Transferring proof obligations from program verification in KeY to Isabelle/HOL

Bachelor's Thesis by

Nils Buchholz

at the KIT Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer: Prof. Dr. Beckert
Advisor: Wolfram Pfeifer, M. Sc.
Second advisor: Dr. Michael Kirsten

08 January 2024 – 08 May 2024

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

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, 08 May 2024

. . . .

(Nils Buchholz)

Abstract

Guarantees of correctness of programs are becoming more and more important nowadays. KeY is one tool, which can be used to prove the formal correctness of Java programs. It uses syntactic rewriting rules, called taclets, to successively simplify proofs. These taclet rules can be applied by an automated proving mode in KeY. Because the automation can be insufficient for closing some proofs and because proofs can contain thousands of rule applications, it is desirable to provide stronger automated tools to the user. One such tool is the Satisfiablity Modulo Theories (SMT) translation of KeY, which allows the use of SMT provers to close proofs. The SMT translation also lacks the required rules or proving strength for some proofs. It is thus still desirable to improve the already present automated toolset of KeY users.

This work designs a translation of KeY proofs to the higher-order logic prover Isabelle/HOL. In addition to designing this translation this work also shows that the translation can be automated and integrated within KeY by implementing it as a GUI-Extension of KeY.

This work tests the developed Isabelle translation on over 2 400 example proof obligations of KeY. In doing so the Isabelle translation has been found to offer some advantages over the SMT translation and the KeY automation. Although the Isabelle translation does not manage to close all proofs the KeY automation or SMT translation could close, it is able to close proofs, which the KeY automation or the SMT translation could not close.

Thus the Isabelle translation expands the KeY user's automated toolset in a meaningful way, while also building a foundation for using Isabelle in ways outside automated proof solving in KeY.

Zusammenfassung

Korrektheitsgarantien für Programme werden heutzutage immer wichtiger. KeY ist ein System, das genutzt werden kann, um die formale Korrektheit von Java Programmen zu zeigen. Es nutzt syntaktische Ersetzungsregeln, genannt taclets, um Beweise sukzessive zu vereinfachen. Diese taclet Regeln können in KeY von einem automatisierten Beweismodus angewandt werden. Weil die Automatisierung nicht ausreichend ist, um manche Beweise zu schließen, und, weil Beweise Tausende von Regelanwendungen enthalten können, ist es wünschenswert, dem Nutzer stärkere automatisierte Werkzeuge bereitzustellen. Ein solches Werkzeug ist die Satisfiability Modulo Theories (SMT) Übersetzung von KeY, die es erlaubt SMT Beweiser zu nutzen, um Beweise zu schließen. Auch der SMT Übersetzung fehlt es für manche Beweise an Regeln oder Beweisstärke. Es ist daher trotzdem wünschenswert, die bereits vorhandenen automatisierten Werkzeuge, die KeY Nutzer zur Verfügung haben, zu erweitern.

Diese Arbeit entwirft eine Übersetzung von KeY Beweisen zum generischen Beweisassistenten Isabelle/HOL, der auf Logik höherer Ordnung aufbaut. Zusätzlich zum Entwerfen dieser Übersetzung zeigt diese Arbeit auch, dass die Übersetzung automatisiert und in KeY integriert werden kann, indem die Übersetzung als eine GUI-Extension für KeY implementiert wird.

Diese Arbeit testet die entwickelte Isabelle Übersetzung auf über 2 400 Beweiszielen aus KeY Beispielen. Dadurch zeigte sich, dass die Isabelle Übersetzung einen Vorteil gegenüber der SMT Übersetzung und der KeY Automatik bietet. Obwohl die Isabelle Übersetzung nicht in der Lage ist, alle Beweise zu schließen, die die KeY Automatik oder die SMT Übersetzung schließen konnten, war sie dennoch in der Lage, Ziele zu schließen, die die KeY Automatik oder die SMT Übersetzung nicht schließen konnten.

Daher ergänzt die Isabelle Übersetzung die automatisierten Werkzeuge des KeY Nutzers auf sinnvolle Weise, und schafft gleichzeitig eine Grundlage dafür, Isabelle in KeY auf andere Weise als zur automatischen Beweisführung zu nutzen.

Contents

List of Figures xi

List of Tables xiii

List of Isabelle listings xv

1 Introduction 1

2 Foundations 3
2.1 KeY . . . 3
2.1.1 Introduction to KeY . . . 3
2.1.2 JavaDL in KeY . . . 3
2.2 Isabelle . . . 5
2.2.1 Introduction to Isabelle . . . 5
2.2.2 Types in Isabelle . . . 5

3 Related Work 9

4 Translation 11
4.1 Type Hierarchy of KeY . . . 12
4.1.1 Overview . . . 12
4.1.2 Generic types . . . 13
4.1.3 Type Hierarchy functions . . . 14
4.1.4 *Object* and *Null* . . . 14
4.2 Integers . . . 18
4.2.1 KeY Integer Operations . . . 18
4.2.2 Java Integer Operations . . . 19
4.2.3 Translation Alternatives . . . 20
4.3 Fields . . . 20
4.3.1 Type Definition . . . 21
4.3.2 Arrays . . . 21
4.4 Location Sets (LocSet) . . . 22
4.4.1 Type Definition . . . 22
4.4.2 Functions . . . 23
4.5 Heaps . . . 25
4.5.1 Type Definition . . . 26
4.5.2 Functions . . . 26
4.5.3 Translation Alternatives . . . 28

4.6 Sequences . . . 29
4.6.1 Type Definition . . . 29
4.6.2 Functions . . . 31
4.7 Limitations of the Translation . . . 33

5 Implementation **37**
5.1 Overview . . . 37
5.2 Scala-Isabelle . . . 38
5.3 Alternatives to Scala-Isabelle . . . 38
5.4 Issues found in KeY . . . 39

6 Evaluation **41**
6.1 Process of Evaluation . . . 41
6.2 Hypotheses and Conclusions . . . 41
6.3 Issues in KeY found during evaluation . . . 42

7 Conclusion **47**
7.1 Future Work . . . 47
7.2 Summary . . . 48

Bibliography **49**

A Appendix **51**
A.1 Extended Evaluation Statistics . . . 51
A.2 Complete TranslationPreamble theory . . . 54

List of Figures

2.1 The type hierarchy in KeY 4

4.1 Scheme of the translation 11
4.2 Signatures of *LocSet* functions 23
4.3 Signatures of *Heap* functions 27
4.4 Signatures of *Sequence* functions 31

5.1 Context Menu of the Implementation 39

List of Tables

4.1 Comparing different divisions 19

6.1 Number of goals translated and closed by each prover 43
6.2 Average proof search duration 44
6.3 Number of successful proofs by prover combination 44

A.1 Detailed proof states for all provers (1) 51
A.2 Detailed proof states for all provers (2) 52
A.3 Detailed proof states for all provers (3) 53

List of Isabelle listings

4.1 Abstraction type for JavaDL types 12
4.2 Axiomatization for predefined subtypes of *any* 13
4.3 Generated definition for *int[]* type 15
4.4 any typeclass declaration 15
4.5 Instance function translations 16
4.6 Definition of *java_lang_Object* type 16
4.7 Definition of *Null* type 17
4.8 jArithmetics locale 19
4.9 Definitions of jMod and euclDiv 20
4.10 Declaration of the *Field* type 21
4.11 Example of the distinct_fields lemma 21
4.12 Definition of special *Field* functions 22
4.13 *LocSet* type definition 23
4.14 *LocSet* functions 25
4.15 *Heap* type definition 26
4.16 *Heap* functions 30
4.17 *Sequence* type definition 31
4.18 *Sequence* Functions 34

A.1 Complete TranslationPreamble theory 54

1 Introduction

Programs nowadays have many applications - such as health and finance systems - that demand a high degree of both security and correctness. Therefore it is of great interest to formally prove the correctness of programs. The KeY system (Ahrendt et al., 2016) is a tool for formally verifying Java programs. It uses a sequent calculus, where each sequent has an antecedent and a succedent, which are sets of formulae. A sequent expresses that, when all of the formulae in its antecedent hold, at least one of the formulae in its succedent holds.

KeY has been continuously improved since its inception, enabling better reasoning about programs with heaps and handling of strings among other improvements. Still there are some constructs for which proofs cannot be completed automatically in KeY and whose interactive proving presents a significant challenge for users, like vector-multiplication and framing problems. In some of these problems the difficulty in interactive proving stems from the user knowing that certain terms are true, but KeY lacking the specific taclet rule to prove this. The issue of missing certain taclet rules could be solved by taking advantage of the larger proof libraries other provers, like the generic theorem prover Isabelle (Nipkow, Paulson, and Wenzel, 2002), possess. There may also be merit in using the automation features of Isabelle in addition to the toolset of KeY to take advantage of a diverse set of proof searching strategies and directions. To use both the Isabelle proof library and the automation features of Isabelle, it is desirable to transfer proof obligations from KeY to Isabelle.

This work focuses on translating proof obligations to Isabelle. The main goal of this work is giving the KeY user an additional automated tool to use, which offers an advantage over the currently available tools in KeY. Enabling KeY proof obligations to be translated to Isabelle can also serve to verify parts of the KeY system itself by translating KeY taclets into theorems in Isabelle, which can then be proven in Isabelle. Additionally, while proving theorems in Isabelle additional lemmas can be created and proven, which could be used to create new taclets for KeY. Ideally these taclets derived from Isabelle lemmas allow KeY to close more proofs without transferring proof obligations to Isabelle.

This work defines a translation of proof parts and their context in KeY to Isabelle/HOL proof goals and axioms where the original proof part is valid in the underlying model of KeY, if its translation is provable in Isabelle. Where possible established structures of Isabelle are used to represent their KeY counterparts. The translation equals the SMT translation of KeY in relative completeness.

As part of this work the translation was automated. The automation provides KeY users the option to transfer a subgoal of a KeY proof to Isabelle and to apply some of the automated proving methods of Isabelle. To allow Isabelle to prove the subgoal, some definitions and taclets of KeY are supplied to Isabelle in the form of a preamble. While it is important that the translation is sound, proving this is not covered by this work. The

translation parts are deliberately kept to either directly transferring semantics from KeY or make very conservative assumptions so they do not introduce any inconsistencies. My hypothesis is that this implementation of the translation allows the user to automatically close proofs that the automated features of KeY cannot close automatically. This will later be tested on over 2 400 proof obligations taken from examples in KeY.

This work begins by introducing the KeY and Isabelle system and some of the more advanced concepts used in the later parts in chapter 2. Related approaches are described in chapter 3. This work then describes the translation itself by describing the translation of the separate types and functions of the KeY core vocabulary in chapter 4. In the following chapter 5 the automation of the translation is described. Then in chapter 6 the testing methodology and the testing results are discussed. Finally chapter 7 gives an outlook on possible future improvements and concludes this work.

2 Foundations

2.1 KeY

This section introduces KeY. Afterwards some of the underlying logic of KeY will be introduced.

2.1.1 Introduction to KeY

KeY is a tool developed for the formal verification of Java programs. KeY allows the user to load Java Modeling Language (JML) contracts out of Java source files. The proof obligations of the contracts are expressed in sequents. A sequent expresses that, when all its antecedents hold, at least one of its succedents holds. KeY uses a special version of sequent calculus where the antecedent and the succedent are sets of formulae. KeY uses syntactic rules for rewriting sequents, so-called taclets, to repeatedly simplify a given sequent until either a formula in its antecedents is a falsity, a formula in the succedent is truth, or a formula in the antecedent matches a formula in the succedent. Then one of three corresponding "closing-rules", called `closeFalse`, `closeTrue`, `close` respectively, is used to close a given proof. Some taclets can split the proof obligation into other subgoals with their own proof obligations. Thus a typical KeY proof forms a proof tree of subgoal nodes.

The KeY user interface seeks to be especially user friendly and intuitive. KeY taclets are applied from a context menu opened by clicking the part of the sequent the taclets should be applied on.

2.1.2 JavaDL in KeY

This section describes the internal logic used by KeY. It should be noted that KeY uses an extended version of first-order logic and as such does not allow quantification over types, functions or sets. There is no set logic in KeY.

KeY uses a form of JavaDL. Its sequents can contain Java program parts in the form of modalities. These modalities are succesively simplified and reduced to atomic assignments, which are expressed in updates. The translation does not handle sequences that contain updates, therefore we will not explore their inner workings.

KeY uses the type *any* as a top type, *boolean* to represent boolean values, *int* to represent integers, *Seq*, *Location Set (LocSet)*, *Heap*, *Field*, as well as *java_lang_Object* for the *Object* class of Java and class types for other classes imported from Java source code. Finally the *Null* type represents the Java class of the same name. The types are structured in a hierarchy where *any* serves as the toptype. Java class types follow the class hierarchy of

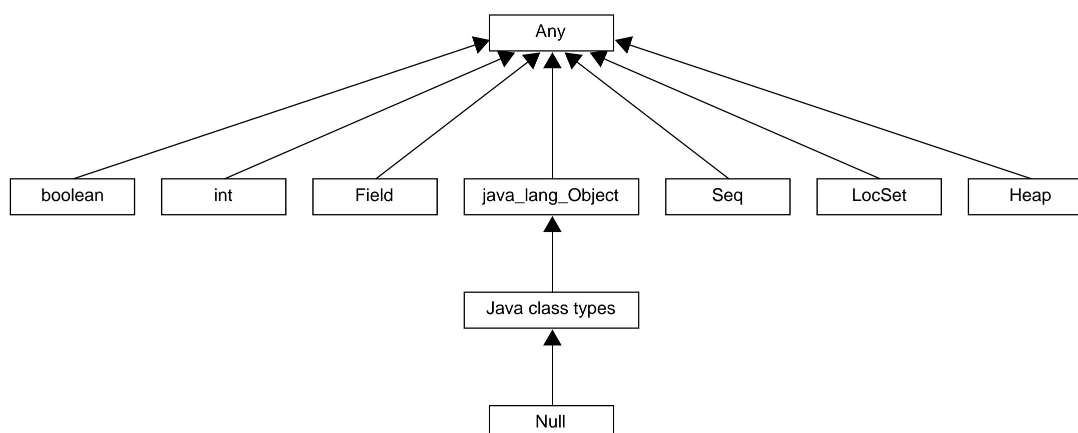

Figure 2.1: The type hierarchy in KeY

Java and are thus subtypes of *java_lang_Object* and parent types of *Null*. The KeY type hierarchy is depicted in Figure 2.1. The semantics of these types, where left unclear, will be briefly introduced in the following paragraphs and later in their respective sections in chapter 4.

