma \vdash \bigwedge_{i \in I_s} L_i \qquad \langle i \in I_s \mid \Gamma_i \vdash \neg L_i \rangle}{\Gamma \cup \bigcup_{i \in I_s} \Gamma_i \vdash \bigvee_{i \in I \setminus I_s} L_i}\ \texttt{unit-resolution} \qquad \text{where } I = \{1, \ldots, n\}, I_s \subseteq I$$

The modus ponens rule is as expected. Note that $\rightsquigarrow$ denotes either $\rightarrow$, $\leftrightarrow$, or $\sim$.

$$\frac{\Gamma_1 \vdash \phi \quad \Gamma_2 \vdash \phi \rightsquigarrow \psi}{\Gamma_1, \Gamma_2 \vdash \psi}\ \texttt{mp}$$

The rules `iff-true` and `iff-false` may be used for basic propositional transformations, while the `iff~` rule can be used to weaken a proposition by replacing equivalence by equisatisfiability:

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \leftrightarrow \top}\ \texttt{iff-true} \qquad \frac{\Gamma \vdash \neg\phi}{\Gamma \vdash \phi \leftrightarrow \bot}\ \texttt{iff-false} \qquad \frac{\Gamma \vdash \phi \leftrightarrow \psi}{\Gamma \vdash \phi \sim \psi}\ \texttt{iff}\sim$$

`and-elim` and `not-or-elim` both weaken a proposition by selecting one of a set of literals:

$$\frac{\Gamma \vdash L_1 \land \ldots \land L_n}{\Gamma \vdash L_i}\ \texttt{and-elim} \qquad \frac{\Gamma \vdash \neg(L_1 \lor \ldots \lor L_n)}{\Gamma \vdash \neg L_i}\ \texttt{not-or-elim}$$

Congruence Rules The EUF solver (congruence closure) may introduce the following rules for justifying either reflexivity, symmetry, or transitivity of any of the relations $=$, $\leftrightarrow$, or $\sim$. As abbreviation, $\simeq$ is used to denote any of them.

$$\frac{}{\Gamma \vdash t \simeq t}\ \texttt{refl} \qquad \frac{\Gamma \vdash t_1 \simeq t_2}{\Gamma \vdash t_2 \simeq t_1}\ \texttt{symm} \qquad \frac{\Gamma_1 \vdash t_1 \simeq t_2 \quad \Gamma_2 \vdash t_2 \simeq t_3}{\Gamma_1, \Gamma_2 \vdash t_1 \simeq t_3}\ \texttt{trans}$$

The `trans*` rule can be used to compress multiple symmetry and transitivity rule applications. The conclusion is deduced by applying symmetry and transitivity to the premises in any order and direction.

$$\frac{\Gamma_1 \vdash t_1 \simeq t_2 \quad \ldots \quad \Gamma_n \vdash t_{n-1} \simeq t_n}{\Gamma_1, \ldots, \Gamma_n \vdash t_i \simeq t_j}\ \texttt{trans*} \qquad \text{for any } i, j \in 1, \ldots, n, i \neq j$$

In addition, for any commutative relation $\diamond$, the following rule may be used:

$$\frac{\Gamma \vdash (t_1 \diamond t_2 \simeq s_1 \diamond s_2)}{\Gamma \vdash (t_2 \diamond t_1 \simeq s_2 \diamond s_1)}\ \texttt{commutatitity}$$

For any congruence relation f, the following rule is available:

$$\frac{\Gamma_1 \vdash t_1 \simeq s_1 \qquad \ldots \qquad \Gamma_n \vdash t_n \simeq s_n}{\Gamma_1, \ldots, \Gamma_n \vdash f(t_1, \ldots, t_n) \simeq f(s_1, \ldots, s_n)}\ \texttt{monotonicity}$$

Note that if $t_i \simeq s_i$ for any $i \in \{1, \ldots, n\}$, this reflexivity premise is omitted to save space.

Quantifier Rules Despite the name, this rule should not be confused with the classical quantifier introduction rules of natural deduction. This can already seen by the fact that the rule can be used with Q either being $\forall$ or $\exists$. In the premise, $\overline{x}$ are free variables, which can be considered implicitly bound by a universal quantifier (for more details, see Section 3.2). Therefore, the rule actually roughly means deducing global from point-wise equisatisfiability.

$$\frac{\Gamma \vdash \phi(\overline{x}) \sim \psi(\overline{x})}{\Gamma \vdash Q\overline{x}.\ \phi(\overline{x}) \sim Q\overline{x}.\ \psi(\overline{x})}\ \texttt{quant-intro}$$

In the current version this rule is always followed by `proof-bind`[4], which converts a proof for a formula $\phi(\overline{x})$ with free variables $\overline{x}$ to a proof of $\lambda\,\overline{x}.\ \phi(\overline{x})$. So, it is actually not a real calculus rule, but a binder, binding "free" variables in the whole subtree of the proof pending from it. This makes understanding the proof easier, since one gets an explicit list of variables which are considered "free" from there.

[4]Versions prior to 4.8.1 (October 2018) had real free variables not bound anywhere. In these versions, `proof-bind` was not present, the variables were silently free from this point in proof tree.

The rule $\text{inst}_\forall$ is used to instantiate multiple quantified variables $\overline{x}$ with terms $\overline{t}$. It is presented as an axiom containing an implication transformed to CNF:

$$\frac{}{\Gamma \vdash \neg(\forall \overline{x}.\ \phi(\overline{x})) \lor \phi(\overline{t})}\ \texttt{quant-inst}$$

Skolemization is done by the `sk` rule, which actually has two forms, one for the existential and one for the negated universal quantifier. Since one could normalize the latter form by pushing the negation to the inside of the quantifier and changing it to an existential one, the only difference is that ϕ is preceded by an additional negation in this form. As in the other quantifier rules, multiple variables can be skolemized in a single rule application. Another important thing to note here is that $\overline{y}$ are free variables (more precisely, those are bound by a `proof-bind` rule closer towards the root of the proof). Consequently, it is necessary to skolemize using fresh function symbols depending on these free variables. It is also worth noting that since the formula on the right hand side of the equisatisfiability sign $\sim$ contains additional function symbols compared to that on the left hand side, both formulas are equisatisfiable, but not equivalent.

$$\frac{}{\Gamma \vdash (\exists \overline{x}.\ \phi(\overline{y}, \overline{x})) \sim \phi(\overline{y}, \overline{f}(\overline{y}))}\ \texttt{sk}_\exists \qquad \frac{}{\Gamma \vdash \neg(\forall \overline{x}.\ \phi(\overline{y}, \overline{x})) \sim \neg\phi(\overline{y}, \overline{f}(\overline{y}))}\ \texttt{sk}_{\neg\forall}$$

Finally, if the formula ϕ does not depend on bound variables $y_1, \ldots, y_m$, these can be removed from the quantifier using `elim-unused`:

$$\frac{}{\Gamma \vdash \forall(x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m).\ \phi(x_1, \ldots, x_n) \leftrightarrow \forall(x_1\ \ldots\ x_n).\ \phi(x_1, \ldots, x_n)}\ \texttt{elim-unused}$$

Theory Rules Before starting the actual "ping-pong game" between CDCL SAT solver and theory solvers, Z3 may use a simplifier to apply rewriting steps to terms. These rewriting steps may be propositional or theory-specific and are presented as axioms. Examples for possible axioms are $\phi \lor \bot \leftrightarrow \phi$ or $x + 0 = x$. The following three cases of rewrite may occur, where always the top level symbol of t is interpreted ($\lor$ and $+$ in the example).

$$\frac{}{\Gamma \vdash t = s}\ \texttt{rewrite} \qquad \frac{}{\Gamma \vdash t \sim s}\ \texttt{rewrite} \qquad \frac{}{\Gamma \vdash t \leftrightarrow s}\ \texttt{rewrite}$$

During the "ping-pong" between SAT solver and theory solvers, facts produced from the latter are introduced via the `th-lemma` rule. In contrast to the `rewrite` rule, Z3 *may* give some additional information about the used theory and the nature of the lemma (in our examples, more often no information was given). For example, `arith` may be given for the theory of arithmetic and `farkas` for Farkas' lemma or `triangle-eq` for $t_1 = t_2 \leftrightarrow t_1 \leq t_2 \land t_2 \leq t_1$. However, no complete list of theory lemmas is given in documentation.

$$\frac{}{\Gamma \vdash \phi}\ \texttt{th-lemma} \qquad (T\text{-tautology})$$

In addition to the use as axiom, the theory lemma rule can be used to deduce $\bot$ from a set of hypotheses (undocumented in API). This use case is tailored to the fit the `lemma` rule already presented above.

$$\frac{\Gamma \vdash L_1 \quad \ldots \quad \Gamma \vdash L_n}{\Gamma, L_1, \ldots, L_n \vdash \bot}\,\texttt{th-lemma}$$

Other Rules Four of Z3' rules can not easily be presented as a single calculus rule, as they consist of a set of schemata:

- For distributing conjunctions over disjunctions or vice versa, the `distributivity` rule is used. This rule contains multiple schemata, since conjunction and disjunction are polyadic in Z3 and may thus have an arbitrary number of parameters.
- For conversion to CNF, Z3 uses Tseitin's conversion. However, instead of introducing fresh names for sub-formulas, as usually done in Tseitin's conversion, Z3 uses the sub-formula itself as name, that is, it treats them atomically. This is called *implicit quotation*. To justify the steps of the Tseitin conversion, the rule `def-axiom` is used. This rule can introduce various axioms for various propositional tautologies into the proof, a non-complete list is given in (Böhme, 2012).
- The rules `nnf-pos` and `nnf-neg` are used to construct proofs of transformation to negation normal form of a formula. Similar to `def-axiom`, they consist of multiple schemata.

Finally, there are 11 rules that only occur in rare situations or when specific options in Z3 are set (during our experiments with Z3, neither of these rules were encountered):

- There is also an `undef` "rule" to represent an invalid/null proof.
- For compressing multiple rewrite steps into one, the rule `rewrite*` is used. However, it is only used if certain options are set to true (for converting bit-vectors to boolean or for contextual simplification).
- The `push-quant` and `pull-quant` rules can be used to move quantifiers from inside of functions to the outside, for example to distribute universal quantifiers over conjunctions.
- `der` (destructive equality resolution) and `hyper-resolve` represent simplification steps that can be composed of several other rule applications.
- According to the sparse documentation, `def-intro` and `apply-def` rules could be used to introduce an abbreviation for a term, and dually to insert the original term for its abbreviation. Since this is exactly what is done in nearly all proof examples by using the `let` binder, it seems that these two rules have been replaced by the more general concept with `let` (let binders can be used to share terms as well as whole subtrees of a proof).

- The rules `assumption-add`, `lemma-add`, `redundant-del`, and `clause-trail`, are used for clausal proofs, which is an alternative proof mode of Z3's core solver "SMT". Clausal proofs must be enabled explicitly by setting an option, however, for the examples tested, the produced proof did not differ if this flag was set.

3 The Replay Technique

3.1 Overview

The starting point for our replay technique is a sequent in KeY, where only first-order formulas are present. This means that all modalities (and thus, Java programs) have been removed by finishing symbolic execution as well as update simplification. Without loss of generality, we assume that there is only a single formula, which is on the right hand side, present in the sequent. We may assume this because the sequent $\Gamma \vdash \Delta$ with formula sets Γ and Δ could be encoded into a sequent with only a single formula[1]:

$$\vdash \bigwedge_{\gamma_i \in \Gamma} (\neg \gamma_i) \wedge \bigwedge_{\delta_i \in \Delta} \delta_i$$

KeY is concerned with the question if a formula is valid. Z3, on the other hand, can only check for (un-)satisfiability of formulas. However, there is a well-known relation between these two problems:

$$\phi \text{ is valid} \quad \text{iff} \quad \neg\phi \text{ is unsatisfiable}$$

For a sequent $\vdash \phi$, the idea is to query Z3 for satisfiability of $\neg\phi$. As shown in Figure 3.1, Z3 gives one of three possible results: For a timeout, it returns `unkown` (3a). If the formula is not valid (and Z3 had enough time), Z3 returns `sat` and a model, that is, a satisfying assignment of the variables in the input formula (3b). In this case, one may run the counterexample generator in KeY to get a hint which part of specification or code was wrong. If the original formula ϕ is valid, Z3 returns `unsat`. Until now, the user could apply the result to the branch in KeY, thus closing it. However, there was no possibility to check the result of Z3 nor to examine inside the KeY GUI how it was produced.

Z3 is able to output certificates for unsatisfiability ("proofs"). The idea is to use these and replay them using KeY's calculus rules. As a first step, proof production must be enabled in Z3 by adding `(set-option :produce-proofs true)` to the SMT-LIB input file. With that, Z3 produces a proof of unsatisfiability (3c). To this proof we apply a replay engine implementing the technique described in this chapter, which transforms the unsatisfiability proof for $\neg\phi$ in Z3 format into a validity proof for ϕ in KeY.

3.2 Clarifying Notions

Before explaining the actual replay technique, we have to clarify some notions. Since the replay technique is meant to be applied if the formula only refers to a single state (that is,

[1] In practice, the sequent does not have to be encoded, since Z3 can be given a set of formulas as input.