There are additional functions *instance* for determining if a particular value is an instance of a given type, *exactInstance* for determining if a value is an instance of a given type and not of any of its bottom types.

Location Set *LocSet* is the type of sets of locations. A location is an (*java_lang_Object*, *Field*) pair and is not explicitly defined as a type in KeY. Locations are used to describe an address on a heap. Most of the *LocSet* functions are set functions that are only defined for *LocSet*, as KeY does not support set logic outside of *LocSet*.

Field *Field* is a type to represent the fields of Java objects. They are mainly used to describe locations on heaps. The *Field* type is left mostly unspecified. The only known facts about fields are that they are distinct from each other and that there is an infinite number of fields. There is a predefined **created** field to state that the object on the heap at a location was created. To describe the location of the values of an array there is an injective function **arr**, which maps integers to unspecified fields.

Seq *Seq* is the type of *Sequences*. *Sequences* are finite sequences of *any* values. They are similar to list types in other logics. Their functions consist of many of the common list functions, such as accessing the value at a given index, concatenation of two sequences, removal of elements at a given index, etc. There are also functions to describe permutations of sequences.

Heap Lastly *Heap* represents heaps of a Java program. Semantically they are functions mapping locations to *any* values. Their functions are heap management functions, such as

store to store a value at a location, *select* to retrieve a value at a location, *create* to create a new object on the heap, *anon*, which anonymizes a heap at a given set of locations and *wellFormed*, which is a predicate to describe the well-formedness of a heap.

2.2 Isabelle

This section introduces Isabelle. A following section will introduce some more advanced concepts that will occur in the later chapters describing the translation.

2.2.1 Introduction to Isabelle

Isabelle is a generic proof assistant, that can be used to formally prove theorems ranging from complex mathematical proofs to formal verification for programs. It provides structures to introduce functions, abstract datatypes, definitions, axioms, etc. Proofs in Isabelle are written in text form. Functions and definitions in Isabelle are written in a functional programming style akin to Haskell. There are many logics implemented in the Isabelle system, but this work only uses Isabelle/HOL (Nipkow, Paulson, and Wenzel, 2002), which uses higher-order logic. The proof is usually conducted semi-automatically, where the user can choose which *tactics* to apply at each proof step by writing their application onto the proof text. Tactics combine multiple rule applications. These tactics can hold immense value for the user as they do not require full knowledge of all rules defined in Isabelle and their names to close proofs. Among these tactics are simpler ones like simp, linarith, force, fastforce, which apply simplification, introduction and arithmetic rules. There are also more high-level tactics like "try0", which tries a list of the simpler tactics to see if they can close a proof's subgoal. Even more advanced is "sledgehammer", which calls a list of *automatic theorem proving* (ATP) provers and *Satisfiability modulo theories* (SMT) solvers.

Isabelle stores proofs in a "theory" file, which contains the proof text. Theory files can be collated in "sessions". Isabelle theories can import the contents of other theories and sessions to take advantage of their theorems and definitions. Isabelle also possesses a large library of theorems for different concepts, which the user can take advantage of.

2.2.2 Types in Isabelle

Isabelle uses a flat type hierarchy. This means there are no subtypes in Isabelle. The types of terms and variables can be explicitly stated by adding "::type" to the term. If the type of a variable is not explicitly stated, Isabelle uses type unification to infer the type of a variable. There are also type variables in Isabelle, which usually follow the pattern *'a*, to express statements about multiple types.

typedef The main method of introducing new types to a theory is through the typedef axiomatization scheme. To define a new type using typedef one must provide a set of an existing type and prove it is non-empty. Isabelle then introduces morphisms, which map between the new type and the specified set. It is possible to rename the morphisms from the default names "Rep_type" and "Abs_type". *Rep_type* is sometimes referred to as the

representation function. It maps from instances of the newly defined abstract type to their representation in the specified set. *Abs_type* is sometimes referred to as the abstraction function. It maps from members of the specified set to their abstraction equivalent in the new type.

locales Locales provide a way to reason about parametric theories like orders. A locale can “fix” constants and provide assumptions, which hold within the locale. Within the body of the locale the assumptions can be used like axioms. The locale provides an encapsulated environment to prove theorems without stating concrete types by using type variables instead. It is possible to instantiate a locale for a concrete type by supplying concrete values for the constants in the locale, that fulfill the assumptions of the locale and whose types can be unified with the type variables of the locale.

Typeclasses Typeclasses are syntactic sugar for locales, and provide a more readable interface for the concepts held within. Some types share functions with similar concepts like addition on the numeric types. Typeclasses can abstract from the concrete definition of these functions and instead provide conditions which must hold for these functions like commutative properties. The body of a typeclass can then be used to prove general statements about the functions which can be used for all types that instantiate this class. During instantiation of a class the fixed functions can be overwritten for the concrete type. It then needs to be proven that these concrete instances fulfill the necessary properties stated in the typeclass’s definition. These typeclasses can have a hierarchy where typeclasses inherit the properties of other classes. The numeric types *nat*, *int*, *quot*, *real* and *complex* are examples of instantiations of various ring and field typeclasses, which form a typeclass hierarchy. The conversion functions between these number types are also declared in the type classes.

In contrast to locales typeclasses always include exactly one type variable, which stands for a member of the class of types. Their statements thus always involve statements about a type and cannot be used to express statements about other constructs like groups.

coercive subtyping Isabelle supports “coercive subtyping” (Luo, Soloviev, and Xue, 2013), which implicitly applies coercion functions to terms of a subtype. The value of this coercion is a member of the parent type. Thus coercive subtyping allows Isabelle to treat elements of a subtype as elements of their parent type. Any function that maps between two types can be used as a coercion. In this way the “Rep” morphism could be used as a coercion. Then the newly defined type could implicitly be used in place of the type its set was taken from.

lifting Some functions and definitions can be transferred from a base type to its top type. An example would be addition for the *nat* type of natural numbers and the *int* type, which is defined based on *nat*. The lifting mechanism (Huffman and Kunčar, 2013) is used for transferring such functions and definitions. It automatically adds the required representation and abstraction functions in definitions to transfer the logic of parent types to the abstract type. Additionally it can automatically prove additional proof obligations

that might arise from the use of representation and abstraction functions in the definition. First the lifting package needs to be setup for a new type by the “setup_lifting” command, which is given the *type_definition* predicate as well as the representation and abstraction functions. Then definitions from the parent type can be lifted by the “lift_definition” command.

3 Related Work

In this work we aim to improve upon the automated toolset of KeY by translating KeY proof obligations in the form of modality- and update-free sequents to the higher-order logic generic theorem prover Isabelle/HOL.

Bian et al. (2021) used Isabelle as a back-end for KeY to reason about collections. They specified abstract data types and parts of JML contracts in Isabelle. These abstract data types were then imported as uninterpreted symbols into KeY. KeY would handle generation of proof obligations from the JML contracts and the underlying Java source code. KeY also handled resolving the modalities in the proof obligations. In order to enable KeY to close these proofs additional properties of the ADTs were formulated as lemmas and subsequently proven in Isabelle. Afterwards these lemmas were used to create taclets, the internal representation of rules in KeY, for KeY to use, which allowed KeY to close proofs. This work will not be adding imported data types to KeY. Consequently Isabelle will not be proving the translation of specific abstract data types and their properties, but will instead be proving formulae generated in KeY. Thus this work we will not be translating lemmas from Isabelle to KeY taclets, but translating KeY proofs to Isabelle lemmas.

Trentelman (2005) translated three pivotal taclets for assignment rules into Bali, a collection of formalizations of Java semantics in Isabelle, and proved them. In contrast to this manual taclet translation, this work translates complete formulae instead.

Pusch (1999) used *Bali* to formalize parts of the Java Virtual Machine to formalize a specification of a Java bytecode verifier and proved its soundness. This differs from my approach as KeY does not operate on Java bytecode, but Java source code.

There have been several implementations of sequent calculi in Isabelle. One of these, the Isabelle theory “Sequents”, is part of the Isabelle distribution. However, as it was implemented in 1993 and has not received significant changes since, it conflicts with more modern parts of Isabelle. The most critical conflict is between the “Sequents” and the “Main” theory of Isabelle. To take advantage of the more recent features of Isabelle, the “Sequents” theory will not be used. Another implementation of a sequent calculus was done by From, Schlichtkrull, and Villadsen (2021), who formalized soundness and completeness proofs of a one-sided sequent calculus - meaning that the succedent of a sequent consists of one formula - for first-order logic in Isabelle. Their approach implemented a sequent calculus for plain first-order logic, whereas this work handles multi-sorted sequents from KeY.

The addition of Satisfiability Modulo Theories (SMT) solvers to KeY (Ahrendt et al., 2016) shares with this work that it aims to aid KeY in finding difficult proofs. The connection to SMT solvers translates certain formulas from KeY into instances of the satisfiability problem for SMT solvers. Both the restrictions on which formulae can be translated and the conversion effort could be reduced when translating to Isabelle, as there are established structures in Isabelle, like integers, which can serve as a foundation for translating

structures from KeY. However, the implementation serves as an architectural foundation for the implementation of the translation to Isabelle.

4 Translation

This chapter begins with an overview of the translation scheme. It continues with describing the translation by describing the translation of the separate parts of a proof obligation. It concludes by listing some of the known limitations of the translation.

This work presents a translation of KeY proof obligations in the form of sequents to Isabelle. The translation is restricted to sequents that are free of any updates/modalities (subsection 2.1.2). Figure 4.1 shows the scheme of the translation that will be explained now. The translation takes a KeY sequent as input and translates it to an Isabelle theorem. As there is not sufficient context for all of the types and proof independent functions of KeY in a theorem, the translation consists of the "TranslationPreamble" theory and the main "Translation" theory, which are described in the following paragraph.

The "TranslationPreamble" theory holds all proof independent constants and types of KeY. The full "TranslationPreamble" theory can be found in the appendix (section A.2). The "Translation" theory imports the "TranslationPreamble" theory. The "Translation" theory contains generated type definitions for all types found in the KeY sequent of the proof obligation, that are proof dependent. This concludes the preamble-esque structure inside the "Translation" theory. The core part of the "Translation" theory is a locale (see subsection 2.2.2), which introduces a constant for every function found on the sequent. Inside the locale is a theorem holding the translated proof obligation. The theorem contains the antecedents of the sequent translations as its assumptions and the succedents of the sequent connected by disjunctions as its proof goal.

The following sections describe the translation of various aspects found in KeY proof obligations. We will first introduce the translation of pure first-order logic proof obligations, then introduce translations for generic types, meaning both Java types and generic subtypes of *any*, and, how the type hierarchy is translated. We then present the translations for other JavaDL types mentioned in subsection 2.1.2, such as *Field*, *LocSet* and *Heap*. During

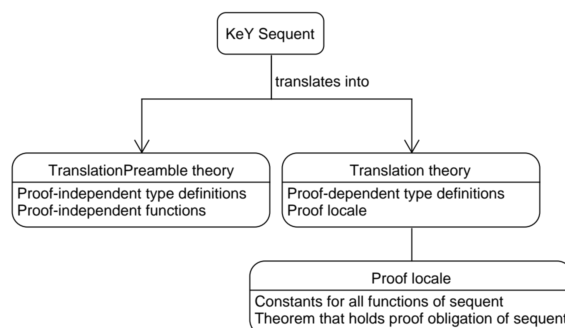

Figure 4.1: Scheme of the translation

```
typedef javaDL_type = "(UNIV::any set set)"
  by auto
setup_lifting type_definition_javaDL_type
lift_definition typeof::"any⇒javaDL_type⇒bool" is Set.member.
lift_definition subtype::"javaDL_type⇒javaDL_type⇒bool" is Set.subset_eq.
lift_definition disjointTypes::"javaDL_type⇒javaDL_type⇒bool" is Set.disjnt.
```

Isabelle listing 4.1: Abstraction type for JavaDL types

these steps we will occasionally remark on alternatives to the translations chosen in this work.

First-order constructs, such as boolean functions and quantifiers, are defined the same in both KeY and Isabelle and thus are translated directly. Predicates are converted to functions that map to *bool*.

4.1 Type Hierarchy of KeY

4.1.1 Overview

KeY types have a hierarchy of subtypes. The toptype of the hierarchy is *any*. Figure 2.1 depicts the hierarchy. There are fixed subtypes of *any* and further Java class types and generic *any* subtypes that can be added as part of the problem description. Isabelle does not have a type hierarchy. The translation defines a *javaDL_type* type as an abstraction of types. This type of types will be useful in further aspects of the translation, such as array types. The *javaDL_type* type is defined based on the universes of *any* subtypes. The **UNIV** constant of Isabelle *'a set* types represents the universe of the given type *'a*. Thus for **UNIV** to represent universes of *any* subtypes it is necessary to explicitly type **UNIV** as a set of sets of *any* values, because Isabelle cannot infer which kind of set **UNIV** is meant to be in this context otherwise. The subtype relation is lifted (subsection 2.2.2) from the subset relation, while the membership relation **typeof** is lifted from the membership relation for sets, as in Isabelle listing 4.1.

The translation defines the *any* type as a nondescript type with a non-empty universe using **typedecl**. The translation defines subtypes of *any* based on the set in *any* that represents their universe. This is done using **typedef** (see subsection 2.2.2), which provides morphisms between the specific subtype and *any*, also referred to as representation and abstraction functions. The morphisms can be used for coercive subtyping in Isabelle (see subsection 2.2.2). The translation of the hierarchy is thus based on the relations of the universes of KeY types. It is necessary to state additional axioms to define the disjointness of some types and their universes.