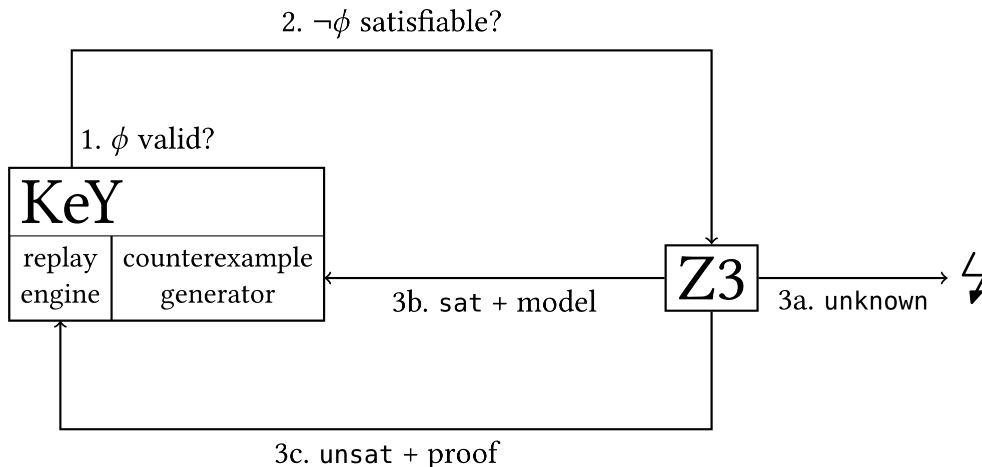

Figure 3.1: Overview of the intended workflow of the technique. The contributions of this thesis are in the "backward" translation (3c) and partly in the "forward" translation (2).

after symbolic execution and update simplification have been executed, see Section 2.1), we omit all features of dynamic logic here. However, the necessary changes for dynamic logic are identical to that described in (Ahrendt et al., 2016).

Type Hierarchy A type hierarchy is a pair $\mathcal{T} = (\text{TSym}, \sqsubseteq)$, where TSym is a set of type symbols, the subtype relation $\sqsubseteq$ is a reflexive and transitive relation on TSym. In addition, there are two designated type symbols, the *empty* type $\bot \in \text{TSym}$ and the *universal* type $\top \in \text{TSym}$, with $\bot \sqsubseteq A \sqsubseteq \top$ for all types $A \in \text{TSym}$.

Signature A *signature* Σ is a triple (FSym, PSym, VSym) of function symbols FSym, predicate symbols PSym, and variables VSym. Each of them has argument and result types as expected.

Universe A universe is a pair (D, δ), where D is a set and $\delta : D \to TSym \setminus \{\bot\}$ such that for every $A \in \text{TSym}$ the set $D^A = \{d \in D | \delta(d) \sqsubseteq A\}$ is non-empty.

First-Order Structure A *first-order structure* $\mathcal{M}$ consists of a domain (D, δ) and an interpretation such that $I(f)$ is a function from $D^{A_1} \times \ldots \times D^{A_n}$ into D^A for $f : A_1 \times \ldots \times A_n \to A$ in FSym, $I(p)$ is a subset $D^{A_1} \times \ldots \times D^{A_n}$ for $p(A_1, \ldots, A_n)$ in PSym, $I(=) = \{(d, d) | d \in D\}$.

Variable Assignment A *variable assignment* is a function $\beta : VSym \to D$ such that $\beta(v) \in D^A$ for $v : A \in \text{VSym}$.

Validity In KeY, the problem that is treated is that of validity of a formula ϕ:

$$\phi \text{ is valid} \qquad \text{iff} \qquad \text{for all } \mathcal{M} \text{ and all } \beta\text{:} \quad (\mathcal{M}, \beta) \models \phi$$

Satisfiability Z3, on the other hand, is concerned with showing *satisfiability*:

$$\phi \text{ is satisfiable} \quad \text{iff} \quad \text{there is any } \mathscr{M} \text{ and any } \beta \text{ such that: } \quad (\mathscr{M}, \beta) \models \phi$$

Equisatisfiability Of special interest is Z3's notion of *equisatisfiability*: According to de Moura and Bjørner (2008b), two formulas ϕ and ψ are equisatisfiable, if they are equivalent when closed by second order existential quantification of skolem constants and functions:

$$\begin{aligned} \phi(\overline{x}) \sim \psi(\overline{y}) \quad \text{iff} \quad \text{for all } \beta: \quad & \text{there is any} \mathscr{M}_1 : \ (\mathscr{M}_1, \beta) \models \phi(\overline{x}) \\ \text{iff} \quad & \text{there is any} \mathscr{M}_2 : \ (\mathscr{M}_2, \beta) \models \psi(\overline{y}) \end{aligned}$$

$\overline{x} = (x_1, \ldots, x_n)$ and $\overline{y} = (y_1, \ldots, y_m)$ are multiple free variables here. It is important to note that the variable assignment β is the same for ϕ and ψ.

3.3 Challenges

There are various challenges when translating a proof from Z3 to KeY. All challenges described in this section are systematic, that is, they are owed to the nature of both proof systems, and must thus be managed by every replay technique. The difficulty of the last two problems in this section, correctly translating the type hierarchy (Section 3.3.4) and different structures of terms (Section 3.3.5), is heavily influenced by the "forward" translation from KeY to Z3.

3.3.1 Proof Directions

Proof trees in KeY are built from the root towards the leaves: The proof obligation formula is decomposed into (possibly multiple) simpler formulas, splitting the proof by case distinctions if necessary, until finally each branch can be closed by one of the three closing rules shown in Section 2.1. In contrast to that, Z3 produces proofs in the opposite direction: It starts at the proof leaves with the user provided assertions, hypotheses, and several axiomatic rules, combining them step by step until finally a refutation is produced. When trying to replay Z3 proofs in KeY, this difference in proof direction is quite challenging:

In a first attempt, one could try to replay a proof by adding all formulas which Z3 had available as assertions to KeY, and then applying rules in the same way as Z3 does to combine the assertions until a refutation is reached. However, Z3's quantifier introduction rule (Figure 2.4) can not be replayed this way, since it would require to add leaves with formulas containing free variables. However, KeY does not have a notion of formulas with free variables. It would be possible to close such formulas by adding universal quantifiers, though, one would introduce many additional quantifiers (for every leaf formula) that is above a `quant-intro` rule in the Z3 proof.

The approach taken in this work replays a Z3 proof starting from its root and working towards the leaves. However, to match Z3's inference direction, we locally invert the proof direction of KeY with the help of its *cut* rule. Figure 3.2 shows the generic scheme used to replay most rules. For a rule with premises $\phi_1, \ldots, \phi_n$ and conclusion ψ, the following

$$\frac{\Gamma_1 \vdash \phi_1 \quad \dots \quad \Gamma_n \vdash \phi_n}{\Gamma_1, \dots, \Gamma_n \vdash \psi}\ \texttt{rulename}$$

$$\Big\downarrow \text{ is replayed as}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{L}{\Gamma_1,\dots,\Gamma_n,\phi_1,\dots,\phi_n \vdash \psi}}{\vdots}\ \mathit{andLeft}}{\Gamma_1,\dots,\Gamma_n,\phi_1\land\dots\land\phi_n\vdash\psi}\ \mathit{andLeft} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{R_1}{\Gamma_1\vdash\phi_1}\ \dots\ \dfrac{R_n}{\Gamma_n\vdash\phi_n}}{\vdots}\ \mathit{andRight}}{\Gamma_1,\dots,\Gamma_n\vdash\phi_1\land\dots\land\phi_n}\ \mathit{andRight}}{\Gamma_1,\dots,\Gamma_n\vdash\phi_1\land\dots\land\phi_n,\psi}\ \mathit{hideRight}}{\Gamma_1,\dots,\Gamma_n\vdash\psi}\ \mathit{cut}$$

Figure 3.2: Generic replay scheme for most rules

replay steps are applied: First, the conjunction of all premises is “added” to the sequent using KeY’s *cut* rule. On the left branch, we apply *andLeft* multiple times to split the cut formula. L denotes a sub-tree of the proof where we have to justify that the conclusion ψ really follows from the premises $\phi_1, \dots, \phi_n$ of the rule. The concrete steps depend on the rule that is replayed.

In the right branch, the validity of the premises has to be shown. Therefore, the conclusion of the original rule ψ is hidden (it is irrelevant here) and afterwards the conjunction is split into the original premises. Replay continues in the sub-proofs $R_1, \dots, R_n$, where the premises of the original rule are justified.

3.3.2 Replacing Equisatisfiability

In Z3 proofs, the equisatisfiability relation $\sim$ may occur in proofs (for example in the skolemization rules, see Section 2.2.1). In KeY, it has no direct counterpart. Therefore, we follow the approach of (Böhme, 2012) and replace equisatisfiability by equivalence. This is possible if the skolemization rules are replayed in a way that their conclusion is an equivalence rather than an equisatisfiability, which is described in the next section. Doing so one can prove by induction that equisatisfiability can be removed: The only rules other introducing it besides the skolem rules are refl and iff$\sim$, where the formulas are also equivalent. All other rules contain equisatisfiability only if their premises contain it also. Therefore we are allowed to replace equisatisfiability $\sim$ by equivalence $\leftrightarrow$ in all formulas.

3.3.3 Skolemization

Replay of the skolemization rules provides multiple challenges. In Z3, a term may contain free variables. Note that in newer Z3 versions, these variables are bound by the `proof-bind`

rule respectively the `lambda` binder in a whole subtree of the proof, which however is semantically the same as if they were free. Free variables can be considered implicitly bound by a universal quantifier due to Z3's notion of equisatisfiability (see Section 3.2). When applying the skolem rule to a term with free variables, the newly introduced skolem symbol may depend on them. Therefore, the fresh skolem symbol must be a function of these free variables.

In contrast to that, KeY has no notion of free variables in formulas. The available skolemization rules (*allLeft*/*exRight*, see Figure 2.4) can only be applied if their quantifiers are top level in the formula (that is, surrounded existential/universal quantifiers have to be instantiated/skolemized first).

Another obstacle is a consequence of the different proof directions of Z3 and KeY: In Z3's proof format, a rule is of the form `(rule-name premise_0 ... premise_n conclusion)`, where `conclusion` is a formula and the premises can be either formulas or proofs. During replay of a Z3 rule in KeY, premises and conclusion must be translated to KeY formulas. However, since replay is performed in the opposite direction as Z3 constructs the proof, during replay, skolem symbols are used before they are introduced (the skolem rule is a closing rule occurring at a leaf of the proof, see Section 2.2.1).

Luckily, we can use the Hilbert choice operator ϵ to solve both problems at once. Intuitively, the term $\epsilon x.\, \phi(x)$ denotes some x, if there exists one, such that the formula $\phi(x)$ holds. Using this binding operator, we can translate skolem symbols as follows: If a skolem symbol s is encountered in a formula $\psi(s)$ during replay, we search the proof tree from there towards the leaves until we find the corresponding skolemization rule where s is introduced[2]. We then construct the term $\epsilon x.\, \phi(x)$ and replace it for s (right hand side). In addition, we are now allowed to replace equisatisfiability by equivalence, as explained in the previous section:

$$
\dfrac{\dots \qquad \dfrac{\dfrac{}{\Gamma \vdash (\exists x.\, \phi(x)) \sim \phi(s)}\,\texttt{sk}}{\vdots}\,\dots \qquad \dots}{\Gamma \vdash \psi(s)}\,\dots
\qquad
\dfrac{\dots \qquad \dfrac{\dfrac{}{\Gamma \vdash (\exists x.\, \phi(x)) \leftrightarrow \phi(\epsilon x.\, \phi(x))}\,\texttt{sk}}{\vdots}\,\dots \qquad \dots}{\Gamma \vdash \psi(\epsilon x.\, \phi(x))}\,\dots
$$

However, for skolem functions it gets more complicated: If a skolem function $s(t)$ occurs depending on any term t, the skolemization occurs in a subtree of the proof with free variables (marked as subtree A in the example below). Note that such free variables in Z3 are usually introduced by the quant-intro rule as shown. In the example, χ and ρ are arbitrary formulas, Q stands for either $\forall$ or $\exists$, t is any term, and y is a free variable in

[2] Without loss of generality, we only consider the existential variant of the `sk` rule here. The variant with universal quantifier is analogue.

subtree A:

$$\dfrac{\dfrac{\cdots \qquad \dfrac{\dfrac{}{\Gamma \vdash (\exists x.\ \phi(x,y)) \sim \phi(s(y))}\ \mathtt{sk}}{\vdots}\ \cdots \qquad \cdots}{\dfrac{\Gamma \vdash \chi(y) \sim \rho(y)}{\Gamma \vdash Qy.\chi(y) \sim Qy.\rho(y)}\ \mathtt{quant\text{-}intro}}\ \cdots \quad \text{subtree A} \uparrow \qquad \cdots \qquad \vdots \qquad \cdots}{\Gamma \vdash \psi(s(t))}\ \cdots$$

For translation, we again collect the epsilon term from the `sk` rule. This ϵ-term effectively denotes a skolem function, which is notable since it is possible to construct skolem functions for KeY this way. When inserting this skolem function into the original formula ψ, the free variable y has to be replaced by the term t, which is denoted by the substitution $[y/t]$:

$$\dfrac{\dfrac{\cdots \qquad \dfrac{\dfrac{}{\Gamma \vdash (\exists x.\ \phi(x,y)) \leftrightarrow \phi(\epsilon x.\ \phi(x,y))}\ \mathtt{sk}}{\vdots}\ \cdots \qquad \cdots}{\dfrac{\Gamma \vdash \chi(y) \leftrightarrow \rho(y)}{\Gamma \vdash Qy.\chi(y) \leftrightarrow Qy.\rho(y)}\ \mathtt{quant\text{-}intro}}\ \cdots \quad \text{subtree A} \uparrow \qquad \cdots \qquad \vdots \qquad \cdots}{\Gamma \vdash \psi([y/t]\epsilon x.\ \phi(x,y))}\ \cdots$$

For the Hilbert choice operator, there are multiple possible semantics. A selection of them is presented by Giese and Ahrendt (1999). In KeY, binders are extensional in general, that is, simplification and replacement of a term by an equivalent one inside a binder do not influence the valuation of the binder term. According to Giese and Ahrendt (1999), it is probably not possible to provide rules for a complete calculus with extensional semantics of epsilon terms which are well-suited for automatic theorem proving. Therefore, we impose additional restrictions for epsilon terms: The only occurrences of the choice operator we allow are those introduced during replay by our technique. With this restriction, for handling epsilon terms we only need the newly added rule *epsDefAdd*, used during replay of the `sk` rule (see Paragraph 3.5.1) to justify the introduction of the choice operator. Note that this axiom describes the relation between existential quantifier and choice operator:

$$\dfrac{}{\vdash \exists x.\ \phi(x) \leftrightarrow \phi(\epsilon x.\ \phi(x))}\ \mathit{epsDefAdd}$$

Notably, epsilon terms always occur inside the formula ϕ that is also the condition of the choice operator. Everywhere else, epsilon terms can be treated as atomically. Note that without this restriction, the calculus would be incomplete. An example formula that could not be proven can be found in Appendix A.1.

Finally, there is a drawback when replacing skolem symbols as described: Due to the introduced epsilon-terms the formulas grow heavily in size. This is a problem yet to be solved for the future.

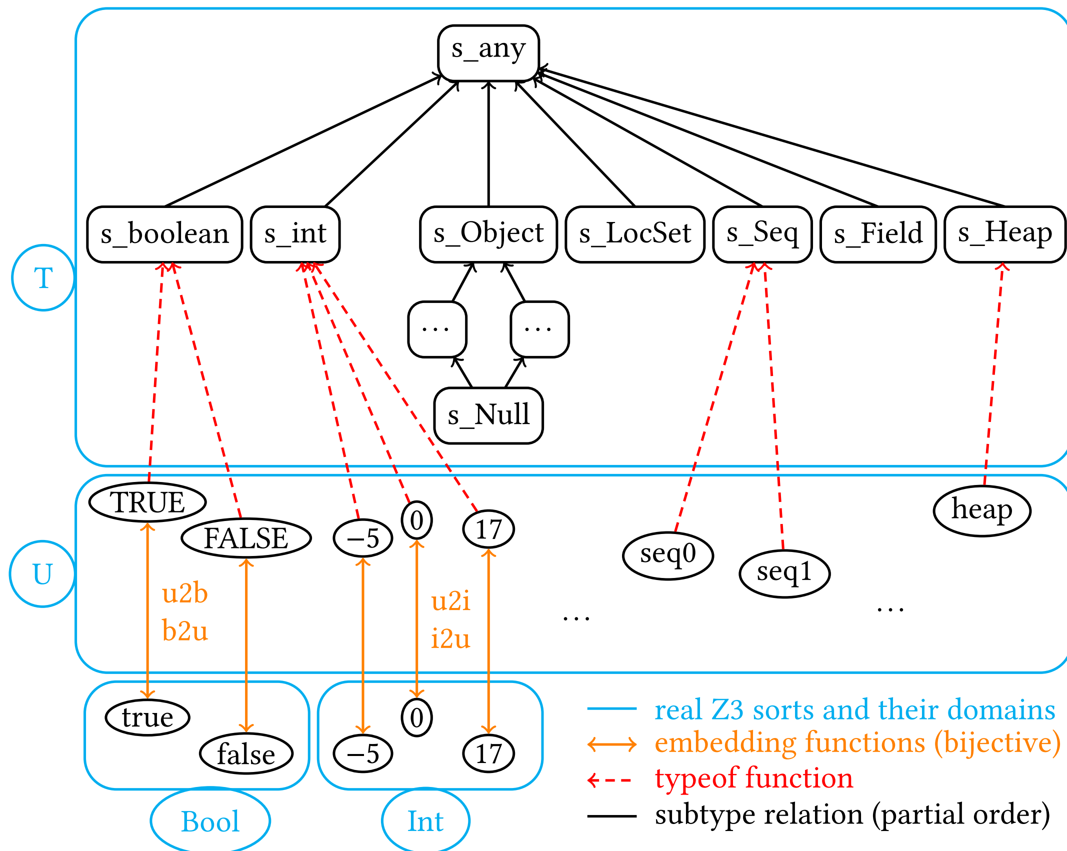

Figure 3.3: The type hierarchy after translation to Z3. For brevity, the "sort_" prefix is shortened to "s_".

3.3.4 Translating the Type Hierarchy

As explained in Section 2.1, KeY uses Java dynamic logic (JavaDL) with a type hierarchy as shown in Figure 2.1. In this type hierarchy, every sort is a sub-sort of *Any*, including *boolean* and *int*. In contrast to that, *Bool* and *Int* are top level sorts in Z3. Therefore, some effort is necessary to retain KeY's type hierarchy while still being able to benefit from Z3's built-in and very fast reasoning strategies for *Bool* and *Int*. The following technique is taken from (Leino and Rümmer, 2010) and has already been implemented in KeY prior to this work. However, the existing implementation was using features unsuitable for replay. Therefore, we present an altered type hierarchy translation that permits us to translate all terms back to KeY during proof replay.

The original translation technique. Core idea of the technique is to encode each KeY sort s as individual "sort_s" of the new Z3 sort T, while all instances of s are embedded into the new Z3 sort U. To establish the connection between the "sort" and their values, a function $typeof : U \to T$ is defined. The embedded sorts are connected via the predicate $subtype : T \times T \to Bool$, which is axiomatized to form a partial order, that is, a reflexive, transitive, and antisymmetric relation.

subtype relation is a partial order:

reflexive:

$\forall t : T.\ subtype(t, t)$

antisymmetric:

$\forall t_1 : T,\ t_2 : T.\ subtype(t_1, t_2) \wedge subtype(t_2, t_1) \rightarrow t_1 = t_2$

transitive:

$\forall t_1 : T,\ t_2 : T,\ t_3 : T.\ subtype(t_1, t_2) \wedge subtype(t_2, t_3) \rightarrow subtype(t_1, t_3)$

cast:

$\forall u : U,\ t : T.\ subtype(typeof(cast(t, u)), t)$

$\forall u : U,\ t : T.\ subtype(typeof(u), t) \rightarrow cast(t, u) = u$

for each sort s with n child sorts $c_1, \ldots, c_n$:

sort "definition":

$instanceof(t, u)$

for every subsort c_i:

$subtype(c_i, s)$

Figure 3.4: Axioms of the original type hierarchy translation

Besides the two Z3 sorts T and U, there are the original Z3 sorts *Bool* and *Int*, both with their own domain. The embedded *int* values are connected by bijective functions $u2i : U \rightarrow Int$ and $i2u : Int \rightarrow U$ to their "real" Z3 counterparts, and equally for boolean. An overview of the resulting type hierarchy in Z3 is given in Figure 3.3, the axioms are shown in Figure 3.4.

Typeguards for quantifiers. The translation as described above has a problem when translating quantifiers: Values of T, for example sort_any, are no real sorts that can be quantified over in Z3. Quantification now has to be done over U, which usually contains much more than the domain of the original sort. The solution for this problem is to introduce typeguards for quantifiers (also described in (Leino and Rümmer, 2010)). To give an example, the formula

$$\forall i : int.\ i > 0$$

from KeY will be translated to the following Z3 formula:

$$\forall i : U.\ instanceof(int, i) \rightarrow i2u(i) > i2u(0)$$

In addition to applications of the embedding function $i2u$, we can clearly see the typeguard $instanceof(int, i)$ here, connected by an implication to the original formula. For typeguards of existential quantifiers, the junctor $\wedge$ is used. Adding typeguards that way ensures that we do not introduce inconsistent formulas by quantifying over the wrong domain.

Type Hierarchy Translation for replay. First of all, for translating quantifier terms back to KeY, one has to know the original sort. However, if given a Z3 term such as

$$\forall u : U.\ instanceof(t, u) \rightarrow \phi(u)$$

for any formula ϕ, we do not know if the subterm $instanceof(t, u)$ is a typeguard or was already present in the original KeY formula prior to its translation to Z3. To be able to distinguish both cases reliably, we introduce an additional predicate $typeguard : T \times U \rightarrow Bool$ that is semantically equivalent to $instanceof$: We establish this connection via the following additional axiom schema (for every type $t \in T$):

$$\forall u : U.\ typeguard(t, u) \leftrightarrow instanceof(t, u),$$

Now we exclusively use the new *typeguard* predicate to represent typeguards. During replay, we can reliably extract the quantified types from them.

As already mentioned, the technique as described above was implemented in KeY using two features that prevent translation of Z3 terms backs to KeY, which is crucial for replay. In the first place the translation uses the function $typeof : U \rightarrow T$ and the predicate $subtype : U \times U \rightarrow Bool$, which both have no direct counterpart in the logic used by KeY. Since the complete type hierarchy has been axiomatized using these, we had to find an alternative equivalent axiomatization without them. The new axioms presented in Figure 3.5 use only the predicates $exactinstanceof : T \times U \rightarrow Bool$ and $instanceof : T \times U \rightarrow Bool$, which have direct counterparts in KeY.

The second problem is quantification over (embedded) types. It can be solved by rolling out the relevant axioms for every sort. This applies to the two cast axioms:

$$\forall t : T.\ \forall u : U.\ instanceof(t, cast_t(u))$$
$$\forall t : T.\ \forall u : U.\ instanceof(t, u) \rightarrow cast_t(u) = u$$

Special care has to be taken for the Null sort (containing only the single individual *null*): It is subsort of every Object sort. While the subtrees pending from two children c_i and c_j of an object sort are mostly disjoint, this is not the case for the null sort: It is indeed contained by both subtrees. Therefore, we add a special axiom for every pair of subsorts (c_i, c_j) with $(i \neq j)$ of every object sort:

$$\forall u : U.\ instanceof(c_i, u) \land instanceof(c_j, u) \rightarrow null$$

This set of axioms can be further optimized by omitting symmetrical cases: For example, the axiom for (c_i, c_j) is equivalent to that for (c_j, c_i).

Counting the needed axioms, we can see that our new translation needs $O(n^3)$ axioms for n sorts, whereas the original translation needed only $O(n^2)$. The detailed calculation can be found in Appendix A.2. While $O(n^3)$ in comparison to $O(n^2)$ sounds bad, we argue that it is a worst case estimation that most likely does not happen in reality: The worst case is if the type hierarchy is very broad and flat, i.e. all Java classes are direct subclasses Object. This is rarely the case for large sets of classes. The inequations displayed above are very coarse, usually $s_{max} \ll n$ and also $s_i \ll s_{max}$ for most s_i. In addition, all generated axioms can be easily triggered via a pattern and do not lead to much performance loss, at least not for the amount of classes that occurred during the evaluation of this work. In practice, the enlarged number of axioms should be no problem for Z3.

for each sort s **with** n **child sorts** $c_1, \ldots, c_n$**:**
sort "definition":
$\forall u : U.\ exactInstanceof(s, u) \lor instanceof(c_1, u) \lor \ldots \lor instanceof(c_n, u) \rightarrow instanceof(s, u)$
cast:
$\forall u : U.\ instanceof(s, cast(s, u))$
$\forall u : U.\ instanceof(s, u) \rightarrow cast(s, u) = u$
typeguard:
$\forall u : U.\ typeguard(s, u) \leftrightarrow instanceof(s, u)$
for every subsort c_i:
at most one type:
$\forall u : U.\ \neg exactInstanceof(s, u) \lor \neg instanceof(c_i, u)$
additional axiom for sort_any only:
for every subsort pair (c_i, c_j), subsorts are disjoint:
$\forall u : U.\ \neg instanceof(c_i, u) \lor \neg instanceof(c_j, u) \qquad (i \neq j)$
additional axioms for object sorts only:
for every subsort pair (c_i, c_j) subsorts are disjoint (except null):
$\forall u : U.\ instanceof(c_i, u) \land instanceof(c_j, u) \rightarrow u = null \qquad (i \neq j)$

Figure 3.5: Axioms of the modified type hierarchy

3.3.5 Structurally Different Terms

A challenge for replay is that there are cases where the structure of terms in Z3 differs from their translation to KeY. There are multiple reasons for this: First of all, in Z3 the boolean connectors "and" and "or" are polyadic, whereas in KeY they are only binary functions. While in theory the solution is easy (unfolding a single polyadic function application to multiple binary ones), it sometimes makes position calculations very difficult on an engineering level. An example where this can be seen is the quantifier instantiation rule.

The second reason is similar to the first: In contrast to KeY, in Z3 a single quantifier may bind multiple variables. Hence during translation, Z3 quantifiers have to be unfolded to multiple identical quantifiers with one bound variable each. Consequently, a single Z3 quantifier rule in general has to be translated to multiple applications of KeY's quantifier rules.

The third reason is the forward translation (from KeY to Z3) using typeguards as described in Paragraph 3.3.4. Intuitively, one would like to translate a Z3 formula with typeguard back to its original KeY formula. However, doing so the formulas in the Z3 proof diverge from the formulas actually available in KeY during replay. The solution is to translate the typeguards back to KeY (using the instanceof function family). A small drawback is that it inflates the sequents in KeY with terms that were not present in the original problem, hence reducing readability.

Finally, there is small semantic difference between Z3 and KeY: While in the former predicates are just functions with boolean result type, the latter explicitly distinguishes between formulas and terms of boolean type (in the implementation this is reflected by the additional type "Formula"). For the translation of a Z3 term back to KeY, this means it is

sometimes necessary to add an additional "= TRUE" term, for example $instanceof(int, 4)$ in Z3 translates back to $instanceof(int, 4) = TRUE$ in KeY, since *instanceof* is boolean in KeY. In addition, KeY has the constant formulas "true/false" as well as the constants "TRUE/FALSE" of boolean sort. Also, it has to be taken care to translate Z3's equality to either = or $\leftrightarrow$, depending on the type of the *translated* arguments.

To sum up, these structural differences of terms provide engineering challenges and decrease readability in KeY, but do not generally prevent replay.