The inclusion of mandatory subtypes of *any*, that are already defined in Isabelle - like *int* and *bool* - cannot be done using **typedef** as they are already defined. So the translation defines their subtype relation using axiomatizations instead. The axiomatization states that the **type_definition** predicate is valid for representation and abstraction functions between *any* and the respective type and the types universe and introduces the required

```
typedecl any
axiomatization where int_sub_any[simp]:"type_definition int2any any2int
(int_UNIV)"
declare [[coercion int2any]]

interpretation int:type_definition int2any any2int int_UNIV
  by simp
```

Isabelle listing 4.2: Axiomatization for predefined subtypes of *any*

representation function, abstraction function and the respective types set representation in *any* as new constants. The **type_definition** predicate states that the representation and abstraction functions are isomorphisms between the abstract types universe and the given set. Using this axiomatization it is ensured that the underspecified universe of *any* contains the universe of the respective type or more specifically its representation as a set of *any*. Based on this axiomatization the type can then be interpreted as an instance of the **type_definition** locale. This models the way **typedef** would work on an undefined type.

4.1.2 Generic types

Java types are defined via the **typedef** mechanism of Isabelle (subsection 2.2.2). The types defined by **typedef** are required to be non-empty, which needs to be shown prior to their definition. Thus it is necessary to show that the parent types of the new type intersect. For this purpose, **bottom**, a constant of type *any* is introduced. The **bottom** constant is later used to define **null**. An abstraction of the bottom element is present in all other types whose definitions are generated by this translation. This simplifies the proof that the parent types intersect, which would otherwise have to be axiomatized, because all type universes introduced by the translation are only stated to be non-empty subsets of the universe of *any*.

The introduction of new types by the translation follows a pattern. An example of this pattern is shown in Isabelle listing 4.3 for the Java type *int[]*.

- It is shown that the universe representations of the parent types of the new type intersect. In the example Isabelle listing 4.3 the corresponding lemma is called "ex_intarr_UNIV".
- The universe of the new type is introduced as a constant.
- **specification** is used to specify that the universe of the new type is a subset of the universes of its parent types. **specification** requires a proof, which shows that a value that fulfills the specification statements exists.
- Because **specification** requires some unusual workarounds to use its statements in Isabelle proofs, the statements are repeated in a lemma.
- Using the "..._UNIV" constant, **typedef** is used to define the new type.

- To shorten the proofs in the definitions of further subtypes of the new type, it is shown that the bottom element is part of the universe of the new type. To more conveniently prove subtype relations it is also shown that the new types universe representation is a subset of its parent types universe representations.
- A "..._type" constant is introduced that represents the universe of the new type.
- Additionally functions, which map from the type to its parent types, are added. These are declared as coercions.

This pattern can be used to translate any type by substituting "intarr" for the name of the type and by replacing "java_lang_Object" with a parent type of that type. For types with multiple parent types the statements about the subset relation between the new type and its parent type can be repeated with conjunctions between them. The proofs in the pattern can also be reused for other type definitions by replacing the type name and the parent type names.

4.1.3 Type Hierarchy functions

cast The *cast* function in the JavaDL of KeY should map values of type *any* to the required type *A*. The translation of *cast* functions is contained in the `any` **typeclass** (see subsection 2.2.2) seen in Isabelle listing 4.4. The `any` **typeclass** contains functions to map from *any* to the respective type and vice versa. The concrete functions are supplied during instantiations for each type, which are either already present in the translation preamble or are subsequently added for each generated type as seen toward the end of Isabelle listing 4.3.

instance* and *exactInstance *instance* is a family of functions in KeY that is used to determine if a value is a member of a specific type. This function shares semantics with the **typeof** function defined in Isabelle listing 4.1. The translation therefore defines it using **typeof**. The translation gives it the "_type" value that corresponds to the type of the *instance* function instance that was translated. While there is another function *exactInstance* in KeY, this serves little purpose in Isabelle, because most variables and terms are already explicitly typed. The translation introduces it as a constant without further semantics. The translation of *instance* and *exactInstance* is contained in Isabelle listing 4.5.

4.1.4 *Object* and *Null*

The *java_lang_Object* is declared similar to the pattern used for the *int[]* type. Because *java_lang_Object* is a direct subtype of *any*, there is no need to show that the parent types intersect. The definition of this type is shown in Isabelle listing 4.6. The *Null* type as stated earlier (subsection 4.1.2) is defined based on the **bottom** constant. *null* is the abstraction of **bottom** in the *Null* type. The definition of *Null* is shown in Isabelle listing 4.7. The relation between *Null* and other types is inferred from the inclusion of **bottom** in all generated Java class type universes.

```
lemma ex_intarr_UNIV:"{bottom} ⊆ (UNIV::java_lang_Object set) ∧ {bottom} ⊆
(UNIV::any set) ∧ bottom ∈ {bottom}"
  by simp

consts
intarr_UNIV::"any set"

specification (intarr_UNIV) "intarr_UNIV ⊆ (UNIV::java_lang_Object set) ∧
intarr_UNIV ⊆ (UNIV::any set) ∧ bottom ∈ intarr_UNIV"
  using ex_intarr_UNIV by blast

lemma intarr_UNIV_specification:"intarr_UNIV ⊆ (UNIV::java_lang_Object set) ∧
intarr_UNIV ⊆ (UNIV::any set) ∧ bottom ∈ intarr_UNIV"
  by (metis (mono_tags, lifting) intarr_UNIV_def someI_ex ex_intarr_UNIV)

typedef intarr = "intarr_UNIV"
  morphisms intarr2any any2intarr
  using intarr_UNIV_specification by auto

declare [[coercion intarr2any]]

lemma intarr_type_specification[simp]:"(UNIV::intarr set) ⊆
(UNIV::java_lang_Object set) ∧ (UNIV::intarr set) ⊆ (UNIV::any set) ∧ bottom ∈
(UNIV::intarr set)"
  using intarr_UNIV_specification using type_definition.Rep_range
type_definition_intarr by blast

fun intarr2java_lang_Object where "intarr2java_lang_Object x =
any2java_lang_Object (intarr2any x)"

declare [[coercion intarr2java_lang_Object]]

definition "intarr_type = Abs_javaDL_type (UNIV::intarr set)"

instantiation intarr::any
fun to_any_intarr where "to_any_intarr x = intarr2any x"
fun cast_intarr  where "cast_intarr x = any2intarr x"
instance by standard
end
```

Isabelle listing 4.3: Generated definition for *int[]* type

```
class any =
  fixes to_any::"'a⇒any"
  fixes cast::"any⇒'a"
```

Isabelle listing 4.4: any typeclass declaration

```
fun instanceof::"any⇒javaDL_type⇒bool"
  where "instanceof x type = typeof x type"

consts
  exactInstance::"any⇒javaDL_type⇒bool"
```

Isabelle listing 4.5: Instance function translations

```
consts
  java_lang_Object_UNIV::"any set"

specification (java_lang_Object_UNIV) "java_lang_Object_UNIV ⊆ (UNIV::any set)"
  "bottom:java_lang_Object_UNIV"
  by auto

lemma java_lang_Object_UNIV_specification:"java_lang_Object_UNIV ⊆ (UNIV::any
set) ∧
      bottom:java_lang_Object_UNIV"
  by (metis (mono_tags, lifting) java_lang_Object_UNIV_def UNIV_I subset_UNIV
verit_sko_ex_indirect)

typedef java_lang_Object = "java_lang_Object_UNIV"
  morphisms java_lang_Object2any any2java_lang_Object
  using java_lang_Object_UNIV_specification by auto

declare [[coercion java_lang_Object2any]]

definition java_lang_Object_type::"javaDL_type" where "java_lang_Object_type ≡
Abs_javaDL_type (UNIV::java_lang_Object set)"

lemma bottom_in_java_lang_Object[simp] :"bottom ∈ (UNIV::java_lang_Object set)"
  using java_lang_Object_UNIV_specification
  using type_definition.Rep_range type_definition_java_lang_Object by blast

instantiation java_lang_Object::any
fun to_any_java_lang_Object where "to_any_java_lang_Object x =
java_lang_Object2any x"
fun cast_java_lang_Object where "cast_java_lang_Object x = any2java_lang_Object x"
instance by standard
end
```

Isabelle listing 4.6: Definition of *java_lang_Object* type

```
typedef (overloaded) Null = "{bottom}"
  morphisms Null2any any2Null
  by simp

declare [[coercion Null2any]]

lemma bottom_Null_set:"(UNIV::Null set) = {bottom}"
  using type_definition.Rep_range type_definition_Null by blast

lemma Null_sub_java_lang_Object_Types: "(UNIV::Null set) ⊆
(UNIV::java_lang_Object set)"
  using bottom_Null_set bottom_in_java_lang_Object by auto

definition "null ≡ any2Null bottom"

instantiation Null::any
fun to_any_Null where "to_any_Null (x::Null) = Null2any x"
fun cast_Null where "cast_Null x = any2Null x"
instance by standard
end

declare [[coercion Null2java_lang_Object]]
```

Isabelle listing 4.7: Definition of *Null* type

4.2 Integers

This chapter contains the details of the translation of integers from KeY to Isabelle. The translation of integers is an important part of the translation, as all arithmetic operations in KeY are converted to integer operations and all array and sequence accesses rely on integers. We will start by explaining the translation of KeY integer operations, then we will explain the translation of the functions representing Java integer operations in KeY. Finally we will discuss some alternative translation options.

4.2.1 KeY Integer Operations

This section describes the translation of the standard integer operations in KeY. The universe of integers in KeY is defined as the universe of all mathematical integers, see subsection 2.1.2. As this aligns with the integer definition in Isabelle, KeY integers are translated directly into Isabelle *int*. Addition, subtraction and multiplication are defined the same in both KeY and Isabelle. Thus addition, subtraction and multiplication are translated directly to their Isabelle counterparts.

Division and modulo are defined differently in Isabelle and KeY. Division and modulo are defined based on the euclidean semantics in KeY, also known as E-Division (Boute, 1992). In contrast division in Isabelle is defined as F-Division (Boute, 1992), that always rounds down. For the reader's convenience the definitions of both division functions are found in Definition 1 and Definition 2. As can be seen in Table 4.1, the different division definitions coincide for positive divisors and differ for negative ones. To translate the KeY modulo function a new function **euclMod** is defined in the preamble. The **euclMod** function coincides with the Isabelle **mod** function for integers for positive divisors and adds the absolute value of the divisor to the result of **mod** for negative divisors. The result of **euclMod** is always nonnegative and smaller than the absolute value of the divisor. Therefore **euclMod** corresponds to the modulo of E-Division. Using Euclid's theorem (Definition 3) the division of E-Division can be defined based on the modulo operation. The translation for KeY division is another function introduced in the preamble called **euclDiv** (Isabelle listing 4.9). This function is defined through Euclid's theorem and the **euclMod** function.

Translating KeY division is complicated by the deliberate underdefinedness of KeY division, which does not prevent division by zero, but does not define its value. The Isabelle definition stating that the multiplicative inverse of 0 is 0 also complicates any translation. To avoid deriving proofs, which include division by 0, the translation uses a locale (see section 2.2) called **jArithmetics** (Isabelle listing 4.8), which does not define these functions, but instead introduces them as constants and provides assumptions from which the values of the functions for non-zero divisors can be derived.

Definition 1 ($\mathbf{div}_{KeY}$ is the integer division function in KeY. This shows the definition for dividing integer a by integer b. The lack of a case for $b = 0$ is intentional and reflects underdefinedness in KeY)**.**

$$a\ \boldsymbol{div}_{KeY}\ b = \begin{cases} \lfloor \frac{a}{b} \rfloor, & b > 0, \\ \lceil \frac{a}{b} \rceil, & b < 0. \end{cases}$$

```
locale jArithmetics =
  fixes jDiv::"int⇒int⇒int"
  assumes jDiv_def [simp]: "b≠0 ⟹ jDiv a b =
(if ((a≤0 ∧ b<0) ∨ (a≥0 ∧ b>0) ∨ (b dvd a)) then (a div b)
else ((a div b) + 1))"

fixes euclMod::"int⇒int⇒int"
assumes eucl_Mod_def [simp]: "l≠0 ⟹ euclMod k l = (if (k mod l < 0) then ((k mod
l) + abs(l))
else (k mod l))"
```

Isabelle listing 4.8: Definition of the jArithmetics locale that holds the integer function definitions

Table 4.1: Comparing different divisions

	Euclidean	F-Division	Truncated
5 **div** 2	2	2	2
5 **div** -2	-2	-3	-2
-5 **div** 2	-3	-3	-2
-5 **div** -2	3	2	2

Definition 2 ($\mathbf{div}_{Isabelle}$ is the integer division function in Isabelle. This shows the definition for dividing integer a by integer b)**.**

$$a \; \boldsymbol{div}_{Isabelle} \; b = \begin{cases} \lfloor \frac{a}{b} \rfloor, b \neq 0, \\ 0, b = 0 \end{cases}$$

Definition 3 (Euclid's theorem)**.**

$$a = b \cdot (a \; \boldsymbol{div} \, b) + a \; \boldsymbol{mod} \, b$$

4.2.2 Java Integer Operations

This section describes the translation of the Java integer operations in KeY. In addition to KeY integer operations, KeY supports Java integer operations. These are modeled by encasing the results of the respective operations in a modulo statement. Thus addition, subtraction and multiplication are translated to functions that combine this modulo statement with the respective translation.

Division in Java follows the truncated division definition, sometimes referred to as division rounding to 0 or T-Division (Boute, 1992). It coincides with the division of Isabelle

```
fun jMod::"int⇒int⇒int" where
"jMod a b = a - (jDiv a b)*b"

fun euclDiv::"int⇒int⇒int" where
"(euclDiv k l) = (k - euclMod k l) div l"
```

Isabelle listing 4.9: Definition of **euclDiv** and **jMod** based on Euclid's theorem

when dividend and divisor have the same sign. The translation introduces the **jDiv** constant in the **jArithmetics** locale (Isabelle listing 4.8) to translate the *jDiv* function that represents integer division in Java from KeY. The values for **jDiv** can be derived from the assumption **eucl_Mod_def** in the locale. The definitional assumption is based on **div** in Isabelle and ensures values are rounded up instead of down in case the divisor and dividend had different signs.

Proofs I have proven that these functions fulfill the specifications of the respective operations using Isabelle. The proofs are included in the full "TranslationPreamble" theory in the appendix (section A.2). For **euclMod** - the equivalent to $\mathbf{mod}_{KeY}$ - this requires showing that it always returns a non-negative value that is smaller than the absolute value of the divisor. Due to **euclDiv** being defined based on Euclid's theorem, its correctness follows from the correctness of **euclMod**. In the same way the correctness of **jMod** follows from the correctness of **jDiv**.