3.4 Additional Rules for KeY as Abbreviations

It turns out that it is beneficial for replay to add some rules to KeY. All of them are only abbreviations for multiple rule applications of existing rules. Their soundness can be proven in KeY easily, the proofs are found by KeY's auto mode within a second. However, for replay they allow to treat certain cases uniformly. It should be noted that these rules are not made available to KeY's normal built-in proof search strategy to avoid changing anything. Instead, they are only applied "manually" during replay or made available to certain replay-only strategies.
The rule *equivSymm* is similar to *eqSymm* presented in Paragraph 2.1, but for $\leftrightarrow$ instead of =.

$$\phi \leftrightarrow \psi \rightsquigarrow \psi \leftrightarrow \phi \qquad \textit{equivSymm}$$

The rules *notElimLeft/Right* introduce an additional negation in front of a formula, thereby moving it to the other side of the sequent. It is on purpose that these rules are not available to KeY's auto mode, since they conflict with the usual proof direction in KeY: Instead of simplifying the formula, they make it more complex by introducing additional operations. Nonetheless they are useful for uniform handling of literals for example during replay of the unit-resolution rule.

$$\frac{\Gamma \vdash \neg\phi}{\Gamma, \phi \vdash}\ \textit{notElimLeft} \qquad\qquad \frac{\Gamma, \neg\phi \vdash}{\Gamma \vdash \phi}\ \textit{notElimRight}$$

Finally, the additional rule *distribute_all_equiv* provides a significant abbreviation for multiple rule applications of KeY's standard rules. It is tailored to the `quant-intro` rule (and an identical schema that may occur during replay of `nnf-pos`/`nnf-neg`) and weakens the proposition by distributing the universal quantifier over equivalence. Using this rule, it is possible to avoid a case split and afterwards on each of the branches a skolemization plus two quantifier instantiations. KeY's auto mode usually does not find the necessary rule applications within a considerable amount of steps (more than 1000 in our examples). Therefore, the rule is made available to a specialized proof search macro when replaying `nnf-pos`, `nnf-neg`, and `quant-intro`.

$$\frac{\Gamma, \forall x.\ \phi(x) \leftrightarrow \forall x.\ \psi(x) \vdash}{\Gamma, \forall x.\ (\phi(x) \leftrightarrow \psi(x)) \vdash}\ \textit{distribute_all_equiv}$$

3.5 Translating Z3's Proof Rules

In this section, we describe for every Z3 rule how it is translated to a tree of rule applications in KeY. As in the previous chapter, we present the rules in a sequent calculus style and assume for readability, that all formulas which could occur on the right side of the turnstile in a sequent are moved into Γ by negating them. On the left side (or on top for long rules), the Z3 rules are presented, while their replay schema is displayed on the right (or below, if it does not fit into the line). For descriptions of the Z3 rules, see Section 2.2.1, while KeY rules are explained in Paragraph 2.1.

A general question that can be asked is why one would not add a new rule in KeY for every Z3 rule and use these in replay. However, most proof rules of Z3 denote rule schemata (for example because conjunction and disjunction are polyadic in Z3) and can thus not be represented by a single KeY rule.

3.5.1 Rules Replayed Manually

The rules in this section can be replayed manually, that is, predetermined by program code without actual proof search. Due to the polyadic nature of some functions, quantifiers, and proof rules of Z3, the concrete number of steps often depends on the arity of the replayed construct.

true-axiom This is the most simple proof rule, a proof of `true`. Of course it can be directly mapped to the use of KeY's *closeTrue* rule:

$$\frac{}{\Gamma \vdash \top}\,\texttt{true-axiom} \qquad \rightsquigarrow \qquad \frac{}{\Gamma \vdash \top}\,\mathit{closeTrue}$$

iff-true The `iff-true` rule also maps straightforward to a single rule application in KeY:

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \leftrightarrow \top}\,\texttt{iff-true} \qquad \rightsquigarrow \qquad \frac{\Gamma \vdash P}{\Gamma \vdash P \leftrightarrow \top}\,\mathit{concrete_eq_3}$$

iff-false Quite similar is the `iff-false` rule:

$$\frac{\Gamma \vdash \neg P}{\Gamma \vdash P \leftrightarrow \bot}\,\texttt{iff-false} \qquad \rightsquigarrow \qquad \frac{\Gamma \vdash \neg P}{\Gamma \vdash P \leftrightarrow \bot}\,\mathit{concrete_eq_4}$$

iff∼ For the rule deducing equisatisfiability from equivalence, nothing has to be done, since we replace all occurrences of equisatisfiability by equivalences earlier.

$$\frac{\Gamma \vdash P_1 \leftrightarrow P_2}{\Gamma \vdash P_1 \sim P_2}\,\texttt{iff}\sim \qquad \rightsquigarrow \qquad \frac{\Gamma \vdash P_1 \leftrightarrow P_2}{\Gamma \vdash P_1 \leftrightarrow P_2}$$

and-elim The and-elim rule selects a single literal from a conjunction. Beginning with this rule, the replay starts to get slightly more involved: It is not possible to directly

translate the Z3 rule to a single KeY rule. This is mainly due to the deconstructive nature of KeY's inference rules in contrast to the constructive ones of Z3.

$$\frac{\Gamma \vdash L_1 \wedge \ldots \wedge L_n}{\Gamma \vdash L_i}\ \texttt{and-elim}$$

We replay this Z3 rule using the generic scheme from Figure 3.2: The conclusion of the Z3 rule is introduced to the sequent in KeY by case distinction with the *cut* rule. On the right branch we can now hide the original conclusion $\neg L_i$, since we know that it is irrelevant here. The sequent of this branch now is equal to the premise of the original Z3 rule. Further replay of Z3 rules continues here. In the left branch we have to justify the soundness of the cut. For the `and-elim` rule this step is rather simple: We apply $n-1$ times KeY's *andLeft* rule, which then leads to the literals L_1 to L_n occurring as individual formulas in the antecedent. Finally, the branch can be closed, since the literal L_i is present in antecedent and succedent:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, L_1, \ldots, L_n \vdash L_i}\ \textit{close}}{\vdots}\ \textit{andLeft}}{\Gamma, L_1 \wedge \ldots \wedge L_n \vdash L_i}\ \textit{andLeft} \qquad \dfrac{\Gamma \vdash L_1 \wedge \ldots \wedge L_n}{\Gamma \vdash L_1 \wedge \ldots \wedge L_n, L_i}\ \textit{hideRight}}{\Gamma \vdash L_i}\ \textit{cut}$$

not-or-elim Similar to `and-elim`, this rule weakens a proposition by selecting a single literal from it:

$$\frac{\Gamma \vdash \neg(L_1 \vee \ldots \vee L_n)}{\Gamma \vdash \neg L_i}\ \texttt{not-or-elim}$$

Replay is very similar to that of `and-elim`, except that we split the formula with top level disjunctions on the right side (*orRight*) and have two additional steps (*notLeft*/*notRight*) to dispose the negations. Again, the right branch ends with the premise of the original Z3 rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, L_i \vdash L_1, \ldots, L_n}\ \textit{close}}{\Gamma \vdash L_1, \ldots, L_n, \neg L_i}\ \textit{notRight}}{\vdots}\ \textit{orRight}}{\Gamma \vdash L_1 \vee \ldots \vee L_n, \neg L_i}\ \textit{orRight}}{\Gamma, \neg(L_1 \vee \ldots \vee L_n) \vdash \neg L_i}\ \textit{notLeft} \qquad \dfrac{\Gamma \vdash \neg(L_1 \vee \ldots \vee L_n)}{\Gamma \vdash \neg(L_1 \vee \ldots \vee L_n), \neg L_i}\ \textit{hideRight}}{\Gamma \vdash \neg L_i}\ \textit{cut}$$

asserted The asserted rule in Z3 looks as follows:

$$\frac{}{\Gamma, P \vdash P}\ \texttt{asserted}$$

Closing is allowed since the proposition P to prove occurs in the antecedent. For Z3 this means that P has been asserted in the input, i.e. P is either an assertion that stems from

the translation of a formula of the original KeY sequent, or an axiom (for example from the type hierarchy). For replaying the rule, we have to distinguish both cases: If the assertion originates from the sequent, replay is trivial:

$$\dfrac{}{\Gamma, P \vdash P}\,\mathit{close}$$

However, if P is an axiom, it is not directly present in the KeY sequent. In this case, we have to rely on KeY's automatic proof search to find a proof for P. This is always possible using the built-in taclets of KeY. We argue that it is not very difficult for KeY to find the necessary rule applications, since the search space is very limited.

hypothesis This rule introduces a new hypothesis into the proof. While P can be any formula, it has to be discharged using the `lemma` rule later on this branch. The lemma rule adds P to the context on the left hand side of the sequent.

$$\dfrac{}{\Gamma, P \vdash P}\,\texttt{hypothesis} \qquad \rightsquigarrow \qquad \dfrac{}{\Gamma, P \vdash P}\,\mathit{close}$$

lemma The lemma rule states the following: If we can deduce a refutation from n lemmata, at least one of them must be invalid:

$$\dfrac{\Gamma, L_1, \ldots, L_n \vdash \bot}{\Gamma \setminus \{L_1, \ldots, L_n\} \vdash \neg L_1 \lor \ldots \lor \neg L_n}\,\texttt{lemma}$$

We can see that this adds (possibly multiple) lemmata to the context on the left side in the antecedent, thus it is the counterpart of the hypothesis rule. Due to its nature with n lemmata, considerable amount of work has to be done in splitting the formulas (*orRight*, *andLeft*) and eliminating negations by moving formulas to the other side of the sequent (*notLeft*, *notRight*):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, L_1, \ldots, L_n \vdash \top}\,\mathit{closeTrue}}{\Gamma, L_1, \ldots, L_n \vdash \top \land \ldots \land \top}\,\mathit{simplify}}{\vdots}\,\mathit{replace_known_right}}{\Gamma, L_1, \ldots, L_n \vdash L_1 \land \ldots \land L_n}\,\mathit{replace_known_right}}{\vdots}\,\mathit{notRight}}{\Gamma \vdash L_1 \land \ldots \land L_n, \neg L_1, \ldots, \neg L_n}\,\mathit{notRight}}{\vdots}\,\mathit{orRight}}{\Gamma \vdash L_1 \land \ldots \land L_n, \neg L_1 \lor \ldots \lor \neg L_n}\,\mathit{orRight}}{\Gamma, \neg(L_1 \land \ldots \land L_n) \vdash \neg L_1 \lor \ldots \lor \neg L_n}\,\mathit{notLeft} \quad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma, L_1, \ldots, L_n \vdash}{\vdots}\,\mathit{andLeft}}{\Gamma, L_1 \land \ldots \land L_n \vdash}\,\mathit{andLeft}}{\Gamma \vdash \neg(L_1 \land \ldots \land L_n)}\,\mathit{notRight}}{\Gamma \vdash \neg(L_1 \land \ldots \land L_n), \neg L_1 \lor \ldots \lor \neg L_n}\,\mathit{hideRight}}{\Gamma \vdash \neg L_1 \lor \ldots \lor \neg L_n}\,\mathit{cut}$$

While each of these rule applications is very simple on its own, the proof tree necessary for just the replay of a single lemma rule already gives an idea about the complexity of the proof resulting from the replay. This complexity gain is owed to the nature of the rules:

In Z3, proof rules often are axiom schemas, whereas the concrete axiom depends on the number n of subterms. In KeY, on the other hand, rules are single axioms and can only reason about a constant number of terms. This is a general challenge (on an engineering level), that we can see in the subsequent rules as well.

unit-resolution One of the core rules of Z3 is unit-resolution, created by the SAT solver in Z3.

$$\frac{\Gamma \vdash \bigvee_{i \in I_s} L_i \qquad \langle i \in I_s \mid \Gamma_i \vdash \neg L_i \rangle}{\Gamma \cup \bigcup_{i \in I_s} \Gamma_i \vdash \bigvee_{i \in I \setminus I_s} L_i}\ \texttt{unit-resolution} \qquad I = \{1, \dots, n\}, I_s \subseteq I$$

We start with a special case: The rule often is used as last rule at the root of a proof, that is $I_s = I = \{1, \dots, n\}$, and thus the conclusion is $\bot$:

$$\frac{\Gamma \vdash L_1 \wedge \dots \wedge L_n \quad \Gamma_1 \vdash \neg L_1 \quad \dots \quad \Gamma_n \vdash \neg L_n}{\Gamma, \Gamma_1, \dots, \Gamma_n \vdash \bot}\ \texttt{unit-resolution}$$

The replay is done as shown below. The dots on the right hand side of the cut indicate that this branch is replayed exactly as shown in the generic replay scheme in Figure 3.2: The original conclusion is hidden, and the cut formula is split into multiple branches, which are the original premises of the `unit-resolution` rule. Note that for easier readability, we combine multiple subsequent applications of the same rule and indicate this by a star after the rule name.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \Gamma_1, \dots, \Gamma_n, \bot \vdash L_1, \dots, L_n}\ \textit{closeFalse}}{\Gamma, \Gamma_1, \dots, \Gamma_n, \bot \vee \dots \vee \bot \vdash L_1, \dots, L_n}\ \textit{simplify*}}{\Gamma, \Gamma_1, \dots, \Gamma_n, L_1 \vee \dots \vee L_n \vdash L_1, \dots, L_n}\ \textit{replace_known_left*}}{\Gamma, \Gamma_1, \dots, \Gamma_n, L_1 \vee \dots \vee L_n, \neg L_1, \dots, \neg L_n \vdash}\ \textit{notLeft*}}{\Gamma, \Gamma_1, \dots, \Gamma_n, (L_1 \vee \dots \vee L_n) \wedge \neg L_1 \wedge \dots \wedge \neg L_n \vdash}\ \textit{andLeft*} \qquad \dots}{\Gamma, \Gamma_1, \dots, \Gamma_n \vdash \bot}\ \textit{cut}$$

For the general case, where the succedent is $\bigvee_{i \in I_s} L_i$ for $I = \{1, \dots, n\}, I_s \subseteq I$, the same rules are applied (possibly less often) with one exception: the last step in the branch is *close*, since a clause containing some of the literals is left in the antecedent, which is exactly the formula to prove in the succedent.

mp, mp~ Replay of the modus ponens rule is relatively simple ($\rightsquigarrow$ denotes either $\rightarrow$, $\leftrightarrow$, or $\sim$). Note that `mp~` reduces to normal `mp` rule, since we replaced equisatisfiability as described in Paragraph 3.2.

$$\frac{\Gamma_1 \vdash \phi \quad \Gamma_2 \vdash \phi \rightsquigarrow \psi}{\Gamma_1, \Gamma_2 \vdash \psi}\ \texttt{mp}$$

We follow our generic replay scheme and add the premises of the rule (connected by $\wedge$) as a cut first. On the right branch, as always we hide the original conclusion and split into

the original premises, from where the further replay continues. On the left branch, we justify the modus ponens step by splitting the premises, replacing the formula ϕ in the (bi-)implication. After simplification, we can close the goal.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1, \Gamma_2, \phi, \psi \vdash \psi}\,\mathit{close}}{\Gamma_1, \Gamma_2, \phi, \top \rightsquigarrow \psi \vdash \psi}\,\mathit{simplify}}{\Gamma_1, \Gamma_2, \phi, \phi \rightsquigarrow \psi \vdash \psi}\,\mathit{replace_known_left}}{\Gamma_1, \Gamma_2, \phi \land (\phi \rightsquigarrow \psi) \vdash \psi}\,\mathit{andLeft} \quad \dfrac{\dfrac{\Gamma_1 \vdash \phi \quad \Gamma_2 \vdash \phi \rightsquigarrow \psi}{\Gamma_1, \Gamma_2 \vdash \phi \land (\phi \rightsquigarrow \psi)}\,\mathit{andRight}}{\Gamma_1, \Gamma_2 \vdash \psi, \phi \land (\phi \rightsquigarrow \psi)}\,\mathit{hideRight}}{\Gamma_1, \Gamma_2 \vdash \psi}\,\mathit{cut}$$

refl The reflexivity rule directly maps to either *eqClose* or *eq_eq*, depending on whether the relation is $=$ or $\leftrightarrow$. Again, $\sim$ has already been replaced by $\leftrightarrow$ earlier.

$$\dfrac{}{\Gamma \vdash t = t}\,\texttt{refl} \quad \rightsquigarrow \quad \dfrac{}{\Gamma \vdash t = t}\,\mathit{eqClose}$$

$$\dfrac{}{\Gamma \vdash t \leftrightarrow t}\,\texttt{refl} \quad \rightsquigarrow \quad \dfrac{}{\Gamma \vdash t \leftrightarrow t}\,\mathit{eq_eq}$$

symm Very similar to reflexivity, the symmetry rule can directly be replayed by a single rule depending on the relation. Note that *equivSymm* is a new rule introduced specifically for the purpose of replay, see Section 3.4.

$$\dfrac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}\,\texttt{symm} \quad \rightsquigarrow \quad \dfrac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}\,\mathit{eqSymm}$$

$$\dfrac{\Gamma \vdash t_1 \leftrightarrow t_2}{\Gamma \vdash t_2 \leftrightarrow t_1}\,\texttt{symm} \quad \rightsquigarrow \quad \dfrac{\Gamma \vdash t_1 \leftrightarrow t_2}{\Gamma \vdash t_2 \leftrightarrow t_1}\,\mathit{equivSymm}$$

trans For replay of the transitivity rule, we apply the generic replay scheme from Figure 3.2.

$$\dfrac{\Gamma_1 \vdash t_1 \simeq t_2 \quad \Gamma_2 \vdash t_2 \simeq t_3}{\Gamma_1, \Gamma_2 \vdash t_1 \simeq t_3}\,\texttt{trans}$$

Since $\simeq$ can denote $=$ or $\leftrightarrow$ here ($\sim$ would have already been replaced by $\leftrightarrow$), the rule marked with A stands for either *applyEq* or *applyEquiv* during replay.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1, \Gamma_2, t_1 \simeq t_2, t_1 \simeq t_3 \vdash t_1 \simeq t_3}\,\mathit{close}}{\Gamma_1, \Gamma_2, t_1 \simeq t_2, t_2 \simeq t_3 \vdash t_1 \simeq t_3}\,A}{\Gamma_1, \Gamma_2, t_1 \simeq t_2 \land t_2 \simeq t_3 \vdash t_1 \simeq t_3}\,\mathit{andLeft} \quad \dfrac{\dfrac{\Gamma_1 \vdash t_1 \simeq t_2 \quad \Gamma_2 \vdash t_2 \simeq t_3}{\Gamma_1, \Gamma_2 \vdash t_1 \simeq t_2 \land t_2 \simeq t_3}\,\mathit{andRight}}{\Gamma_1, \Gamma_2 \vdash t_1 \simeq t_3, t_1 \simeq t_2 \land t_2 \simeq t_3}\,\mathit{hideRight}}{\Gamma_1, \Gamma_2 \vdash t_1 \simeq t_3}\,\mathit{cut}$$

quant-inst In the quantifier instantiation rule, $\overline{x} = (x_1, \ldots, x_n)$ denotes the bound variables and $\overline{t} = (t_1, \ldots, t_n)$ its instantiations.

$$\dfrac{}{\Gamma \vdash \neg(\forall \overline{x}.\ \phi(\overline{x})) \lor \phi(\overline{t})}\,\texttt{quant-inst}$$

Since quantifiers in KeY only bind a single variable, during term translation we have to unfold the single Z3 quantifier into n quantifiers in KeY. In the same way we have to apply KeY's instantiation rule *allLeft* multiple times, since it only can be used to instantiate a single quantifier. Also note that after each application of *allLeft*, the original quantified formula is still present on the sequent. However, for readability, we omit those and indicate those formulas only by the dots after Γ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \ldots, \forall x_2 \ldots \forall x_n.\ \phi(t_1, x_2, \ldots, x_n) \vdash \phi(t_1, \ldots, t_n)}\textit{allLeft}}{\Gamma, \forall x_1 \ldots \forall x_n.\ \phi(x_1, \ldots, x_n) \vdash \phi(t_1, \ldots, t_n)}\textit{allLeft}}{\Gamma \vdash \neg(\forall x_1 \ldots \forall x_n.\ \phi(x_1, \ldots, x_n)), \phi(t_1, \ldots, t_n)}\textit{notRight}}{\Gamma \vdash \neg(\forall x_1 \ldots \forall x_n.\ \phi(x_1, \ldots, x_n)) \lor \phi(t_1, \ldots, t_n)}\textit{orRight}}{}$$

where the top of the derivation is:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \ldots, \phi(t_1, \ldots, t_n) \vdash \phi(t_1, \ldots, t_n)}\textit{close}}{\Gamma, \ldots, \forall x_n.\ \phi(t_1, \ldots, t_{n-1}, x_n) \vdash \phi(t_1, \ldots, t_n)}\textit{allLeft}}{\vdots}\textit{allLeft}$$

proof-bind This is actually not a calculus rule, but a binder, binding "free" variables in the whole sub-tree from the rule application towards the leaves. It has only a single argument and is always succeeded by a lambda term. Note that the only rules that may introduce free variables are `quant-intro`, `nnf-pos`, and `nnf-neg`. Since KeY does not have a notion of free variables in formulas, we apply the following: The formula $\lambda\, x_1, \ldots, x_n.\ \phi(x_1, \ldots, x_n)$ is translated as $\forall\, x_1, \ldots, x_n.\ \phi(x_1, \ldots, x_n)$. Later, we apply KeY's skolemization rule *allRight* n times, thereby introducing n fresh skolem constants. Since skolem constants are implicitly universally quantified in KeY (see validity in Section 3.2), this is semantically equivalent to having "real" free variables.

quant-intro The `quant-intro` rule is always immediately followed by `proof-bind` and `lambda`, since the variables $\overline{x}$ in its conclusion are free. We follow the description from the section about the `proof-bind` rule and bind the free variables $\overline{x}$ by additional universal quantifiers. Fresh skolem constants are denoted by $\overline{s}$ here. The rule is available in two variants: Q can be either ∀ or ∃.

$$\dfrac{\Gamma \vdash \phi(\overline{x}) \sim \psi(\overline{x})}{\Gamma \vdash Q\overline{x}.\ \phi(\overline{x}) \sim Q\overline{x}.\ \psi(\overline{x})}\ \texttt{quant-intro}$$

As in the `quant-inst` rule, the single quantifier of Z3 has to be replaced by multiple quantifiers in KeY, we denote multiple steps of a rule by appending a star after the rule name and use dots to indicate the hidden irrelevant formulas that are generated. The existential variant of the rule is replayed as follows:

$$\dfrac{A \qquad \dfrac{\dfrac{\Gamma \vdash \phi(\overline{s}) \leftrightarrow \psi(\overline{s})}{\Gamma \vdash \forall\overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x})}\textit{allRight}}{\Gamma \vdash \exists\overline{x}.\ \phi(\overline{x}) \leftrightarrow \exists\overline{x}.\ \psi(\overline{x}), \forall\overline{x}.\phi(\overline{x}) \leftrightarrow \psi(\overline{x})}\textit{hideRight}}{\Gamma \vdash \exists\overline{x}.\ \phi(\overline{x}) \leftrightarrow \exists\overline{x}.\ \psi(\overline{x})}\textit{cut}$$

For reasons of space, the sub-tree denoted by A is given separately. It can be seen, that the two branches are perfectly symmetric with ϕ and ψ swapped:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \ldots, \psi(\overline{s}), \phi(\overline{s}) \vdash \psi(\overline{s}), \ldots}\,\textit{close}}{\Gamma, \ldots, \top \leftrightarrow \psi(\overline{s}), \phi(\overline{s}) \vdash \psi(\overline{s}), \ldots}\,\textit{simplify}}{\Gamma, \ldots, \phi(\overline{s}) \leftrightarrow \psi(\overline{s}), \phi(\overline{s}) \vdash \psi(\overline{s}), \ldots}\,\textit{repl_known_left}^*}{\Gamma, \ldots, \forall \overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x}), \phi(\overline{s}) \vdash \psi(\overline{s}), \ldots}\,\textit{allLeft}^*}{\Gamma, \ldots, \forall \overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x}), \phi(\overline{s}) \vdash \exists \overline{x}.\ \psi(\overline{x})}\,\textit{exRight}^*}{\Gamma, \forall \overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x}), \exists \overline{x}.\ \phi(\overline{x}) \vdash \exists \overline{x}.\ \psi(\overline{x})}\,\textit{exLeft}^* \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \ldots, \phi(\overline{s}), \phi(\overline{s}) \vdash \phi(\overline{s}), \ldots}\,\textit{close}}{\Gamma, \ldots, \psi(\overline{s}) \leftrightarrow \top, \psi(\overline{s}) \vdash \phi(\overline{s}), \ldots}\,\textit{simplify}}{\Gamma, \ldots, \phi(\overline{s}) \leftrightarrow \psi(\overline{s}), \psi(\overline{s}) \vdash \phi(\overline{s}), \ldots}\,\textit{repl_known_left}^*}{\Gamma, \ldots, \forall \overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x}), \psi(\overline{s}) \vdash \phi(\overline{s}), \ldots}\,\textit{allLeft}^*}{\Gamma, \ldots \forall \overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x}), \psi(\overline{s}) \vdash \exists \overline{x}.\ \phi(\overline{x})}\,\textit{exRight}^*}{\Gamma, \forall \overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x}), \exists \overline{x}.\ \psi(\overline{x}) \vdash \exists \overline{x}.\ \phi(\overline{x})}\,\textit{exLeft}^*}{\Gamma, \forall \overline{x}.\ \phi(\overline{x}) \leftrightarrow \psi(\overline{x}) \vdash \exists \overline{x}.\ \phi(\overline{x}) \leftrightarrow \exists \overline{x}.\ \psi(\overline{x})}\,\textit{equiv_right}$$

Finally, the variant of the rule with the universal quantifier replays in an analogous manner, with the only difference that instead of the rule sequence (*exLeft**, *exRight**) the sequence (*allRight**, *allLeft**) is used on both branches.

sk For replaying the two variants of the skolemization rule, we apply the changes described in Section 3.3.2 and Section 3.3.3: The Hilbert choice operator is used to represent skolem symbols (constants and functions), and equisatisfiability is replaced by equivalence.

With the rule additional *epsDefAdd* described in Section 3.4, replaying both variants of the `sk` rule is rather easy:

$$\dfrac{}{\Gamma \vdash (\exists \overline{x}.\ P(\overline{y}, \overline{x})) \leftrightarrow P(\overline{y}, \epsilon \overline{x}.\ P(\overline{y}, \overline{x}))}\ \mathtt{sk}_{\exists}$$

The existential variant of the skolemization rule is replayed as follows:

$$\dfrac{\dfrac{}{\Gamma, (\exists \overline{x}.\ P(\overline{y}, \overline{x})) \leftrightarrow P(\overline{y}, \epsilon \overline{x}.\ P(\overline{y}, \overline{x})) \vdash (\exists \overline{x}.\ P(\overline{y}, \overline{x})) \leftrightarrow P(\overline{y}, \epsilon \overline{x}.\ P(\overline{y}, \overline{x}))}\,\textit{close}}{\Gamma \vdash (\exists \overline{x}.\ P(\overline{y}, \overline{x})) \leftrightarrow P(\overline{y}, \epsilon \overline{x}.\ P(\overline{y}, \overline{x}))}\,\textit{epsDefAdd}$$

There is a second variant of the skolem rule, where a universal quantifier is present, but surrounded by a negation. By DeMorgan's Law, $\neg \forall x.\ \phi(x) \leftrightarrow \exists x.\ \neg \phi(x)$. Therefore, the rule is effectively the same as the existential variant, with the only difference of the negated P:

$$\dfrac{}{\Gamma \vdash \neg(\forall \overline{x}.\ P(\overline{y}, \overline{x})) \leftrightarrow \neg P(\overline{y}, \epsilon \overline{x}.\ \neg P(\overline{y}, \overline{x}))}\ \mathtt{sk}_{\neg\forall}$$