4.2.3 Translation Alternatives

While there is an Isabelle theory about euclidean , the "Euclidean_Division" theory, using it requires reinterpreting *int* as a member of a different **typeclass**. Doing so would restrict access to the theories involving the standard division for *int*. Such a translation would also be incompatible with the possible use of multiple types of division in KeY. A KeY proof obligation can contain both the Java division and the E-Division of KeY. Therefore the translation defines KeY division and Java division as new functions in Isabelle.

Instead of establishing the division operations as constants in a locale and using assumptions to define their values, one could use partial functions in Isabelle to translate division, however partial functions significantly hamper the ability of Isabelle to automatically prove a goal.

4.3 Fields

In KeY *Field* represents the fields of Java Objects. Their semantics are left fairly open. The only specifications Ahrendt et al. (2016) makes are, that the *Field* universe contains the *created Field*, and that all *Field* values are distinct. KeY itself additionally makes *Field* a subtype of *any*.

```
consts
  Field_UNIV::"any set"
specification (Field_UNIV) "Field_UNIV ⊆ (UNIV::any set)"
  "Field_UNIV ≠ {}"
  by auto

lemma Field_UNIV_specification:"Field_UNIV ⊆ (UNIV::any set) ∧
  Field_UNIV ≠ {}"
  by (metis (mono_tags, lifting) Field_UNIV_def empty_not_UNIV someI_ex
top_greatest)

typedef Field = Field_UNIV
  morphisms Field2any any2Field
  using Field_UNIV_specification by auto

declare [[coercion Field2any]]
```

Isabelle listing 4.10: Declaration of the *Field* type

```
locale varsAndFunctions = jArithmetics +
fixes SumAndMax_max::"Field"
fixes SumAndMax_sum::"Field"
assumes distinct_fields:"(distinct [SumAndMax_max,SumAndMax_sum, created]) ∧
(({SumAndMax_max,SumAndMax_sum, created} ∩ image arr (UNIV::int set)) = {})"
```

Isabelle listing 4.11: Example of the distinct fields lemma in a translation of the "SumAndMax" example

4.3.1 Type Definition

The *Field* type is introduced using a subset of *any*. The subset is a constant that is only specified to be non-empty. The subset relation between this constant and the universe of *any* is trivially true.

The *created Field* is introduced as a constant. During the translation of the sequent all *Field* values are collected and a lemma is added to the proof locale. This lemma states that all *Field* values found in the sequent are distinct, and that they are neither the *created Field* nor do they lie in the image of *arr*. An example instance of this lemma can be found in Isabelle listing 4.11

4.3.2 Arrays

KeY models arrays using an injective function, *arr*, which maps integers to a *Field*. Accessing an array at index i is then equivalent to accessing the heap at the *Field arr* (i). The translation retains this way of accessing arrays. As such the *arr* function is defined in Isabelle as an injective function using axiomatization in Isabelle listing 4.12.

```
consts
  created::"Field"

axiomatization arr::"int⇒Field" where arr_inject[simp]:"(arr x = arr y) = (x
= y)"
```

Isabelle listing 4.12: Definition of special *Field* functions

Some axioms require determining the element type of an array. As such a function **elementType** mapping from array type instances to their element types is declared in the `array` **typeclass**. All array types are then given an instantiation of this **typeclass**, where the concrete function for the type is defined.

Alternatively, due to arrays always having fixed types, it would be possible to translate arrays to the *'a list* type in Isabelle. This could offer a performance boost in automated reasoning, because the *'a list* functions are much better integrated and easier to resolve than the semantics of the necessary *Heap* functions. However, this would require severe changes when translating how array access is written on the sequent. Furthermore, translating them to *'a list* directly would cause a problem for axioms involving arrays. Axioms would now have to account for the type variable in the definition of the *'a list* type. Lastly array access would have to be masked, as arrays are accessed using integers, while *'a list* instances can only be accessed using *nat*, the Isabelle type of natural numbers. The translation would hence also need to check if the integers value can be converted to a natural number. Isabelle cannot do this on its own, as the present functions for converting *int* to *nat* map all negative *int* values to 0.

4.4 Location Sets (LocSet)

The *Location Set* type (abbreviated *LocSet*) models sets of locations. Locations are (*Object*, *Field*) tuples, that describe a location on a heap. *LocSet* values are translated to an abstraction of sets of (*Object*, *Field*) tuples.

4.4.1 Type Definition

The translation defines the *LocSet* type through **typedef** as seen in Isabelle listing 4.13. *LocSet* is defined based on sets of (*java_lang_Object*, *Field*) tuples. “UNIV::(java_lang_Object × Field set set)” is the universe of all sets of (*java_lang_Object*, *Field*) tuples.

To comply with the KeY type hierarchy, *LocSet* must be made into a subtype of *any*. Thus this subtype relationship is axiomatized by the translation using the **type_definition** predicate. Declaring the **LocSet2any** function as a coercion (subsection 2.2.2) ensures that Isabelle uses this function to transform *LocSet* occurrences where terms of type *any* are needed.

```
typedef LocSet = "UNIV::(java_lang_Object × Field) set set"
  by simp

consts
  LocSet_Image::"any set"
  LocSet2any::"LocSet⇒any"
  any2LocSet::"any⇒LocSet"
axiomatization where LocSet_sub_any:"type_definition LocSet2any any2LocSet
LocSet_Image"
declare [[coercion LocSet2any]]
```

Isabelle listing 4.13: Definition of the *LocSet* type in the translation preamble

empty : *LocSet*
elementOf(*java_lang_Object*, *Field*, *LocSet*)
subset(*LocSet*, *LocSet*)
disjoint(*LocSet*, *LocSet*)
union : *LocSet* × *LocSet* ⟶ *LocSet*
intersect : *LocSet* × *LocSet* ⟶ *LocSet*
setMinus : *LocSet* × *LocSet* ⟶ *LocSet*
allLocs : *LocSet*
allFields : *java_lang_Object* ⟶ *LocSet*
allObjects : *Field* ⟶ *LocSet*
arrayRange : *java_lang_Object* × *int* × *int* ⟶ *LocSet*

Figure 4.2: Signature of *LocSet* functions. Function signatures are separated from their name by":". Predicate signatures are shown in brackets.

4.4.2 Functions

As KeY does not include its own set logic, most *LocSet* functions have the semantics of operations on sets. Here the translation uses the **lifting** package (subsection 2.2.2) to great effect, as most *LocSet* functions are directly transferred from sets. The *LocSet* functions and their translation (Isabelle listing 4.14) will be explained in the following paragraphs. The pattern "Set.subset" states that this refers to the **subset** function of the "Set" theory to avoid duplicate function names in Isabelle. The required *LocSet* functions consists of *empty*, *subset*, *union*, *intersect*, *setMinus*, *allLocs*, *singleton*, *elementOf*. A list of these functions and their signatures is shown in Figure 4.2 for the reader to compare them to their translations in Isabelle listing 4.14.

empty The *empty* function returns an empty *LocSet*, which has the semantics of an empty set of locations. As such its translation lifts the definition of the Isabelle function **empty** for sets.

elementOf *elementOf* is used to check if a location is in a given *LocSet*. Because all functions are curryied upon translation, the translation cannot simply lift the membership

function for sets **member** in Isabelle to define *elementOf*. It is necessary to convert the two parameters to a tuple, before lifting the definition of **member**. The translation does this by lifting a lambda construct, that reconstructs the tuple from the parameters.

subset The *subset* predicate has the semantics of the subset predicate on sets. Thus its translation lifts the definition of the corresponding **subset** function of Isabelle.

disjoint The *disjoint* predicate states that two *LocSet* values are disjoint sets of locations. Thus its translation lifts the definition of the **disjnt** function of Isabelle.

union The *union* function returns the union of two *LocSet* values. Thus its translation lifts the definition of the Isabelle function **union**.

intersect The *intersect* function returns the intersection of two *LocSet* values. Thus its translation lifts the definition of the Isabelle function **inter**, which returns the intersection of two sets.

setMinus The *setMinus* function returns the set difference of two *LocSet* values. Its translation lifts the definition of the Isabelle function **minus** of the “Set” theory, which returns the set difference of two *LocSet* values.

singleton *singleton* returns a *LocSet* containing only a single location. There is no direct counterpart to this in Isabelle. Isabelle instead uses the **Collect** function with a single parameter. The translation uses a lambda construct to convert the two parameters in a location back to a tuple and to return the value of the **Collect** function, then lifts it to define *singleton*.

allLocs The *allLocs* function returns a *LocSet* containing all possible locations. Its translation is the **UNIV** constant of the type of (*Object*, *Field*) tuples, which is the constant representing the universe of (*Object*, *Field*) tuples.

allFields *allFields* returns the *LocSet* containing all locations that involve a given *java_lang_Object*. The translation lifts a lambda construct to define *allFields*. This lambda construct maps the given *java_lang_Object* to the cartesian product of the set containing just the given *java_lang_Object* and the **UNIV** constant - the universe of the *Field* type.

allObjects *allObjects* returns the *LocSet* containing all locations that involve a given *Field*. The translation lifts a lambda construct to define *allObjects*. This lambda construct maps the given *Field* to the cartesian product of the set containing just the given *Field* and the **UNIV** constant - the universe of the *java_lang_Object* type.

```
setup_lifting type_definition_LocSet

lift_definition empty::"LocSet" is Set.empty.
lift_definition subset::"LocSet⇒LocSet⇒bool" is Set.subset.
lift_definition disjoint::"LocSet⇒LocSet⇒bool" is Set.disjnt.
lift_definition union::"LocSet⇒LocSet⇒LocSet" is Set.union.
lift_definition intersect::"LocSet⇒LocSet⇒LocSet" is Set.inter.
lift_definition setMinus::"LocSet⇒LocSet⇒LocSet" is minus.
lift_definition singleton::"java_lang_Object⇒Field⇒LocSet" is "λobj f. {(obj, f)}".
lift_definition elementOf::"java_lang_Object ⇒ Field ⇒LocSet⇒bool" is "λobj f s.
(obj, f) ∈ s".
lift_definition allLocs::"LocSet" is Set.UNIV.
lift_definition allFields::"java_lang_Object⇒LocSet" is "λx. {x} × (UNIV::Field
set)".
lift_definition allObjects::"Field⇒LocSet" is "λx. (UNIV::java_lang_Object set) ×
{x}".
lift_definition arrayRange::"java_lang_Object⇒int⇒int⇒LocSet" is "λobj x y. {obj}
× (image arr {x..y})".
```

Isabelle listing 4.14: The translations of core *LocSet* functions

arrayRange *arrayRange* maps a *java_lang_Object* and two integers *i* and *j* to the *LocSet* containing all locations that involve the given *java_lang_Object* and the *Field* values, that the *arr* function (see subsection 4.3.2) maps the integers between *i* and *j* to. The translation defines a lambda construct, that maps the *java_lang_Object* and the two *int* values to the cartesian product of the set containing only the *java_lang_Object* and the image of *arr* on the set containing all *int* values that lie between the two given *int* values. This lambda construct is lifted to define *arrayRange*.

4.5 Heaps

KeY defines the *Heap* type to represent Java heaps. The semantics presented in Ahrendt et al. (2016) state that Heaps are functions mapping locations (Object, Field pairs) to values of type any. KeY uses placeholder variables, that represent *Heap* instances in the various *Heap* functions. This allows KeY to quantify over *Heap* without expanding the first-order logic based reasoning of KeY to include higher-order logic features like quantifying over functions. KeY deviates from the definition of *Heap* presented in Ahrendt et al. (2016). While in Ahrendt et al. (2016) *Heap* is not a subtype of *any*, KeY itself makes them a subtype of *any*. As a consequence nested *Heap* values are possible, where a *Heap* stores another *Heap* at a location.

```
typedef Heap = "UNIV::(java_lang_Object ⇒ Field ⇒ any) set"
  by simp
declare [[coercion Rep_Heap]]

consts
  Heap_Image::"any set"
  Heap2any::"Heap⇒any"
  any2Heap::"any⇒Heap"

axiomatization where Heap_sub_any:"type_definition Heap2any any2Heap Heap_Image"
declare [[coercion Heap2any]]
```

Isabelle listing 4.15: The definition of the *Heap* type in the translation preamble

4.5.1 Type Definition

The translation reflects the semantics of Ahrendt et al. (2016). By defining *Heap* based on functions the translation takes advantage of the higher-order logic of Isabelle for reasoning about *Heap*.

The translation defines the *Heap* type (Isabelle listing 4.15) using **typedef**. The defining set for *Heap* is the set of all functions from *java_lang_Object* and *Field* to *any*. The coercion mechanism (subsection 2.2.2) is then applied to allow *Heap* values to implicitly be used as functions.

Heap being a subtype of *any* necessitates some level of abstraction from functions to satisfy the type unification requirements of Isabelle. The use of the **typedef** definition over functions by the translation fulfills the need for this abstraction. It is still required to define the now defined *Heap* type as a subtype of *any*. Making *Heap* a subtype of *any* is done akin to the way *int* and *bool* were made into subtypes of *any* (see section 4.1) by introducing constants, which by axiomatization fulfill the properties of representation and abstraction functions.