Replay is done as in the other skolem rule, with the addition of a single proof step to push the negation into the quantifier. Note that ϕ in the axiom schema *epsDefAdd* has to be instantiated with $\neg P(\overline{y}, \overline{x})$ here, while for $\mathrm{sk}_{\exists}$ only $P(\overline{y}, \overline{x})$ has to be used.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, (\exists \overline{x}.\ \neg P(\overline{y}, \overline{x})) \leftrightarrow \neg P(\overline{y}, \epsilon \overline{x}.\ \neg P(\overline{y}, \overline{x})) \vdash (\exists \overline{x}.\ \neg P(\overline{y}, \overline{x})) \leftrightarrow \neg P(\overline{y}, \epsilon \overline{x}.\ \neg P(\overline{y}, \overline{x}))}\,\textit{close}}{\Gamma, (\exists \overline{x}.\ \neg P(\overline{y}, \overline{x})) \leftrightarrow \neg P(\overline{y}, \epsilon \overline{x}.\ \neg P(\overline{y}, \overline{x})) \vdash \neg(\forall \overline{x}.\ P(\overline{y}, \overline{x})) \leftrightarrow \neg P(\overline{y}, \epsilon \overline{x}.\ \neg P(\overline{y}, \overline{x}))}\,\textit{nnf_notAll}}{\Gamma \vdash \neg(\forall \overline{x}.\ P(\overline{y}, \overline{x})) \leftrightarrow \neg P(\overline{y}, \epsilon \overline{x}.\ \neg P(\overline{y}, \overline{x}))}\,\textit{epsDefAdd}$$

elim-unused This rule is used by Z3 to eliminate variables that do not occur in the formula quantified over:

$$\dfrac{}{\Gamma \vdash \forall x_1, \ldots x_n, y_1, \ldots, y_m.\ \phi(x_1, \ldots x_n) \leftrightarrow \forall x_1, \ldots x_n.\ \phi(x_1, \ldots x_n)}\ \mathtt{elim\text{-}unused}$$

Replay in KeY is also quite simple, apart from the fact that the corresponding rule in KeY can only eliminate a single variable at once. Note that *eq_close* and *all-unused* are rewrite rules that work by locally rewriting terms.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \top}\,close_true}{\Gamma \vdash \forall x_1, \dots x_n.\ \phi(x_1, \dots x_n) \leftrightarrow \forall x_1, \dots x_n.\ \phi(x_1, \dots x_n)}\,\begin{matrix}eq_close\\ all\text{-}unused\end{matrix}}{\vdots}}{\Gamma \vdash \forall x_1, \dots x_n, y_1, \dots, y_{m-1}.\ \phi(x_1, \dots x_n) \leftrightarrow \forall x_1, \dots x_n.\ \phi(x_1, \dots x_n)}\,all\text{-}unused}{\Gamma \vdash \forall x_1, \dots x_n, y_1, \dots, y_m.\ \phi(x_1, \dots x_n) \leftrightarrow \forall x_1, \dots x_n.\ \phi(x_1, \dots x_n)}\,all\text{-}unused$$

3.5.2 Rules Replayed With Automatic Proof Search in KeY

For the rules in this section, KeY's built-in proof search strategy is used. Replay is done using the generic scheme as displayed in Figure 3.2, where KeY's built-in proof search is applied to the left branch L.

monotonicity For any congruence relation f, the following rule is available:

$$\dfrac{\Gamma_1 \vdash t_1 \simeq s_1 \qquad \dots \qquad \Gamma_n \vdash t_n \simeq s_n}{\Gamma_1, \dots, \Gamma_n \vdash f(t_1, \dots, t_n) \simeq f(s_1, \dots, s_n)}\,\texttt{monotonicity}$$

In theory we could close the branch L manually by applying all equalities, closing the equation, and finally closing with `true` on the right hand side of the sequent. However, during our experiments, we observed additional (undocumented) rewriting steps that may occur. Therefore, it is necessary to use KeY's automatic proof search.

commutativity In this rule, $\diamond$ stands for any commutative relation. Therefore, the replay has to rely on KeY's built-in proof search to find the proof.

$$\dfrac{\Gamma \vdash (t_1 \diamond t_2 \simeq s_1 \diamond s_2)}{\Gamma \vdash (t_2 \diamond t_1 \simeq s_2 \diamond s_1)}\,\texttt{commutativity}$$

def-axiom This rule contains schematic axioms for justifying conversion steps to Tseitin CNF form of a formula. Theoretically, one could collect all of those by experimenting (the documentation of Z3 lists at least 19 schemata, however, this list is known to be incomplete). For this work, we chose to use proof search in KeY for replay.

nnf-pos and nnf-neg These rules are used by Z3 to reason about conversion to negation normal form. As for `def-axiom`, there are multiple schemata used, which is why we rely on KeY to find a proof.

rewrite Before starting the "ping-pong game" between CDCL SAT solver and theory solvers, Z3 may use a simplifier to apply simplification steps to terms. The rewriting steps may be propositional or theory-specific and are presented as axioms. Examples for possible axioms are $\phi \vee \bot \leftrightarrow \phi$ or $x + 0 = x$. The following three cases of rewrite may occur, where always the top level symbol of t is interpreted ($\vee$ and $+$ in the example).

$$\frac{}{\Gamma \vdash t \simeq s}\ \texttt{rewrite}$$

There is no list of allowed rewrites, therefore, we have to rely on the proof search of KeY to justify the rewrite step. However, the assumption is that only rewrite steps for theories present in KeY occur. For example, for bit-vector simplifications KeY will clearly not be able to find a proof, since the theory of bit-vectors is not built-in.

th-lemma During the "ping-pong" between SAT solver and theory solvers, facts produced from the latter are introduced via the `th-lemma` rule. In contrast to the `rewrite` rule, Z3 *may* give some additional information about the used theory and the nature of the lemma (in our examples, more often no information was given). For example, `arith` may be given for the theory of arithmetic and `farkas` for Farkas' lemma or `triangle-eq` for $t_1 = t_2 \leftrightarrow (t_1 \leq t_2 \wedge t_2 \leq t_1)$. However, no complete list of theory lemmas is given in documentation. Therefore, as for the `rewrite` rule, a proof search in KeY is necessary to legitimate the lemma.

$$\frac{}{\Gamma \vdash \phi}\ \texttt{th-lemma} \qquad (T\text{-Tautology})$$

In addition to the use as axiom, the `th-lemma` rule can be used to deduce $\bot$ from a set of hypotheses (undocumented in API). This use case is tailored to fit the `lemma` rule already presented above.

$$\frac{\Gamma \vdash L_1 \quad \dots \quad \Gamma \vdash L_n}{\Gamma, L_1, \dots, L_n \vdash \bot}\ \texttt{th-lemma}$$

For this case we apply our generic replay scheme from Figure 3.2 and rely on KeY to find a proof for the sequent L.

trans* The `trans*` rule combines multiple subsequent applications of the rules `trans` and `symm`. For replay, we apply our generic replay scheme with automatic proof search in KeY for the branch L.

$$\frac{\Gamma_1 \vdash t_1 \simeq t_2 \quad \dots \quad \Gamma_n \vdash t_{n-1} \simeq t_n}{\Gamma_1, \dots, \Gamma_n \vdash t_i \simeq t_j}\ \texttt{trans*} \qquad \text{for any } i, j \in 1, \dots, n, i \neq j$$

distributivity The `distributivity` rule is used by Z3 to justify applications of DeMorgans's law, that is, distributing conjunctions over disjunctions or vice versa. The polyadic nature of Z3's conjunctions and disjunctions makes the implementation difficult, therefore we currently use our generic replay scheme complemented by proof search in KeY.

rewrite* When specific options for simplification are set, Z3 may use the `rewrite*` rule to justify multiple rewriting steps at once. Again, we apply the generic replay scheme with automatic proof search in KeY.

3.5.3 Other Rules

From the total of 42 proof rules of Z3, currently 11 rules can not be replayed. During experiments (more than 100 proofs started from problems shipped with KeY), 11 rules never appeared. These rules are:

- `undef`: This "rule" is used to indicate the `null` proof object. Since we start replay only if we have a valid proof, we can safely ignore this rule.
- `def-intro` and `apply-def`: According to the sparse documentation, these rules could be used to introduce an abbreviation for a term, and to apply the definition later. Since this is exactly what is done in nearly all proof examples by using the `let` binder, it seems that these two rules have been replaced by the more general concept with `let` (let binders can be used to share terms as well as whole subtrees of a proof).
- The rules `assumption-add`, `lemma-add`, `redundant-del`, and `clause-trail` are used for clausal proofs, which is an alternative proof mode of Z3's core solver "SMT". Clausal proofs must be enabled explicitly by setting an option, however, for some tests we conducted, the produced proof did not differ if this flag was set. Hence for now we do not set the option and ignore these rules.
- `pull-quant` and `push-quant`: These rules can be used to move quantifiers from the inside of functions to the outside (for example to distribute a universal quantifier over a conjunction). When this was necessary in our examples, always the rules `nnf-pos` and `nnf-neg` were used, the rule `pull-quant` and `push-quant` did never occur. Therefore, we also ignore them for now.
- `der` (destructive equality resolution) and `hyper-resolve`: These rules did never occur in any example, therefore, replay of them is currently not implemented. They both represent simplification steps that can be composed of several other rule applications. No conceptual difficulties exist for replay, they could be implemented (possibly using KeY's built-in proof search) if needed.

3.6 Properties of the Technique

In this section we describe theoretical properties of our replay technique. The translation of a KeY problem to Z3 is not the focus of this thesis and is, apart from the changes in type hierarchy translation described in Section 3.3.4, not part of this work. For this section, we assume it to be sound, but under-approximating: Some concepts of KeY, for example rules to symbolically execute Java code inside modalities, are not translated at all. Therefore, it may happen that even if a formula is valid (and provable in KeY), its translation to Z3 is no longer provable.

3.6.1 Soundness

The soundness of the technique is a direct consequence of the soundness of KeY's calculus rules.

Theorem 1. *The replay technique described in this work is sound.*

Proof. We assume that all KeY rules are sound, and all Z3 rules are replayed using these rules only[3]. As a consequence, the replay technique is sound. □

3.6.2 Totality

The replay technique is a (total) function: Every proof found by Z3 for a problem translated from KeY can be replayed in KeY, provided the assumptions below hold. These assumptions can be classified into two categories. The first category (assumptions 0, 1, and 2) relates to properties of Z3 and the produced proof, while the second (assumptions 3 and 4) contains statements about the completeness of KeY's proof search:

- Assumption 0: The Z3 proof is wellformed and closed. In addition, we assume that no silent rewriting of terms occurs in a way not described in the documentation[4].
- Assumption 1: Only proof rules which are part of the replay technique, that is, their replay is described in Section 3.5.1 and Section 3.5.2, occur.
- Assumption 2: For each quantifier, the sort of the quantified variable can be extracted from the typeguard. This is possible if Z3 does not introduce new typeguards and does not remove them from formulas during reasoning.
- Assumption 3: For the closing rules `rewrite` and `th-lemma`, KeY's built-in proof search strategy is able to find a proof.
- Assumption 4: For Z3 proof rules containing multiple schemas, KeY's automatic proof search strategy finds the necessary rule applications to close the proof. Rules where this is required are: `monotonicity`, `commutativity`, `def-axiom`, `nnf-pos`/`nnf-neg`, `trans*`, `distributivity`, `rewrite*`.

Theorem 2. *Under the assumptions given above, the replay technique is a total function, that is, every Z3 proof for a problem translated from KeY can be replayed into a KeY proof.*

Proof. Under the given assumptions, a closed proof in KeY can always be created by the construction technique described in this chapter. □

[3] Although tedious and difficult, the whole replay could theoretically be done interactively in the graphical user interface of KeY.

[4] Unfortunately, such behaviour has already been observed for the rules `asserted` and `monotonicity`. However, we can only hope it does not occur, and run KeY's proof search as a fallback.

4 Implementation of the Replay Technique in KeY

In this chapter, we describe how the previously described proof replay technique was implemented as a prototype within the KeY system. This implementation consists of an ANTLR 4 grammar for the Z3 proof format and 50 Java classes with approximately 4100 lines of code (not including the generated ANTLR parser). The code is available on the branch `pfeiferZ3ProofReplay` in the Git repository of the KeY system, the commit hash of the latest version at the time of writing is `a1d10fbbd6aafeb0f60a2deae43d268a7b2e490e`.

4.1 Communication with Z3

The idea to use Z3 (as well as other SMT solvers) for externalizing the proof search from KeY has long been there. A translation of KeY sequents to SMT-LIB input has already been implemented in KeY long before the start of this work. However, during the production of this thesis, there was ongoing work in writing a new, modular SMT translation with separate modules for different theories and features of the logic (such as sequences, arrays, heaps, quantifiers, ...). Our replay implementation is based on this new modular SMT translation, altered with type hierarchy changes as described in Section 3.3.4. In addition, a map from SMT-LIB style s-expressions to KeY terms is created for formulas of the original sequent in KeY. This allows us the find and insert the original formula during replay of the `asserted` rule.