4.5.2 Functions

The translation for *Heap* includes the *select*, *store*, *create*, *anon*, *unusedLocs* functions and the *wellFormed* predicate for *Heap* values. The signatures of these functions is presented in Figure 4.3, so the reader can better compare the functions to their translations. Their translations are shown in Isabelle listing 4.16 and will be explained in the following paragraphs.

select The *select* function that accesses the value stored at a location in a *Heap*, is equivalent to the value of the function representing the *Heap*, when the function is given the location as parameters. Consequently the *select* operation is translated to a function that returns the value of the *Heap* at the given location. Due to the *select* operation also ensuring that the return has a specific type, the value of the *Heap* is cast to the required type by way of applying the cast function to the value. In KeY there are multiple *select*

$\text{select}_A : \mathit{Heap} \times \mathit{java_lang_Object} \times \mathit{Field} \longrightarrow A$ for any subtype A of the *any* type
$\text{store} : \mathit{Heap} \times \mathit{java_lang_Object} \times \mathit{Field} \times \mathit{any} \longrightarrow \mathit{Heap}$
$\text{create} : \mathit{Heap} \times \mathit{java_lang_Object} \longrightarrow \mathit{Heap}$
$\text{unusedLocs} : \mathit{Heap} \longrightarrow \mathit{LocSet}$
$\text{anon} : \mathit{Heap} \times \mathit{LocSet} \times \mathit{Heap} \longrightarrow \mathit{Heap}$
wellFormed(Heap)

Figure 4.3: Signature of *Heap* functions. Function signatures are separated from their name by":". Predicate signatures are shown in brackets.

functions, which form a family of functions with a *select* function for each returned type. In the translation, the return type of *select* is stated to belong to any type instantiating the `any` **typeclass**. Every occurrence of *select* in the translation of the sequent is then explicitly typed. In this way it is ensured that the return type of each *select* function occurrence is correct.

store The counterpart to the *select* function, the *store* function, which stores a value in a given *Heap* at a location, should return a new *Heap* that has the given value at the given location and otherwise coincides with the previous *Heap*. Therefore, the translation defines a function that maps the old *Heap*, location and value to a new *Heap*, that fulfills this property. The new *Heap* is the abstraction of a lambda construct that follows the *select* operations semantics presented in Ahrendt et al. (2016).

create The *create* operation used to create new objects on a *Heap* should alter the *Heap* only for the given *Object* and only at the *created Field*. The *created Field* should be set to *true* for the given *Object*, while the rest of the Heap remains unaltered, provided the *Object* is not *null*, in which case the *Heap* should remain unaltered. The translation maps the old *Heap* and the *Object* to the abstraction of a lambda construct, which maps an *Object* and *Field* to **True** - should the *Object* be the one just created - and to whatever value the old *Heap* held otherwise.

unusedLocs A location on a *Heap* is unused, when the *Object* at this location has not been created. Thus the function *unusedLocs* is translated to a function that maps a *Heap* to the *LocSet* abstraction of the set holding all locations, whose *Object* values are not created - meaning that the value of the *created Field* of this *Object* is **False**.

anon The *anon* operation, which is used to obfuscate access to a *Heap* outside a given set of locations, maps two *Heap* values and a *LocSet* to a new *Heap*, which coincides with the first *Heap* at locations in the *LocSet* and coincides with the second *Heap* for locations outside the *LocSet* and locations that were unused in the first *Heap*. The translation defines *anon* as a function, which maps both *Heap* values and the *LocSet* to the lambda construct, which abstracts the function corresponding to the *Heap* that maps locations to the required values.

wellFormed KeY uses the *wellFormed* predicate to express if a *Heap* is well formed. The value of the operation is determined through a series of axioms stating how transformative *Heap* operations - such as *select* and *store* - create new well formed *Heap* values out of other well formed *Heap* values.

As the semantics of the *wellFormed* predicate heavily rely on type information, which is either not available in Isabelle or not available to the translation, the translation chooses to translate the axioms of *wellFormed* instead of its semantics. The main issue in transferring the semantics directly are array types. Translating the semantics of *wellFormed* directly would require quantifying over all array types or, alternatively, checking, if a given *Object* is an array type, both of which are not possible in Isabelle. While it is possible to quantify over values of a theoretical polymorphic array type - similar to the type *'a list* - by way of the `array` **typeclass**, any quantification over values belonging to a polymorphic type would only quantify over the values of a single type instead of all possible types. This also introduces a type variable to the definition, which while possible, is discouraged and requires instantiation of the type variable for the definition to be used. Defining *wellFormed* as a function is not possible due to this extra type variable occurring on the right hand side of the function.

4.5.3 Translation Alternatives

This section describes some alternative approaches to translating heaps. I have tried multiple ways to transfer arrays and their element types, none of which enabled a true transfer of the semantics of *wellFormed*. One approach introduces a general array top type, through which introducing any type variables in the definition can be avoided. However, using elements of such a top type would mean that information about the true element types of array values are lost. The element types would have to be individually declared for each array instance. This defeats the purpose of translating the *wellFormed* predicate, because instead of axiomatizing every array, the well-formedness of every *Heap* on the sequent could be axiomatized. Introducing the element type through type membership of the specific array types would fail when taking subtype arrays into account. Subtype arrays would inherit the larger element types of their parents, which is not correct.

Another approach to translating the array axioms declares arrays as polymorphic types, but also suffers from introducing type variables into wellFormeds definition when quantifying over a generic array type. Because of these issues, the semantics of *wellFormed* are not translated. Instead, the translation opts for introducing the axioms present in KeY, which preserve the semantics where arrays are not concerned. The axiom involving arrays suffers from the type variable problem, but is present nonetheless.

It might be feasible to define an inductive function to reconstruct a *Heap* and derive its well-formedness this way. However the necessary information is not available in a KeY sequent. As most JML contract obligations do not require proving the well-formedness of a *Heap*, this definition through axioms should suffice.

Presumably *Bali* could prove useful in this aspect, but due to heaps in *Bali* differing too much semantically from the *Heap* of KeY *Bali* was not used in this translation.

Use of *if-then-else* statements in the definition of *store* could be replaced by preconditions. This could make the values of the *store* function easier to find for automatic provers, but

hampers readability and requires the definition to be split into multiple parts. Splitting the definition also prevents using the lifting package or defining *store* as a function at all. The *store* function would have to be translated to a constant that is specified by axioms stating its properties.

Instead of defining some *Heap* operations as functions and transferring the underlying semantics presented in Ahrendt et al. (2016), it is possible to transfer the axioms/their taclet representations. As the goal of this work was not to transfer taclets from the taclet collection of KeY to Isabelle axioms and lemmas, or including many axioms in the translation preamble, the taclet axioms were not translated.

It is possible and may be beneficial to the performance of **sledgehammer** to define additional lemmas, which prove these taclets based on the present semantics. In doing so one could use the present semantics of the translation to verify the correctness of the transferred taclets.

4.6 Sequences

This section describes the translation of the *Sequence* type. First we will briefly recap the explanation of the *Sequence* type from paragraph 2.1.2. Then I will explain the translation of *Sequence* and the translation of its functions.

A *Sequence* represents a sequence of elements of type *any*. *Sequence* values can be nested since *Sequence* is a subtype of any. *Sequence* is used as a list-esque structure in KeY.

4.6.1 Type Definition

Sequence being a list-esque structure means that lists in Isabelle are used as the translation target for *Sequence* values. A *Sequence* corresponds to an *any list* meaning a *list* containing elements of type *any*. Both *Sequence* and *'a list* describe list-like structures of finite length with most functions of *Sequence* finding their equivalents in Isabelle theories on *'a list*. A key difference between the two types is that, while the Isabelle *list* type is strictly typed, *Sequence* values allow elements to be of any type, so long as it is a subtype of *any*. In KeY this means all values can occur inside a *Sequence* as *any* is functionally a toptype in KeY. Another difference between *Sequence* and *'a list* is *Sequence* values are accessed with integers and provide a constant that represents the value of accessing the *Sequence* outside its bounds, while access of the Isabelle type *'a list* is non-exhaustive - meaning it does not necessarily have a value - and uses natural numbers. To preserve most of the theorems on the Isabelle type *list*, *Sequence* is defined using **typedef** (subsection 2.2.2) given the universe of *any list*. Conveniently this solves the issue of *Sequence* being ambiguous in its typing by hiding nested *Sequence* values as elements of type *any*, which allows for infinite nesting depths and mixed nesting depths in a single *Sequence*.

Defining *Sequence* based on *list* necessitates using axiomatization to introduce the representation and abstraction functions to the *any* type, as well as the representative universe of *Sequence* in the *any* type. Isabelle listing 4.17 is the part of the preamble that introduces the *Sequence* type. *Sequence* is abbreviated to *Seq* inside KeY and in the translation.

```
fun select::"Heap⇒java_lang_Object⇒Field⇒'a::any" where
  "select h obj f = cast (h obj f)"

fun store::"Heap⇒java_lang_Object⇒Field⇒any⇒Heap" where
  "store h obj f x = Abs_Heap (λ(obj'::java_lang_Object) (f'::Field). (if obj'=obj
∧ f'=f ∧ f≠created then x else h obj' f'))"

fun create::"Heap⇒java_lang_Object⇒Heap" where
  "create h obj = Abs_Heap (λ(obj'::java_lang_Object) (f'::Field). (if obj'=obj ∧
f'=created ∧ obj≠null then cast True else h obj' f'))"

fun unusedLocs where "unusedLocs (h::Heap) = Abs_LocSet {((obj::java_lang_Object),
(f::Field)). (h obj created=False)∧ obj≠null}"

fun anon::"Heap⇒LocSet⇒Heap⇒Heap" where
  "anon h1 s h2 = Abs_Heap (λ(obj::java_lang_Object) (f::Field). (if elementOf obj f
s ∧ f≠created ∨ elementOf obj f (unusedLocs h1)
    then select h2 obj f else select h1 obj f))"

axiomatization wellFormed::"Heap⇒bool" where
  onlyCreatedjava_lang_ObjecteAreReferenced:"wellFormed h ⟹ select h obj f = null
∨
      ((select h (select h obj f) created)::bool)"
  and onlyCreatedjava_lang_ObjectsAreInLocSets:"wellFormed h ∧ elementOf
(o2::java_lang_Object) f2 ((select
      h obj f)::LocSet) ⟹ Null2java_lang_Object null=o2 ∨ ((select h o2
      created)::bool)"
  and wellFormedStorejava_lang_Object:"wellFormed h ∧ ((x::java_lang_Object)=null
∨ ((select
      h x created) ∧ instanceof x (fieldType f))) ⟹ wellFormed (store h obj f x)"
  and wellFormedStoreLocSet:"wellFormed h ∧ (∀ov fv. (elementOf ov fv y ⟶ ov =
null ∨ select h ov created))
      ⟹ wellFormed (store h obj f y)"
  and wellFormedStorePrimitive:"(typeof x (fieldType f) ⟹ ¬typeof x
java_lang_Object_type ⟹ ¬typeof x LocSet_type  ⟹  wellFormed h
    ⟹ wellFormed (store h obj f x))"
  and wellFormedCreate:"wellFormed h ⟹ wellFormed (create h obj)"
  and wellFormedAnon:"wellFormed h ∧ wellFormed h2 ⟹ wellFormed (anon h y h2)"

axiomatization where wellFormedStoreArray:"wellFormed h ∧
((x::java_lang_Object)=null ∨ (select h x created ∧ (typeof x (element_type obj))))
      ⟹ wellFormed (store h (cast (to_any (obj::'a::{array, any}))) (arr idx) x)"
```

Isabelle listing 4.16: The translations of the core *Heap* functions

```
typedef Seq = "UNIV::any list set"
  by auto

axiomatization Seq2any any2Seq Seq_UNIV
  where Seq_sub_any:"type_definition (Seq2any::Seq⇒any) (any2Seq::any⇒Seq)
(Seq_UNIV::any set)"

interpretation Seq:type_definition Seq2any any2Seq Seq_UNIV
  by (rule Seq_sub_any)
```

Isabelle listing 4.17: Definition of the *Sequence* type in the translation preamble

seqEmpty : *Sequence*
seqSingleton : *Sequence*
seqLen : *Sequence* $\longrightarrow$ *int*
seqGet : *Sequence* $\times$ *int* $\longrightarrow$ A for any subtype A of type *any*
seqDef : *int* $\times$ *int* $\times$ *Sequence* $\longrightarrow$ *Sequence*
seqConcat : *Sequence* $\times$ *Sequence* $\longrightarrow$ *Sequence*
seqSub : *Sequence* $\times$ *int* $\times$ *int* $\longrightarrow$ *Sequence*
seqReverse : *Sequence* $\longrightarrow$ *Sequence*
seqIndexOf : *Seq* $\times$ *any* $\longrightarrow$ *int*
seqSwap : *Sequence* $\times$ *int* $\times$ *int* $\longrightarrow$ *Sequence*
seqRemove : *Sequence* $\times$ *int* $\longrightarrow$ *Sequence*
seqPerm(Seq, Seq)
seqNPerm(Seq)

Figure 4.4: Signature of *Sequence* functions. Function signatures are separated from their name by“:”. Predicate signatures are shown in brackets.