The input to Z3 is given in the standard SMT format SMT-LIB. A relatively small example can be seen in Figure 4.1. Proof production has to be explicitly enabled in Z3 (line 4). Since it is useful to make the proof more readable for humans, we enable pretty printing (line 5). This mainly changes indentation, we also observed by trial and error that it sometimes prevents from simplifying the proof (or sub-proofs) in such a way that it does not contain sufficient information for replay: Without setting the option, in our experiments the whole proof could be simplified to the single rule application `(asserted false)`. Unfortunately it is unclear (undocumented and not easily observable by examining Z3's source code) how proof simplification is done, and if and how it could be disabled reliably. However, since it was helpful in our experiments, we decided to enable the pretty printer. Note that apart from `:print-success` and `:produce-models` which were already set by the current translation and do not influence the proof in any way, we set no other options in Z3. The proof output for the example from Figure 4.1 is shown in Figure 4.3.

```
1  ; --- Preamble
2  (set-option :print-success true)
3  (set-option :produce-models true)
4  (set-option :produce-proofs true)
5  (set-option :pp.pretty-proof true)
6  (set-logic ALL)
7
8  ; --- Declarations
9  (declare-sort T 0)
10 (declare-sort U 0)
11 (declare-const sort_any T)
12 (declare-fun u2b (U) Bool)
13 (declare-fun b2u (Bool) U)
14 (declare-const sort_boolean T)
15 (declare-const TRUE U)
16 (declare-const FALSE U)
17 (declare-fun instanceof (U T) Bool)
18 (declare-fun exactinstanceof (U T) Bool)
19 (declare-fun u2i (U) Int)
20 (declare-fun i2u (Int) U)
```

```
21 (declare-const sort_int T)
22 (declare-fun u_p () Bool)
23 (declare-fun u_q () Bool)
24 ; ... other declarations
25
26 ; --- Axioms
27 (assert (distinct sort_boolean sort_any))
28 (assert (forall ((b Bool)) (= (u2b (b2u b)) b)))
29 (assert (exactinstanceof TRUE sort_boolean))
30 (assert (exactinstanceof FALSE sort_boolean))
31 (assert (forall ((u U))
32              (=> (exactinstanceof u sort_boolean)
33                  (or (= u TRUE) (= u FALSE)))))
34 ; ... more type hierarchy axioms
35
36 ; --- Sequent
37 (assert (not (=> (=> u_p u_q) (=> (not u_q) (not u_p)))))
38
39 (check-sat)
40 (get-proof)
```

Figure 4.1: Z3 input for the sequent $\vdash (p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$

4.2 Overview and Architecture

The entry point for replay is the corresponding method of the class SMTReplayer (Figure 4.2). From there the parser is started, which parses the Z3 proof given in text format and creates an abstract syntax tree (AST). The parser is generated from an ANTLR 4 grammar for SMT-LIB extended with proof terms. Since many operations have to traverse the AST, it makes sense that they are implemented as visitors, extending the abstract ANTLR-generated AST visitor. In total, there are 7 visitors that serve various purposes:

1. Before the actual replay is started, namespaces are constructed for shared terms. This is is necessary, since the same symbols may contain in various contexts as abbreviations for different terms. The bindings are collected by the BindingsCollector and stored in a map in SMTReplayer.

2. The actual replay is done by a the class ReplayVisitor which is responsible for creating instances of ProofRule and initiating the replay of each individual rule.

3. When replaying a rule, its conclusion and premise terms must be collected from the Z3 proof tree and converted to KeY. This is done by the DefCollector.

4. As described in Section 3.3.3, during conversion of an SMT term to KeY, skolem symbols may occur. In such case, the SkolemCollector is responsible for traversing the proof until the corresponding `sk` rule is found. From there, the definition of the skolem symbol is extracted as term using the Hilbert choice operator.

5. When translating terms, for quantified variables the original sort has to be recovered from typeguards (see Section 3.3.4) by the TypeguardSortCollector.Note that this

Figure 4.2: Overview of the implementation architecture. Dashed arrows indicate the control flow (return directions are omitted). The class ReplayTools provides static utility methods and is used from nearly every class.

step is heuristical and may result in a wrong sort (a supersort of the sort the variable had originally in KeY), since Z3 could transform terms containing typeguards and (in theory) even introduce additional ones. However, this only occurs in very rare cases.

6. When replaying the `lemma` rule, it is not possible to extract the used hypotheses directly from the premises. Instead, we have to descend into the proof tree until we reach the leaves containing the corresponding `hypothesis` rules for the lemma. This is done by the HypothesesCollector.

7. Finally, there is an additional visitor (simply named TextCollector) for collecting the text corresponding to a rule node, thereby inlining all shared terms. It is used purely for debugging and usability purposes. After the text is collected, it is attached to proofs nodes in the KeY GUI as notes.

4.3 Implementation Challenges

Figure 4.3 show Z3's output for the contraposition example from Figure 4.1. As can already be seen in this small example, Z3 makes extensive use of sharing of terms (`a!1` in the example) as well as subtrees of the proof (`a!3`): It introduces new symbols as abbreviations and connects them to their definitions via the binder `let`. This extensive use of sharing poses a great engineering challenge: It makes the implementation difficult and error prone,

```
1  unsat
2  (let ((a!1 (not (=> (=> u_p u_q) (=> (not u_q) (not u_p)))))
3        (a!2 (or u_q (not u_p)) (not (or u_q (not u_p)))))
4  (let ((a!3 (mp (asserted a!1) (rewrite (= a!1 (not a!2))) (not a!2))))
5  (let ((a!4 (trans (monotonicity (iff-true (not-or-elim a!3 u_p) (= u_p true))
6                    (= (not u_p) (not true)))
7              (rewrite (= (not true) false))
8              (= (not u_p) false))))
9  (let ((a!5 (monotonicity (iff-false (not-or-elim a!3 (not u_q)) (= u_q false))
10        a!4 (= (or u_q (not u_p)) (or false false)))))
11 (let ((a!6 (trans a!5
12                  (rewrite (= (or false false) false))
13                  (= (or u_q (not u_p)) false))))
14 (mp (not-or-elim a!3 (or u_q (not u_p))) a!6 false)))))
```

Figure 4.3: Z3 proof output for the input in Figure 4.1. Note that the root of the proof tree is the term `false` in line 14, the first rule for replay is `mp` in the same line.

since an AST node traversed during replay could actually be a symbol bound by let, which would have to be unwrapped first. In addition, the current prototypical implementation does not reuse created terms and proofs, therefore often doing the same work multiple times. Of course, this comes with disadvantages in memory use of the replayed proof and time necessary to construct it. The evaluation in Chapter 5 shows that it is indeed a crucial performance factor.

For representing skolem symbols, the replay technique uses the Hilbert choice operator ϵ (see Section 3.3.3). While the choice operator is currently not implemented in KeY, a very similar and even more generic construct is available: The `ifEx` binder, which extends the choice operator by two separate explicit return values in case the condition is or is not met. This can be used in our translation:

$$\epsilon\, x : T.\ \phi(x) \quad \equiv \quad \text{ifEx}\ x : T.\ \phi(x)\ \text{then}\ x\ \text{else}\ d_T,$$

where T is any type and d_T (called `T::defaultValue` in KeY) an uninterpreted function depending only on T. Note that `ifEx` was actually only intended to be used for $T = integer$, hence there are no calculus rules for other sorts present in KeY. However, with the additional rule *epsDefAdd* as introduced in Section 3.4 (where every occurrence of ϵ is replaced by *ifEx* as described above) it can by used for any sort during the replay.

4.4 Current Limitations of the Prototype

As the current implementation is a prototype, there are limitations on the replayable Z3 proofs. First of all, a proof can currently not be replayed if multiple skolem symbols are introduced in a single occurrence of the `sk` rule. Second, the translation of terms to KeY only handles simple sorts such as Int and Bool, and object sorts. That is, compound sorts such as arrays and special KeY sorts such as sequences (more precisely, their translation to Z3) currently prevent replay.

The current implementation transfers identifiers to KeY without renaming. These may contain characters which are problematic for identifiers in KeY, such as the exclamation mark often contained in skolem symbols. While this does not hinder proof construction as well as saving, it turned out that KeY currently is not able to load saved proofs that contain such identifiers.

Closely connected to the previous limitation, the replay technique currently does not handle escaped symbols. Escaping is done with vertical bars in SMT-LIB, for example `|sort_int[]|`. These are introduced when translating arrays and fields to KeY, features that prevent replay anyway.

5 Evaluation

For an evaluation of the replay technique and its implementation, we provide some statistics of replayed proofs. Due to the prototypical state, out of a large set of benchmark and example problems shipped with KeY we selected 29 that could successfully be replayed with the current implementation. These selected problems are propositional or first-order, possibly containing uninterpreted functions, quantifiers, and arithmetic. For 26 of the 29 proofs KeY was also able to find a proof using its automatic proof search. For the remaining three problems KeY's built-in proof search strategy is not able to find a proof, however, they are provable when applying some rules interactively.

All tests have been run on a desktop computer featuring a quad-core 3.4 GHz processor and 12 GB DDR3 RAM. However, neither the replay technique nor the automatic proof search in KeY are parallelized, and also the parallelization features of Z3 have not been utilized. The problems have been loaded using KeY's API from custom test code, that is, without starting the graphical user interface of KeY. Afterwards, either the automatic proof search of KeY or the translation to SMT-LIB plus Z3 plus replay have been executed. KeY's proof search has been run using its default settings, with the maximum number of steps extended to 1 000 000 and a timeout of 5 min. As can be seen in the tables provided below, neither the step limit nor the timeout have been reached anywhere close in our examples.

5.1 Problems Automatically Provable by KeY

In Table 5.1, time measurements for the selected problems are shown. For easier comparability, the lines have been roughly sorted by difficulty of the proofs, that is, by the time needed for replay. The second column shows how long KeY's built-in proof search strategy needed for closing the proof. The other columns show values measured during the replay: The time necessary to translate the problem to SMT-LIB format plus the time Z3 needs to find a proof, and the time actually needed for replay. Finally, a total time adding together SMT-LIB translation, Z3 proof search, and replay time is given.

It stands out that even for the smallest proof, the time needed by KeY is not smaller than 100 ms. This is due to active waiting in KeY's command line interface: It waits for the automatic proof search in steps of 100 ms. The time that is really needed is anywhere beneath. However, this overhead does not change the fundamental result: When comparing KeY's automatic proof search with the total time of replay, that is, translation to Z3, proof search by Z3, and the actual replay, it can be clearly seen that the latter is currently usually much slower. The only exception to this are the very small examples, where the constant overhead as explained above dominates. For the larger examples, replay is even magnitudes slower. In addition, it is notable that the proof search in Z3 is typically faster than the proof search in KeY (third column), considering the fact that the corresponding

values also contains the time needed for translation of the problem to SMT-LIB. Finally, it becomes apparent that usually more than half of the total replay time is spent with automatic proof search in KeY.

File Name	KeY	Replay			
		Transl. + Z3	Replay	Auto	Total
`Ex3.41-formula1`	110	63	15	0	78
`doubleNeg`	110	63	16	0	79
`andCommutes`	110	47	31	0	78
`equalities`	109	47	31	0	78
`Sect2.5.1-ExampleProof`	110	62	31	15	93
`contraposition`	110	47	47	16	94
`mv1`	109	63	78	31	141
`subsumptionExample`	110	47	78	62	125
`simpleEps`	110	47	93	62	140
`liarsville`	109	62	110	94	172
`inequations2`	110	62	141	94	203
`liarsWithInt`	109	47	141	141	188
`Ex2.56`	109	78	156	31	234
`check_jdiv_concrete`	110	79	187	109	266
`projection`	110	63	188	31	251
`mv2`	109	62	203	32	265
`disjoint`	110	110	250	47	360
`inequations0`	109	63	265	204	328
`liars`	110	47	281	249	328
`castOperators`	110	62	359	123	421
`Ex2.57`	109	62	359	204	421
`Ex2.53`	109	63	843	311	906
`generalProjection`	110	62	969	782	1 031
`SET063p3`	109	62	1 875	1 314	1 937
`project`	1 057	172	13 437	7 497	13 609
`PUZ001p1`	2 953	375	83 344	50 742	83 719

Table 5.1: Time statistics for problems provable in KeY (in ms)

Apart from the time, also the size of proof is of interest for comparison. Table 5.2 show such statistics. The second column shows the number of rule applications needed in KeY to close the proof. The next two columns denote the line count of input (translated problem) and output (proof) of Z3. The final two columns indicate, how many rule applications were used in KeY during replay and how many thereof were created using automatic proof search. Of special interest are the columns “Shared Terms” and “Shared Sub-Proofs”. They give an idea, how essential sharing of terms and sub-proofs (via the `let` binder) is in Z3. For our table, we omitted the symbols that were only used once. The number given in the column “Shared Terms” denotes the sum of occurrences of each symbol defining an abbreviation for shared (that is, used at least twice) term, and likewise for “Shared Sub-Proofs”. We can see that these numbers get really large, the larger the Z3 proof is. This explains the enormous grow in the size of replayed proofs (column “Replay:Steps”) compared to the proof size in KeY: The current replay implementation has to inline the sub-proof each time it occurs, often doing the same work again and again. This indicates great potential optimizations: If we were able to detect a shared proof, we could introduce

File Name	KeY	Z3				Replay	
	Steps	Input Lines	Proof Lines	Shared Terms	Shared Sub-Proofs	Steps	Auto
Ex3.41-formula1	6	90	9	9	8	52	20
doubleNeg	4	62	17	12	6	175	87
andCommutes	6	62	18	9	8	199	92
equalities	7	80	22	12	12	294	114
Sect2.5.1-ExampleProof	6	62	18	9	8	199	92
contraposition	8	62	17	12	6	202	87
mv1	4	81	77	42	13	370	213
subsumptionExample	16	86	14	14	6	219	64
simpleEps	7	83	35	82	20	434	198
liarsville	14	95	66	19	2	161	52
inequations2	27	86	22	3	1	516	95
liarsWithInt	24	140	54	53	2	374	52
Ex2.56	6	82	62	140	23	483	264
check_jdiv_concrete	31	86	46	67	16	334	94
projection	10	83	83	109	27	764	416
mv2	7	81	85	82	17	516	243
disjoint	6	196	185	163	50	1 194	572
inequations0	50	88	30	141	22	771	177
liars	38	129	71	49	2	382	52
castOperators	6	137	190	232	40	1 101	577
Ex2.57	8	82	95	308	26	629	357
Ex2.53	14	83	149	590	38	2 011	719
generalProjection	11	84	60	370	35	1 013	460
SET063p3	89	114	264	586	61	4 867	778
project	148	138	770	1 789	226	15 989	3 280
PUZ001p1	3 197	199	1 200	5 578	380	51 373	6 136

Table 5.2: Size statistics for problems provable in KeY

its conclusion early in the proof (at the lowest common ancestor of the occurrences of the shared sub-proof) via KeY's cut rule. In this way each sub-tree would have to be replayed only once, probably at the cost of cluttering up the sequent.

In addition, in the current implementation we translate a shared formula from SMT format to KeY anew each time it is used in premise or conclusion of any rule. This is also the reason for the magnitudes of difference between time needed for proof search in KeY compared to replay time. A reasonable improvement would be to add a cache for translated formulas/terms. However, implementing this is not as easy as it sounds: Subterms may only be cached if they do not contain free variables, since in the text format of Z3 the sort of a free variable can only be determined at a binder. Implementing a cache for terms depending on free variables may be possible, but difficult. However, it surely would speed up the implementation further.