4.6.2 Functions

The *Sequence* functions in consist of *seqEmpty*, *seqSingleton*, *seqLen*, *seqGet*, *seqDef*, *seqConcat*, *seqSub*, *seqReverse*, *seqIndexOf*, *seqSwap*, *seqRemove*, *seqPerm* and *seqNPerm*. The signatures of these functions in KeY are collated in Figure 4.4 to allow the reader to compare the functions to their translations. The following paragraphs describe the *Sequence* functions and their translations in Isabelle listing 4.18.

seqEmpty The *seqEmpty* function represents the empty *Sequence*. The “List” theory holds an equivalent function **empty** for the *list* type. Thus the translation of *seqEmpty* lifts the **empty** function for use with *Sequence* values.

seqSingleton The *seqSingleton* function returns a *Sequence* containing a single element. While there is no function that returns a *list* containing a single element, such a function can be emulated with a lambda construct, which maps the single element to a *list* containing the single element. This lambda construct can then be lifted to the *Sequence* type.

seqLen The *seqLen* function returns the length of a *Sequence*. This nearly matches the **length** function for *list* values. They differ in their return type. *seqLen* returns an *int* whereas the **length** function returns a *nat*. Thus the conversion function **int** is prepended.

seqGet *seqGet* is used to access the value of the *Sequence* at a specified position. The position is specified by an integer. This again differs from *list* in Isabelle, which supports the similar function **nth** that uses *nat* to specify the position. Additionally, where KeY somewhat uncharacteristically specifies a value used for all accesses of positions outside of a *Sequence* instead of using underdefined functions, Isabelle uses non-exhaustive recursive functions, which do not necessarily have a value. Because of this, the access needs to be gated before passing the values to the Isabelle function **nth**. This is done by the translation in a lambda construct with an if-then-else statement, that maps to the *seqGetOutside* constant, if the *int* is not a valid position of the *Sequence*. The lambda construct is then lifted to define *seqGet*.

seqDef The *seqDef* function is used to generate more complex *Sequence* values. It maps two integers i, j and a term to a new *Sequence*. Additionally it binds a variable of type *int*. The new *Sequence* is constructed by substituting the bound variable in the term for each of the integers between i and j. The new *Sequence* consists of these substituted values in ascending order of the substituted integers. In effect this is similar to mapping the list containing i to j using the supplied term. The **map** function in Isabelle accepts a *'a list* and a function mapping from *'a* to *'b* and returns a *'b list*. To use the term of *seqDef* for the **map** function, the term needs to be converted to a function from *int* to *any*. This is accomplished by turning the term into a lambda construct which binds the bound variable of the *seqDef* occurrence. The *int list* containing i to j is easily constructed with the **upto** function in Isabelle, which returns a *int list* containing all integers starting from the first integer up to the second one. It is written in its infix notation in Isabelle listing 4.18. With **map** and **upto**, the translation transfers the semantics of *seqDef* to the Isablle type *list* and then lifts it to the *Sequence* type.

seqConcat *seqConcat* describes the concatenation of two *Sequences* and finds its counterpart in the Isabelle function **append** for *list*. The translation lifts **append** to the *Sequence* type to define *seqConcat*.

seqSub *seqSub* is used to return a sub-*Sequence* of the *Sequence* from its ith to its jth value - where i and j are integers. There is no direct counterpart to *seqSub* in the "List" theory, so the translation uses the semantics of *seqSub* presented in Ahrendt et al. (2016) to define the translation for *seqSub*. Alternatively, a new function could be introduced, which returns a sublist of an Isabelle *list* using the **take** and **drop** functions, which in turn return the first n or last n elements in a *list* - where n is an integer.

seqReverse *seqReverse* returns a *Sequence*, which holds the same elements in the reverse order of the original *Sequence*. *seqReverse* has a counterpart in the Isabelle function **rev** for the Isabelle type *list*, which is lifted to the *Sequence* type by the translation.

seqIndexOf *seqIndexOf* takes a *Sequence* s and a value of type *any* x and returns the index of the first occurrence of x in s. If x does not occur in s, the value of *seqIndexOf* is not defined. Because of this underdefinedness, it is not possible to use a regular function in Isabelle for the semantics of *seqIndexOf*. The translation first defines a non-exhaustive recursive function on the *list* type, which finds the index of the first occurence of x in the *list*. This new function is then lifted to define **seqIndexOf**.

seqSwap *seqSwap* returns a *Sequence* where the elements at the indices i and j are swapped. There is no equivalent function for the Isabelle type **list** in the "List" theory, but the translation defines such a function in the translation preamble. The definition of the **listSwap** function uses the **list_update** function to set elements at the indices i and j, provided they are valid indices. The translation then lifts this function to define *seqSwap*.

seqRemove *seqRemove* removes the element in the *Sequence* at position k and returns the resulting *Sequence*. If there is no element at position k, the unchanged *Sequence* is returned. *seqRemove* does not have an equivalent function in the "List" theory and thus is defined in the preamble. The semantics are first defined on the *list* type and then lifted to the *Sequence* type.

seqPerm Finally, there are two functions dealing with *Sequence* values and their permutations. The first is *seqPerm*. *seqPerm* is a predicate which holds for all *Sequence* pairs, where the *Sequence* values are permutations of each other. The "List_Permutation" theory contains an equivalent predicate for the *list* type, which the translation lifts to define *seqPerm*.

seqNPerm The second of the *Sequence* permutation functions is *seqNPerm*. *seqNPerm* holds, if the *Sequence* is a permutation of the *Sequence* holding all integers from 0 to the length of the *Sequence*. The translation uses the semantics of *seqNPerm* to define the function directly without first defining an equivalent function on *list*.

This concludes the translation for *Sequence* functions present in KeY. There is a further function *seqDepth* in Ahrendt et al. (2016) to return the nesting depth of a *Sequence*, however this function is not present in KeY and thus is not translated.

4.7 Limitations of the Translation

This section will describe the known limitations of the translation.

First and foremost, the translation currently does not have any way to translate modalities/updates (subsection 2.1.2) on the KeY sequent. There is merit in adding a translation for these, even if it does not translate their contents, because the SMT translation has shown that some proofs are provable even without taking updates into account. Adding this should be straightforward and can likely be copied from the SMT translation with minor adjustments.

Another limitation are proofs that involve proving the wellformedness of a *Heap*. Due to the nature of the translation of *wellFormed* (section 4.5) it is not possible to prove a heap

```
setup_lifting type_definition_Seq

consts
  seqGetOutside::any

lift_definition seqEmpty::"Seq" is "[]".
lift_definition seqSingleton::"any⇒Seq" is "λx. [x]".
lift_definition seqLen::"Seq⇒int" is "int ∘ List.length".
lift_definition seqGet::"Seq⇒int⇒'a::any" is "λs i. (if (0::int)≤i∧i<(int (length
s)) then cast (s ! (nat i)) else cast seqGetOutside)".
lift_definition seqDef::"int⇒int⇒(int⇒any)⇒Seq" is "λle ri e. map e [le..ri -
1]".
lift_definition seqConcat::"Seq⇒Seq⇒Seq" is List.append.

fun seqSub::"Seq⇒int⇒int⇒Seq" where
  "seqSub s i j = seqDef i j (λx. seqGet s x)"

lift_definition seqReverse::"Seq⇒Seq" is List.rev.

primrec (nonexhaustive) listIndexOf::"'a list⇒'a⇒int" where
  "listIndexOf (x#xs) a = (if (x=a) then 0 else 1+(listIndexOf xs a))"
lift_definition seqIndexOf::"Seq⇒any⇒int" is "listIndexOf".

fun listSwap::"'a list⇒int⇒int⇒'a list"
  where "listSwap l i j =
(if ¬(0≤i ∧ i<int (length l) ∧ 0≤j ∧ i<int (length l))
then l
else list_update (list_update l (nat i) (l ! (nat j))) (nat j) (l ! (nat i)))"
lift_definition seqSwap::"Seq⇒int⇒int⇒Seq" is listSwap.

fun listRemove::"'a list⇒nat⇒'a list"
  where "listRemove [] _ = []"
  | "listRemove (x#xs) 0 = xs"
  | "listRemove (x#xs) (Suc k) = x # (listRemove xs k)"
lift_definition seqRemove::"Seq⇒int⇒Seq" is "λs (i::int). (if ¬(0≤i ∧ i<int
(length s)) then s else listRemove s (nat i))".

lift_definition seqPerm::"Seq⇒Seq⇒bool" is List_Permutation.perm.
fun seqNPerm::"Seq⇒bool"
  where "seqNPerm s = seqPerm s (seqDef 0 (seqLen s - 1) (to_any))"
```

Isabelle listing 4.18: Translations of the core *Sequence* functions

is well formed, if it contains arrays. Enhancing *wellFormed* would be a difficult task as this requires redesigning the way arrays are translated. It could be possible to merge the heap definition of Bali with the *Heap* type of KeY to form new semantics for the *wellFormed* predicate, but this falls out of the scope of this work.

Non-core functions in KeY, like *bprod* for bounded products, are currently translated as unknown symbols. Adding translations for these should be fairly straightforward as most of them have a definitional taclet from which the semantics can be extracted. There may be some functions that require more complex translations because they involve arrays or updates. Functions that are not totally defined could also require more thoughtful translation.

5 Implementation

In this chapter we will describe the implementation. As a side effect we will showcase some bugs that have been found in KeY during the implementation. The code of the implementation can be found at https://gitlab.kit.edu/uguxj/key.

5.1 Overview

This section is meant to give an overview of the Implementation. I have implemented the Isabelle translation as a GUI extension of KeY. It is possible to call the Isabelle translation from the context menu of a sequent. The context menu is shown in Figure 5.1. The Isabelle translation extension also implements settings for the path where the translations are saved and the path to the Isabelle directory. The GUI implementation does lack refinement in the sense that the user is given minimal information about any errors during the translation or during **sledgehammer** and the success of the translation can only be inferred by the goal it originated from closing.

The architecture of the translation is strongly based on the way the new SMT translation of KeY is designed. Building the general structure of the translation theories, including the theory header, constant declarations for translated variables and the placement of locales, is handled by the **IsabelleTranslator** class. The actual translation of terms and collection of types and fields present on the sequent are handled by the **IsabelleMasterHandler** class. The **IsabelleMasterHandler** is given a collection of handler classes, which can each handle specific types of terms. Examples of these handlers are **IntegerOpHandler** for handling integers and their operations and the **FieldHandler**, which handles the translation of Field instances. Handlers can have preambles attached to them, which form the "TranslationPreamble" theory.

The master handler collects all predefined functions from the handlers, which should be defined in their preambles. This avoids redefining already present functions like the *store* operation on *Heap* values (see subsection 4.5.2) and losing their semantics. The master handler also collects all *Field* values found on the sequent to form the translation locale assumption that states that the *Field* values on the sequent are distinct (see subsection 4.3.1). In addition the master handler needs to collect all functions and variables which need to be declared in the translation locale. Because KeY sorts can include symbols that cannot be included in a name in Isabelle, the master handler accounts for illegal symbols in function names. In a similar manner, variable and function names need to be checked for illegal symbols.

The **IsabelleTranslator** class also handles saving the "TranslationPreamble" theory and the "Translation" theory to a translation directory specified in the settings. Additionally a "ROOT" file, which describes an Isabelle session (see section 2.2) is added to the translation

directory. This “ROOT” file is used to build the Isabelle session for use in the automated prove attempts. By building the session, the content of the “TranslationPreamble” theory can be accessed quicker, because Isabelle stores heap images of built sessions. Isabelle uses the built images of sessions instead of opening the theory and building the images on demand, which would take longer.

This concludes the general overview of the modus operandi of the translation.

5.2 Scala-Isabelle

In this section we introduce *Scala-Isabelle*, which is used to automatically call Isabelle. Calling Isabelle from other programs was not intended by the developers of Isabelle and is not favored by parts of the Isabelle community. They have favored embedding other programs in ML in Isabelle instead. Unruh (2021) provides an interface to interact with an Isabelle process from a Scala or Java process in *Scala-Isabelle*. The implementation in this work uses *Scala-Isabelle* to call the **try0** and **sledgehammer** tactics from KeY. *Scala-Isabelle* can send and receive values and functions to/from ML. To this end it can also convert ML code into function objects.

To call **sledgehammer** on a theory, the text of the theory is parsed first. The parsing creates a state object on which the **sledgehammer** and **try0** ML code can be executed. **try0** returns a boolean object, while **sledgehammer** returns an “option”, which contains the successful tactics found by **sledgehammer**.

A disadvantage of this approach is that building the session containing the “TranslationPreamble” theory is executed before every **sledgehammer** attempt. Perhaps even worse is that the underlying logic of the Isabelle process cannot be set and thus additional time is required to create the object representing the “Translation” theory before calling **sledgehammer**.

This can be circumvented by storing the Isabelle process object and the theory object. The theory object is not loaded from the translation theory file, but generated from the header of the “Translation” theory. Executing the ML functions that call **sledgehammer** and **try0** does not change the theory object, which is why it can be reused for another proof. The **IsabelleLauncher** class implements reusing the process and theory instances to speed up calling **sledgehammer** on multiple goals.

5.3 Alternatives to Scala-Isabelle

Instead of using *Scala-Isabelle* to call **sledgehammer** it is possible to use *Mirabelle*. *Mirabelle* is an Isabelle tool for benchmarking the proof steps in a theory. It can only be applied to closed theories and as such its applicability is not immediately apparent, because the “Translation” theory contains the open proof obligation that was translated. With some workarounds, that involve axiomatizing the proof obligation with a lemma that cannot be used by **sledgehammer**, it is possible to use *Mirabelle* in place of *Scala-Isabelle*. However, this requires additional theory splitting to ensure that there are no other proofs *Mirabelle* could be trying in addition to the translated proof obligation. These additional

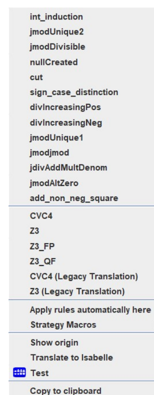

Figure 5.1: Context Menu of the Implementation

requirements for the translation theory would make interactive proving attempts very confusing. Additionally, *Mirabelle* was not designed with the application of calling tactics on open goals to search for proofs in mind. Therefore it was not chosen.

It should also be possible to emulate user inputs on a command line to interact with an Isabelle process, but this likely does not improve upon *Scala-Isabelle* in most cases. Where this has an advantage, enhancing *Scala-Isabelle* should be preferred instead of building a new interface between Java and Isabelle.

5.4 Issues found in KeY

Finally I want to address some bugs in the KeY system, which became apparent during the implementation.

For one KeY sequents can contain multiple functions of the same name. This is unintentional especially in the case of the “self” variable, which is meant to represent the Java object whose JML contract is being proven. The implementation of the translation does not account for duplicate functions and therefore errors can occur during the **sledgehammer** call. Future additions to the implementation could prevent both this and other duplicate variable translations.

Some of the examples from KeY suffer from the duplicate variable problem as well, where the SMT translation cannot differentiate between symbols introduced from the JML contract and Java code and symbols already present as part of the JavaDL in KeY (subsection 2.1.2).

During the design process of the translation of integers (section 4.2), it became apparent that the SMT translation of KeY lacked a proper translation for the modulo operation. The modulo translation for SMT could be added easily as KeY already had a translation in place for division. Thus modulo could be translated similarly due to Euclid’s theorem.

6 Evaluation

This chapter discusses hypotheses about the performance of the implementation, the results of my testing and the conclusions drawn from it.

6.1 Process of Evaluation

I tested my implementation on most of the examples in the KeY system. Some examples were not tested because the KeY API could not load their contracts or the SMT preparation macro could not be applied. Subgoals in which the SMT preparation macro could not fully resolve the heap updates were also not tested. Subgoals which were already solved by the SMT preparation macro or the propositional expansion macro, were not tested.

I tested my implementation against the automation of KeY and the Z3 solver (De Moura and Bjørner, 2008) as a representative of the SMT translation in KeY. Each of these were given a timeout of 30 seconds. The KeY automation used the default strategy of KeY with no restriction to the number of proof steps.

In the case of Isabelle this timeout is applied first to **try0** and then to **sledgehammer**, leading to the possibility of up to a 60 second timeout. In practice this did not occur often, because **try0** usually concludes within less than 5 seconds that it could not find a proof. The additional time does not give Isabelle an advantage as the calls to **try0** and **sledgehammer** are separated.

The testing was performed on a machine running 64-bit Ubuntu 22.04.4 LTS on a Ryzen 5 5600 6-core processor at 4.6 GHz with 32 GiB of RAM at 3600MHz. section 6.1 holds a summary of the results. The columns contain the number of successful proofs for KeY, Z3 and Isabelle. Table 6.2 holds the average time spent searching for a successful proof by each prover - note that unsuccessful proofs are not accounted for in this table.

A more expansive list of results can be found in section A.1.

6.2 Hypotheses and Conclusions

Since the translation does not remove any semantics and at most renames a function, my initial assumption was that the automation of Isabelle would be successful in finding a proof if the direct call from KeY to SMT solvers was successful in finding a proof. The results of my testing contradict this assumption. Especially in cases where there are nested *if-then-else* statements that need to be resolved, Isabelle seems to struggle to find a proof. The translations generated by the SMT translation contain additional axioms, which might affect this difference in proving strength too. Tests on the difference these axioms make were out of the scope of this work due to time restrictions.

The initial assumption that Isabelle offers stronger automated reasoning capabilities for bounded sums appears to be false. Proofs involving bounded sums require induction in Isabelle. This induction is not done by the automation of Isabelle. The induct tactics also require special parameters to use the inbuilt induction rules for int and other predefined types. The induct tactic defaults to the induction rules generated by the interpretation of int and other predefined types as any subtypes (section 4.1), which are not useful for induction over sums.

An initial assumption was that Isabelle offers stronger automated reasoning for proofs involving *Heap* and sets. Based on the present examples it cannot be said that Isabelle offers stronger automated reasoning capabilities than the automation of KeY for sets. This may in part be due to the examples in KeY being largely provable by KeY already. More complex examples could offer more insights, but were not in the scope of this work due to time constraints.

Isabelle works especially poorly on the examples “MiniExamples”, “MethodContracts” and “NewObjects”. These heavily involve invariants of JML contracts. These are translated without semantics, causing Isabelle to lack the required information to close these proofs. There may be merit in adding a KeY strategy macro to automatically remove them. Alternatively the taclets containing the semantics of invariants could be translated to Isabelle at time of translation, but an automatic translation of taclets is out of the scope of this work.

The main hypothesis of this work was that Isabelle offered an advantage compared to the SMT translation and the KeY automation. The results compiled in section 6.1 seem to suggest that the Isabelle translation offers no advantage and is a weaker automated tool than the SMT translation and KeY automation. When counting the number of goals closed by each combination of provers (Table 6.3), it is apparent, that the automation of KeY is very suitable for the chosen examples. This does not come as a surprise as the examples were taken from the example library of KeY. Still there are 104 goals, which the KeY automation could not close. Of these 104 goals Isabelle manages to close 10. SMT manages to close 19. It therefore appears that Isabelle offers an advantage when compared to the KeY automation. There is one goal in the “PairInsertionSort” example, that was only closed by Isabelle and not by the SMT translation or the KeY automation. The proof for this goal can be found by SMT too, but requires a timeout of around 10 minutes. Perhaps more extensive testing on other examples can find additional proofs, which Isabelle manages to close, but the other provers do not. Because there is an example in which Isabelle closes a proof no other prover could close, it appears that the Isabelle translation does offer an advantage over the SMT translation and the KeY automation.

In conclusion the Isabelle translation should serve as a valuable addition to the toolbox of the KeY user for finding proofs automatically. Due to its longer computation times compared to the SMT translation (see Table 6.2) and the lower overall rate of success it would seem prudent to first try the SMT translation and the KeY automation before calling Isabelle to close less computationally intense subgoals.

6.3 Issues in KeY found during evaluation

The issue of duplicate variables in the sequent mentioned in section 5.4 affected the SMT translation in a different way in the *removeDups* example. Where the Isabelle translation

Table 6.1: Number of goals translated and closed by each prover. Goals, that were not translated, are not counted toward the number of closed goals for other provers.

Example	# Goals	# Translated	# KeY	# Z3	# Isabelle
SumAndMax	18	18	18	13	15
information_flow	18	18	17	10	10
BinarySearch	19	19	17	19	12
fm12_01_LRS	19	19	16	19	18
AddAndMultiply	22	22	22	14	12
Sum	37	37	37	30	32
quicksort	47	47	45	33	24
Transactions	51	51	45	18	14
removeDups	57	57	57	41[i]	47
list_ghost	59	59	59	30	21
ToyVoting	78	76	72	19	15[ii]
SmansEtAl	99	99	99	21	13[iii]
Quicktour	106	104	104	68	58
MiniExamples	106	106	92	47	5
MethodContracts	130	130	127	34	24
block_loop_contracts/Simple	194	194	189	123[iv]	170
NewObjects	214	208	203	65	24[v]
LoopInvariants	223	181	177	150	120
PairInsertionSort	240	240	214	209	139
list_seq	298	298	278	200	146
ToyBanking	457	446	436	326	319
All Examples combined	2492	2429	2324	1489	1238

[i] There are 13 goals, on which Z3 encountered an error. [ii] Duplicate Variable caused 2 errors.
[iii] Duplicate Variable caused 3 errors.
[iv] There are 55 goals, on which Z3 encountered an error. These are caused by a bitvector issue in the SMT translation.
[v] Duplicate Variable caused 3 errors

Table 6.2: Average time spent searching for a successful proof (in ms) by the provers

Example	KeY	Z3	Isabelle
SumAndMax	122	39	10 145
information_flow	100	36	12 242
BinarySearch	132	77	13 087
fm12_01_LRS	100	32	12 560
AddAndMultiply	100	43	5 879
Sum	100	44	11 447
quicksort	847	516	14 357
Transactions	539	41	6 400
removeDups	134	361	10 984
list_ghost	114	43	15 333
ToyVoting	377	42	9 968
SmansEtAl	155	41	18 784
Quicktour	127	34	11 636
MiniExamples	100	39	6 226
MethodContracts	110	36	11 854
block_loop_contracts/Simple	111	33	8 402
NewObjects	465	44	8 781
LoopInvariants	135	37	4 413
PairInsertionSort	2089	502	14 126
list_seq	535	51	12 919
ToyBanking	841	71	11 158

Table 6.3: Number of successful proofs by prover combination

Provers that found proofs	#
All provers	1 153
KeY and Isabelle	21
KeY and Z3	315
KeY	760
Z3 and Isabelle	9
Isabelle	1
Z3	10
No prover	84

is affected by duplicate declarations for these variables, the SMT translation cannot differentiate between the *length* function to determine the length of an object and *length* as a program variable. Thus the SMT translation fails for some of the goals in *removeDups*.

There were also 8 subgoals among the chosen examples, which caused errors in the translation due to duplicate variables appearing on the sequent.

There is an issue with bitvectors in *block_loop_contracts/Simple*, which causes the SMT translation to fail on multiple goals.

7 Conclusion

7.1 Future Work

As this work was a foray into an unexplored direction, there are many open avenues.

Improving the translation and automation The first to come to mind are improvements to the translation, which could not be pursued in this work due to time restrictions. Improving the translation can come in the form of adding some of the less integral functions in KeY that do not appear in the translation. Fine tuning the parameters given to **sledgehammer** could be more successful in improving the automation of the translation. Changing the rules for the handling of if-then-else constructs could be of particular interest here. Improving the **sledgehammer** parameters can also come in the form of adding better tags to the lemmas in the translation. By using certain tags for lemmas they are added to the default repertoire of certain tactics, like "[simp]" for the **simp** tactic, but there are many other possibilities for which tags to use.

Another approach to improving the translation is adding more taclets of KeY to the preamble. While adding the taclets could be done manually for select taclets, due to the large collection of taclets in KeY an automated approach would be preferred. Finding a way to automatically transfer taclets to Isabelle would also allow to extensively check the correctness of the taclet base of KeY.

This work intentionally did not try to translate update applications of KeY on the sequent. If not for the SMT preparation macro failing to fully resolve all updates in the sequent for some problems, this would not affect the translation. Because the SMT preparation macro does not fully resolve all updates, it might be worthwhile to research a macro that is able to resolve the updates in more cases. Alternatively there may be merit in finding a method for translating updates as well.

Better KeY integration Future work may resolve around better integrating Isabelle in KeY. Some proofs may not be provable by Isabelle, but Isabelle can reduce the proof to an easier form. It is feasible that these reduced proofs could then be closed by KeY. Therefore there could be merit in re-translating open proof goals from Isabelle to KeY. Likewise helpful lemmas proven by the user in Isabelle could be of use in KeY. In this way KeY's reasoning could be strengthened.

It should also be possible to use Isabelle for the "focus" mechanism of the SMT translation. The "focus" mechanism allows the user to hide all antecedent and succedent formulae that are not part of the unsat core found by SMT. As **sledgehammer** indicates which rules and which assumptions of the theorem were used, it should be possible to at least reduce

the number of formulae in the antecedent in KeY. Perhaps it is even possible to direct the automation of KeY toward using select taclets.

As the testing has shown, there are quite a few proofs in KeY, which require invariants and method contracts. These were translated as unknown symbols. At present there is no automated way of resolving or adding the semantics of these invariants and contracts to the sequent or the translation. An automated translation of taclets like the one mentioned earlier in this section could also solve this problem by translating the invariant taclets. As the testing has shown, there are some goals which cannot be closed by **sledgehammer** and require user interaction. Therefore it is beneficial to allow KeY to use the results of interactive proving in Isabelle. This would require extensive checking of the translation theory file to ensure that the proofs are correct, as the user could have introduced additional axioms to close the proof.

7.2 Summary

This work presented a translation of KeY proof obligations without modalities and updates to Isabelle. It presented the translation for each of the core parts of KeY. This work provided an implementation of this translation and a way to utilize the automated proving methods of Isabelle in KeY. This work tested these against the already present automated proving methods in KeY on over 2 400 proof obligations and showed that there is merit in adding the Isabelle translation to the automated toolset of KeY. Additionally this work laid the groundwork for using Isabelle in other ways than automatic proof searching.

The current implementation, once its user interface has been refined, should serve as a valuable addition to KeY.

Bibliography

Ahrendt, Wolfgang et al., eds. (2016). *Deductive Software Verification - The KeY Book: From Theory to Practice.* Vol. 10001. Lecture Notes in Computer Science. Springer. DOI: 10.1007/978-3-319-49812-6.

Bian, Jinting et al. (2021). "Integrating ADTs in KeY and Their Application to History-Based Reasoning". In: *Formal Methods.* Ed. by Marieke Huisman, Corina Păsăreanu, and Naijun Zhan. Cham: Springer International Publishing, pp. 255–272. ISBN: 978-3-030-90870-6.

Boute, Raymond T. (Apr. 1992). "The Euclidean definition of the functions div and mod". In: *ACM Trans. Program. Lang. Syst.* 14.2, pp. 127–144. ISSN: 0164-0925. DOI: 10.1145/128861.128862. URL: https://doi.org/10.1145/128861.128862.

De Moura, Leonardo and Nikolaj Bjørner (2008). "Z3: an efficient SMT solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, pp. 337–340. ISBN: 3540787992.

From, Asta Halkjær, Anders Schlichtkrull, and Jørgen Villadsen (2021). "A Sequent Calculus for First-Order Logic Formalized in Isabelle/HOL". English. In: *Proceedings of the 36th Italian Conference on Computational Logic.* CEUR Workshop Proceedings. 36th Italian Conference on Computational Logic, CILC 2021 ; Conference date: 07-09-2021 Through 09-09-2021. CEUR-WS, pp. 107–121. URL: http://www.ailab.unipr.it/cilc21/.

Huffman, Brian and Ondřej Kunčar (2013). "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL". In: *Certified Programs and Proofs.* Ed. by Georges Gonthier and Michael Norrish. Cham: Springer International Publishing, pp. 131–146. ISBN: 978-3-319-03545-1.

Luo, Z., S. Soloviev, and T. Xue (2013). "Coercive subtyping: Theory and implementation". In: *Information and Computation* 223, pp. 18–42. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2012.10.020. URL: https://www.sciencedirect.com/science/article/pii/S0890540112001757.

Nipkow, Tobias, Lawrence C Paulson, and Markus Wenzel (2002). *Isabelle/HOL: a proof assistant for higher-order logic.* Vol. 2283. Springer Science & Business Media.

Pusch, Cornelia (1999). "Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 89–103. ISBN: 978-3-540-49059-3.

Trentelman, K. (2005). "Proving correctness of JavaCard DL taclets using Bali". In: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pp. 160–169. DOI: 10.1109/SEFM.2005.37.

Unruh, Dominique (2021). *Scala-Isabelle.* https://github.com/dominique-unruh/scala-isabelle.

A Appendix

A.1 Extended Evaluation Statistics

The proof states are as follows:
CLOSED means the prover found a proof.
OPEN means the prover did not manage to find a proof.
ERROR means there was an error (see section 6.3) during the proof attempt.
UNKNOWN means the translation failed, because there were updates on the sequent.