5.2 Problems Not Provable by KeY's Built-In Proof Seach

There are examples where KeY is not able to find a proof, and Z3 is able to find one. An example of such a problem is (for formulas ϕ and ψ):

$$(\exists x.\phi(x) \rightarrow \exists x.\psi(x)) \rightarrow \exists x.(\phi(x) \rightarrow \psi(x))$$

This problem defined in the input file `Ex2.54`, statistics can be found in the tables below. While it is provable in KeY when applying a manual case distinction, the built-in proof search does not find any proof, but stops after only a few steps, since no more rules are applicable. In contrast to that, Z3 is able to find a proof of 226 lines, which can be replayed using our technique KeY in about one second. As explained in the previous section, due to the extensive sharing in Z3's proof, to replay the proof the rather large number of 2 131 proof steps is necessary. The other two examples here are very similar: KeY stops after a few steps, since it can not apply any rule.

File Name	KeY		Replay		
	time	Transl./Z3	Replay	Auto	Total
`exists1`	109	63	157	79	220
`SET043p1`	109	47	235	125	282
`Ex2.54`	109	93	859	312	952

Table 5.3: Time statistics for problems Not automatically provable by KeY (in ms)

There may be several reasons for the advantage of Z3 over KeY here: First of all, Z3 is highly optimized for finding quantifier instantiations. Second, since currently KeY's rules/taclets are not translated at all in the translation to SMT input, proof search in Z3 is necessarily "less complete". This could be a curse and a blessing: On the one hand, there are formulas KeY could (at least theoretically) find a proof for and Z3 can not. On the other hand, maybe Z3 can find a proof because it is not distracted by the additional knowledge. Since the overhead usually is small compared to a proof search in KeY, it can be beneficial to at least translate the problem to Z3 and run Z3 with a small time limit.

File Name	KeY	Z3				Replay	
	Steps	Input Lines	Proof Lines	Shared Terms	Shared Sub-Proofs	Steps	Auto
`exists1`	9	83	130	108	22	751	335
`SET043p1`	13	86	86	193	17	621	215
`Ex2.54`	15	86	226	363	58	2 131	879

Table 5.4: Size statistics for problems Not automatically provable by KeY

6 Conclusion

6.1 Summary

In this thesis, we have shown that is possible to replay Z3 proofs in KeY. We have shown what the systematic difficulties are and how to deal with them: The differences in type hierarchies have been taken into account by adapting the translation from KeY to Z3 using typeguards and embeddings of primitive types into a universe type. The type hierarchy has been axiomatized using only functions and predicates that have a counterpart in KeY and thereby can be replayed. The problem of different proof directions of Z3 and KeY has been solved by locally "reverting" the proof direction in KeY by introducing the premises of a Z3 rule as cut where necessary. Equisatisfiability has been replaced by equivalence in combination with a changed handling of skolem symbols, where skolem constants and functions are represented using the Hilbert choice operator ϵ. Free variables of Z3 have been replaced by fresh skolem constants in KeY, a translation that is justified by the notions of validity, equisatisfiability and free variables, which have been carefully compared and determined to admit such a step. The problem of structurally different terms during replay has been handled on an engineering level. However, a lesson learned is that when replay is aimed for, the translation from KeY to Z3 should change formulas as little as possible.

For most of Z3's proof rules, translations have been described how to replay them in KeY, complemented by automatic proof search in KeY where necessary. The remaining ones are currently not included in the replay technique as they did never occur in any of the examples and tests we conducted.

Soundness of the replay technique has been proven. In addition, the technique is a total function: Under some given assumptions, every Z3 proof is successfully translated into a KeY proof.

The technique has been implemented as a prototype in KeY, which has been evaluated using multiple example problems. The presented replay statistics provided strong evidence that our replay technique would be a good complement to KeY's built-in proof search. Finally, it has also shown some great starting points and possibilities for future optimization.

6.2 Future Work

From an engineering point of view, there is much future work possible in improving the prototypical replay implementation. The current limitations could be resolved by implementing missing cases, rules, and translation features. In addition, for some rules that use KeY's automatic proof search during replay, for example `def-axiom`, it would be possible to implement separate manual replay for each of the multiple schemas represented

by this rule. This would reduce the dependency on KeY's built-in proof search strategy and aside from that most likely result in a speed-up, since the necessary replay steps would be known in advance and not have to be searched by KeY.

Probably most important for making the replay technique "production-ready" is optimization: As the evaluation statistics show, proper caching of terms and even sub-proofs should result in a massive speed-up as well as significant decrease in memory consumption. While it is clear (apart from engineering) how caching of terms could be done, for sub-proofs it is not that easy. There are at least two ideas: Possibly the so called *Merge Rules* could be used. Another idea is to introduce the shared sub-proof's conclusion early using the *cut* rule. With this, each shared proof would have to be replayed only once.

There is also some optimization potential in the fact that sometimes superfluous rules like below are introduced by Z3:

$$\frac{\vdash \phi \quad \vdash \phi \leftrightarrow \phi}{\vdash \phi}\,\texttt{mp}$$

Since the conclusion is identical to one of the premises, this rule could be completely omitted during replay, thus shortening the proof and saving time and memory.

Apart from optimization of the presented replay technique and its implementation, some completely different ideas arose while working on this thesis: Instead of aiming for a complete replay of Z3 proofs, one could exploit only some specific information contained in it. For example, KeY often seems to have problems finding useful quantifier instantiations, a feature Z3 is quite good at. Therefore, an idea would be to extract only the quantifier instantiations from a Z3 proof and use them to guide the KeY prover, otherwise relying completely on its built-in proof search. This could be even taken one step further: In addition (or, alternatively), one could introduce theory lemmas from Z3 via the *cut* rule to give hints to KeY which theory specific reasoning steps could be useful.

Finally, a sequent in KeY often contains many formulas that are not needed for a proof. From multiple formulas given as input, Z3 is able to compute the smallest set of unsatisfiable formulas – more specifically, *one* smallest set, since there may be multiple – which is called *unsat-core*. An idea would be to use this information to hide all other formulas in KeY except those really needed for the proof.

Bibliography

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., and Ulbrich, M. (2016). *Deductive Software Verification – The KeY Book*. Springer International Publishing.

Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., and Werner, B. (2011). A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Certified Programs and Proofs*, pages 135–150. Springer Berlin Heidelberg.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., and Tinelli, C. (2011). CVC4. In Gopalakrishnan, G. and Qadeer, S., editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer. Snowbird, Utah.

Barrett, C., Fontaine, P., and Tinelli, C. (2017). The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa. Available at `www.smt-lib.org`.

Barrett, C. and Tinelli, C. (2007). Cvc3. In Damm, W. and Hermanns, H., editors, *Computer Aided Verification*, pages 298–302, Berlin, Heidelberg. Springer Berlin Heidelberg.

Besson, F., Fontaine, P., and Théry, L. (2011). A flexible proof format for smt: A proposal.

Bjørner, N., de Moura, L., Nachmanson, L., and Wintersteiger, C. M. (2019). *Programming Z3*, pages 148–201. Springer International Publishing, Cham.

Böhme, S. (2009). Proof reconstruction for z3 in isabelle/hol. In *7th International Workshop on Satisfiability Modulo Theories (SMT'09)*.

Bouton, T., de Oliveira, D. C. B., Déharbe, D., and Fontaine, P. (2009). veriT: An open, trustable and efficient SMT-solver. In *Automated Deduction – CADE-22*, pages 151–156. Springer Berlin Heidelberg.

Böhme, S. (2012). *Proving Theorems of Higher-Order Logic with SMT Solvers*. Dissertation, Technische Universität München, München.

de Moura, L. and Bjørner, N. (2008a). Z3: An efficient SMT solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg.

de Moura, L. M. and Bjørner, N. (2008b). Proofs and refutations, and z3. In *LPAR Workshops*, volume 418, pages 123–132. Doha, Qatar.

Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L. P., and Tiu, A. (2006). Expressiveness + automation + soundness: Towards combining smt solvers and interactive proof assistants. In Hermanns, H. and Palsberg, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–181, Berlin, Heidelberg. Springer Berlin Heidelberg.

Giese, M. and Ahrendt, W. (1999). Hilbert's epsilon-terms in automated theorem proving. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, TABLEAUX '99, page 171–185, Berlin, Heidelberg. Springer-Verlag.

Leino, K. R. M. and Rümmer, P. (2010). A polymorphic intermediate verification language: Design and logical encoding. In Esparza, J. and Majumdar, R., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327, Berlin, Heidelberg. Springer Berlin Heidelberg.

Reynolds, A., Tinelli, C., Stump, A., Hadarean, L., Ge, Y., and Barrett, C. (2010). Cvc3 proof conversion to lfsc. Technical report.

Rümmer, P. (2008a). *Calculi for Program Incorrectness and Arithmetic*. PhD thesis, University of Gothenburg.

Rümmer, P. (2008b). A constraint sequent calculus for first-order logic with linear integer arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 274–289. Springer.

Stump, A. (2009). Proof checking technology for satisfiability modulo theories. *Electronic Notes in Theoretical Computer Science*, 228:121–133.

Stump, A., Oe, D., Reynolds, A., Hadarean, L., and Tinelli, C. (2012). SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118.

A Appendix

A.1 Example for Incompleteness With Epsilon in Input

When the choice operator is allowed to occur in the formula to prove, the calculus becomes incomplete. The following formula is an example that can not be proven with KeY:

$$(\forall x.\ \phi(x) \leftrightarrow \psi(x)) \quad \rightarrow \quad \epsilon x.\ \phi(x) = \epsilon x.\ \psi(x)$$

Since there are no taclets in KeY that are applicable to epsilon terms directly, a proof can not be conducted. However, this formula is actually characterizing the extensionality of the choice operator, and thus valid.

A.2 Comparison of the Type Hierarchy Axiom Count

Our new translation adds for each sort *s*: One type definition, two cast axioms, one typeguard axiom, and furthermore for each subsort of *s* one axiom ensuring that a value can not have top level sort and child sort simultaneously. In addition, for all object sorts we have to add axioms for making sure that if a value has two child sorts simultaneously, it can only be *null*. This leads to the following calculation (where n is the total sorts count, o the number of object sorts, s_{max} the maximum occurring number of children of any object sort, excluding the null sort and all interface sorts):

$$\begin{aligned}
\#axioms_{new} &= \sum_{i=1}^{n}(1+2+1+s_i) + \sum_{i=1}^{o}\left(\frac{s_i * (s_i - 1)}{2}\right) \\
&< \sum_{i=1}^{n}(4+n) + \sum_{i=1}^{n}\left(\frac{s_i * (s_i - 1)}{2}\right) \\
&< n * (4+n) + \sum_{i=1}^{n}\left(\frac{s_i * (s_i - 1)}{2}\right) \\
&< n * (4+n) + n * \left(\frac{s_{max} * (s_{max} - 1)}{2}\right) \\
&= n * \left(4 + n + \frac{s_{max} * (s_{max} - 1)}{2}\right) \qquad \in O(n * (n + s_{max}^2)) \\
&< n * \left(4 + n + \frac{s_{max} * (s_{max} - 1)}{2}\right) \qquad \in O(n^3)
\end{aligned}$$

In contrast to that, the original translation with $typeof$ and $subtype$ had much less axioms: It only needed 3 axioms for making the relation formed by $subtype$ a partial order, 2 cast

axioms, for each sort s a “definition” axiom and for each subsort of s an axiom defining the relation between them:

$$\begin{aligned}\#axioms_{original} &= 3 + 2 + \sum_{i=1}^{n}(1 + s_i) \\ &< 5 + n * (1 + s_{max}) \qquad \in O(n * s_{max}) \\ &< 5 + n^2 \qquad \in O(n^2)\end{aligned}$$

A.3 File Paths of Evaluation Examples Inside the KeY Repository

File Name	Path (relative to example directory `key/key.ui/examples/`)
provable automatically by KeY:	
Ex3.41-formula1	standard_key/BookExamples/03DynamicLogic/Ex3.41-formula1.key
doubleNeg	standard_key/prop_log/doubleNeg.key
andCommutes	standard_key/BookExamples/10UsingKeY/andCommutes.key
equalities	standard_key/pred_log/equalities.key
Sect2.5.1-ExampleProof	standard_key/BookExamples/02FirstOrderLogic/Sect2.5.1-ExampleProof.key
contraposition	standard_key/prop_log/contraposition.key
mv1	standard_key/pred_log/mv1.key
subsumptionExample	standard_key/inEqSimp/subsumptionExample.key
simpleEps	standard_key/pred_log/simpleEps.key
liarsville	standard_key/prop_log/liarsville.key
inequations2	standard_key/inEqSimp/inequations2.key
liarsWithInt	standard_key/pred_log/liarsWithInt.key
Ex2.56	standard_key/BookExamples/02FirstOrderLogic/Ex2.56.key
check_jdiv_concrete	standard_key/arith/check_jdiv_concrete.key
projection	standard_key/BookExamples/10UsingKeY/projection.key
mv2	standard_key/pred_log/mv2.key
disjoint	standard_key/types/disjoint.key
inequations0	standard_key/inEqSimp/inequations0.key
liars	standard_key/pred_log/liars.key
castOperators	smt/taclettranslation/castOperators.key
Ex2.57	standard_key/BookExamples/02FirstOrderLogic/Ex2.57.key
Ex2.53	standard_key/BookExamples/02FirstOrderLogic/Ex2.53.key
generalProjection	standard_key/BookExamples/10UsingKeY/generalProjection.key
SET063p3	standard_key/pred_log/tptp/SET/SET063p3.key
project	firstTouch/01-Agatha/project.key
PUZ001p1	standard_key/pred_log/tptp/PUZ/PUZ001p1.key
not provable automatically by KeY:	
exists1	standard_key/pred_log/exist1.key
SET043p1	standard_key/pred_log/tptp/SET/SET043p1.key
Ex2.54	standard_key/BookExamples/02FirstOrderLogic/Ex2.54.key

Table A.1: Paths of the evaluation examples from Chapter 5 (inside the KeY repository)