Table A.1: Detailed proof states for all provers (1)

Example	KeY	Z3	Isabelle	#
AddAndMultiply	CLOSED	CLOSED	CLOSED	12
AddAndMultiply	CLOSED	CLOSED	OPEN	2
AddAndMultiply	CLOSED	OPEN	OPEN	8
BinarySearch	CLOSED	CLOSED	CLOSED	12
BinarySearch	CLOSED	CLOSED	OPEN	5
BinarySearch	OPEN	CLOSED	OPEN	2
LoopInvariants	CLOSED	CLOSED	CLOSED	120
LoopInvariants	CLOSED	CLOSED	OPEN	30
LoopInvariants	CLOSED	OPEN	OPEN	58
LoopInvariants	CLOSED	OPEN	UNKNOWN	4
LoopInvariants	OPEN	OPEN	OPEN	9
LoopInvariants	OPEN	OPEN	UNKNOWN	2
MethodContracts	CLOSED	CLOSED	CLOSED	24
MethodContracts	CLOSED	CLOSED	OPEN	10
MethodContracts	CLOSED	OPEN	OPEN	93
MethodContracts	OPEN	OPEN	OPEN	3
MiniExamples	CLOSED	CLOSED	CLOSED	5
MiniExamples	CLOSED	CLOSED	OPEN	42
MiniExamples	CLOSED	OPEN	OPEN	45
MiniExamples	OPEN	OPEN	OPEN	14

Table A.2: Detailed proof states for all provers (2)

Example	KeY	Z3	Isabelle	#
NewObjects	CLOSED	CLOSED	CLOSED	24
NewObjects	CLOSED	CLOSED	ERROR	2
NewObjects	CLOSED	OPEN	ERROR	1
NewObjects	CLOSED	CLOSED	OPEN	39
NewObjects	CLOSED	OPEN	OPEN	137
NewObjects	CLOSED	OPEN	UNKNOWN	6
NewObjects	OPEN	OPEN	OPEN	5
PairInsertionSort	CLOSED	CLOSED	CLOSED	133
PairInsertionSort	CLOSED	OPEN	CLOSED	3
PairInsertionSort	CLOSED	CLOSED	OPEN	66
PairInsertionSort	CLOSED	OPEN	OPEN	12
PairInsertionSort	OPEN	CLOSED	CLOSED	2
PairInsertionSort	OPEN	OPEN	CLOSED	1
PairInsertionSort	OPEN	CLOSED	OPEN	8
PairInsertionSort	OPEN	OPEN	OPEN	15
Quicktour	CLOSED	CLOSED	CLOSED	58
Quicktour	CLOSED	CLOSED	OPEN	10
Quicktour	CLOSED	OPEN	OPEN	36
Quicktour	OPEN	OPEN	UNKNOWN	2
SmansEtAl	CLOSED	CLOSED	CLOSED	13
SmansEtAl	CLOSED	OPEN	ERROR	3
SmansEtAl	CLOSED	CLOSED	OPEN	8
SmansEtAl	CLOSED	OPEN	OPEN	75
Sum	CLOSED	CLOSED	CLOSED	27
Sum	CLOSED	OPEN	CLOSED	5
Sum	CLOSED	CLOSED	OPEN	3
Sum	CLOSED	OPEN	OPEN	2
SumAndMax	CLOSED	CLOSED	CLOSED	13
SumAndMax	CLOSED	OPEN	CLOSED	2
SumAndMax	CLOSED	OPEN	OPEN	3
ToyBanking	CLOSED	CLOSED	CLOSED	315
ToyBanking	CLOSED	CLOSED	OPEN	7
ToyBanking	CLOSED	OPEN	OPEN	114
ToyBanking	CLOSED	CLOSED	UNKNOWN	8
ToyBanking	CLOSED	OPEN	UNKNOWN	3
ToyBanking	OPEN	CLOSED	CLOSED	4
ToyBanking	OPEN	OPEN	OPEN	6

Table A.3: Detailed proof states for all provers (3)

Example	KeY	Z3	Isabelle	#
ToyVoting	CLOSED	CLOSED	CLOSED	15
ToyVoting	CLOSED	OPEN	ERROR	2
ToyVoting	CLOSED	CLOSED	OPEN	4
ToyVoting	CLOSED	OPEN	OPEN	51
ToyVoting	CLOSED	OPEN	UNKNOWN	2
ToyVoting	OPEN	OPEN	OPEN	4
Transactions	CLOSED	CLOSED	CLOSED	14
Transactions	CLOSED	OPEN	OPEN	27
Transactions	CLOSED	CLOSED	OPEN	4
Transactions	OPEN	OPEN	OPEN	6
block_loop_contracts/Simple	CLOSED	CLOSED	CLOSED	118
block_loop_contracts/Simple	CLOSED	ERROR	CLOSED	41
block_loop_contracts/Simple	CLOSED	OPEN	CLOSED	11
block_loop_contracts/Simple	CLOSED	CLOSED	OPEN	5
block_loop_contracts/Simple	CLOSED	ERROR	OPEN	11
block_loop_contracts/Simple	CLOSED	OPEN	OPEN	3
block_loop_contracts/Simple	OPEN	ERROR	OPEN	1
block_loop_contracts/Simple	OPEN	OPEN	OPEN	4
fm12_01_LRS/lcp.key	CLOSED	CLOSED	CLOSED	15
fm12_01_LRS/lcp.key	CLOSED	CLOSED	OPEN	1
fm12_01_LRS/lcp.key	OPEN	CLOSED	CLOSED	3
information_flow	CLOSED	CLOSED	CLOSED	10
information_flow	CLOSED	OPEN	OPEN	7
information_flow	OPEN	OPEN	OPEN	1
list_ghost	CLOSED	CLOSED	CLOSED	21
list_ghost	CLOSED	CLOSED	OPEN	9
list_ghost	CLOSED	OPEN	OPEN	29
list_seq	CLOSED	CLOSED	OPEN	54
list_seq	CLOSED	OPEN	OPEN	78
list_seq	OPEN	OPEN	OPEN	20
quicksort	CLOSED	CLOSED	CLOSED	24
quicksort	CLOSED	CLOSED	OPEN	9
quicksort	CLOSED	OPEN	OPEN	12
quicksort	OPEN	OPEN	OPEN	2
removeDups	CLOSED	CLOSED	CLOSED	34
removeDups	CLOSED	ERROR	CLOSED	13
removeDups	CLOSED	CLOSED	OPEN	7
removeDups	CLOSED	OPEN	OPEN	1
removeDups	CLOSED	ERROR	OPEN	2

Isabelle listing A.1: Complete TranslationPreamble theory

A.2 Complete TranslationPreamble theory

```
theory TranslationPreamble imports Main "HOL-Combinatorics.List_Permutation" begin

locale jArithmetics =
  fixes jDiv::"int⇒int⇒int"
  assumes jDiv_def [simp]: "b≠0 ⟹ jDiv a b =
(if ((a≤0 ∧ b<0) ∨ (a≥0 ∧ b>0) ∨ (b dvd a)) then (a div b)
else ((a div b) + 1))"

fixes euclMod::"int⇒int⇒int"
assumes eucl_Mod_def [simp]: "l≠0 ⟹ euclMod k l = (if (k mod l < 0) then ((k mod l)
+ abs(l))
else (k mod l))"
begin

definition int_HALFRANGE::int where [simp, intro]:"int_HALFRANGE=2^31"
definition int_RANGE::int where [simp, intro]:"int_RANGE=2^32"
definition int_MIN::int where [simp, intro]:"int_MIN=-(2^31)"

lemma jDiv_spec_requirement:
  fixes a::int
  fixes b::int
  assumes "b≠0"
  shows "abs(jDiv a b * b) ≤ abs(a) ∧ abs(a) < abs(jDiv a b * b) + abs(b)"
proof -
  have "abs(jDiv a b * b) + abs(b) ≡ abs(jDiv a b) * abs(b) + abs(b)"
    by (simp add: abs_mult)
  also have "... ≡ (abs(jDiv a b) + 1) * abs(b)"
    by algebra
  finally have dist_jDiv_largest: "abs(jDiv a b * b) + abs(b) ≡ abs((abs(jDiv a b) + 1)
* b)"
    by (simp add: abs_mult)
  consider (is_Div) "(a≥0 ∧ b>0) ∨ (a≤0) ∧ (b<0) ∨ (b dvd a)" | (not_Div) "(a<0∧b>0
∨ a>0∧b<0) ∧ ¬(b dvd a)" using assms by linarith
  then show ?thesis
  proof cases
    case is_Div
    then have jDiv_eq_div: "jDiv a b = a div b" using assms by auto
    consider (b_dvd_a) "b dvd a" | (both_pos) "(a>0 ∧ b>0)" | (both_neg) "(a<0) ∧ (b<0)"
using is_Div by (metis dvd_0_right leD linorder_neqE_linordered_idom)
    then show ?thesis
    proof cases
      case (b_dvd_a)
      then have "a div b * b = a" using assms by simp
```

```
      then show ?thesis by (simp add: assms jDiv_eq_div)
    next
      case (both_pos)
      then have "abs(a div b * b) ≤ abs(a) ≡ a div b * b ≤ a"
        by (simp add:pos_imp_zdiv_nonneg_iff)
      also
      have "... ≡ a div b * b ≤ a div b * b + a mod b" using both_pos by simp
      also
      have "... ≡ 0 ≤ a mod b" using both_pos by linarith
      finally
      have req:"abs(jDiv a b * b) ≤ abs(a)"
        by (simp add: both_pos jDiv_eq_div)
      have "a mod b < abs(b)" using both_pos by auto
      then have "a < a div b * b + abs(b)"
        by (metis add.commute add_less_cancel_right mod_mult_div_eq mult.commute)
      then have largest:"abs(a) < abs(jDiv a b * b) + abs(b)" using both_pos by auto
      then show ?thesis using req largest by blast
    next
      case (both_neg)
      then have "abs(a div b * b) ≤ abs (a) ≡ -(a div b * b) ≤ -a"
        by (simp add: div_int_pos_iff mult_nonneg_nonpos)
      also
      have "... ≡ a div b * b ≥ a div b * b + a mod b" by simp
      also
      have "... ≡ 0 ≥ a mod b" by linarith
      finally
      have req:"abs(jDiv a b * b) ≤ abs(a)"
        by (simp add: both_neg jDiv_eq_div)
      have "abs(a) < abs(jDiv a b * b) + abs(b) ≡ -((a div b * b) + a mod b) < abs(a
div b * b) - b" using both_neg by simp
      also have "... ≡ -((a div b * b) + a mod b) < -(a div b * b) - b" using both_neg
by (simp add: div_int_pos_iff mult_nonneg_nonpos)
      also have "... ≡ (a div b * b) + a mod b > (a div b * b) + b" by linarith
      also have "... ≡ a mod b > b" by linarith
      finally have largest:"abs(a) < abs(jDiv a b * b) + abs(b)" using both_neg neg_mod_bound
by blast
      then show ?thesis using req largest by blast
      qed
  next
    case not_Div
    then have jDiv_eq_divplus: "jDiv a b = (a div b) + 1" using assms by auto
    then have "abs(jDiv a b * b) ≤ abs(a) ≡ abs(a div b * b + b) ≤ abs(a div b * b
+ a mod b)"
      by (simp add: distrib_left mult.commute)
    consider (b_neg) "b<0∧a>0" | (b_pos) "b>0∧a<0" using assms not_Div by linarith
    then show ?thesis
    proof cases
```

```
    case (b_neg)
    then have quotient_neg:"a div b < 0"
      by (simp add: neg_imp_zdiv_neg_iff)
    then have abs_jDiv:"abs((jDiv a b) * b) = (a div b + 1) * b" using b_neg jDiv_eq_divplus
      by (simp add: mult_nonpos_nonpos)
    then have "abs(jDiv a b * b) ≤ abs(a) ≡ (a div b + 1) * b ≤ a div b * b + a mod
b"
      by (simp add: abs_of_pos b_neg jDiv_eq_divplus)
    also have "... ≡ a div b * b + b ≤ a div b * b + a mod b"
      by (simp add: distrib_left mult.commute)
    also have "... ≡ b ≤ a mod b"
      by linarith
    finally have requirement:"abs(jDiv a b * b) ≤ abs(a)" using b_neg neg_mod_bound
order_less_imp_le
      by blast

    have mod_le_zero:"a mod b < 0" using mod_eq_0_iff_dvd not_Div b_neg neg_mod_sign
      by (metis linorder_not_less verit_la_disequality)

    have "abs(a) < abs(jDiv a b * b) + abs(b) ≡ a < ((a div b + 1) * b) + abs(b)" us-
ing jDiv_eq_divplus b_neg abs_jDiv
      by simp
    also have "... ≡ a < a div b * b + b + abs b"
      by (simp add: distrib_left mult.commute)
    also have "... ≡ a < a div b * b" using b_neg abs_of_neg
      by simp
    also have "... ≡ a div b * b + a mod b < a div b * b" using mult_div_mod_eq
      by simp
    also have "... ≡ a mod b < 0"
      by linarith
    finally have largest:"abs(a) < abs(jDiv a b * b) + abs(b)" using mod_le_zero
      by blast

    show ?thesis using requirement largest by blast
  next
    case (b_pos)
    then have "a div b < 0"
      by (simp add: pos_imp_zdiv_neg_iff)
    then have abs_jDiv:"abs((jDiv a b) * b) = -((a div b + 1) * b)" using b_pos jDiv_eq_divplus
      by (simp add: mult_le_0_iff)
    then have "abs(jDiv a b * b) ≤ abs(a) ≡ -((a div b + 1) * b) ≤ -(a div b * b
+ a mod b)"
      by (simp add: abs_of_neg b_pos jDiv_eq_divplus)
    also have "... ≡ (a div b + 1) * b ≥ a div b * b + a mod b"
      by simp
    also have "... ≡ a div b * b + b ≥ a div b * b + a mod b"
      by (simp add: distrib_left mult.commute abs_of_neg b_pos jDiv_eq_divplus)
```

```
      also have "... ≡ b ≥ a mod b" by linarith
      finally have requirement:"abs(jDiv a b * b) ≤ abs(a)" using b_pos pos_mod_bound
order_less_imp_le
        by blast

      have mod_greater_zero:"a mod b > 0" using mod_eq_0_iff_dvd not_Div
        by (metis b_pos mod_int_pos_iff order_antisym_conv verit_comp_simplify1(3))

      have "abs(a) < abs(jDiv a b * b) + abs(b) ≡ -a < -((a div b + 1) * b) + abs(b)"
using jDiv_eq_divplus b_pos abs_jDiv
        by simp
      also have "... ≡ -a < -(a div b * b) - b + abs b"
        by (simp add: distrib_left mult.commute)
      also have "... ≡ a > a div b * b" using b_pos abs_of_pos
        by simp
      also have "... ≡ a div b * b + a mod b > a div b * b" using mult_div_mod_eq
        by simp
      also have "... ≡ a mod b > 0"
        by linarith
      finally have largest:"abs(a) < abs(jDiv a b * b) + abs(b)"
        using mod_greater_zero by blast

      show ?thesis using requirement largest by blast
    qed
  qed
qed

fun jMod::"int⇒int⇒int" where
"jMod a b = a - (jDiv a b)*b"

lemma jMod_jDiv_eq:
  fixes a::int
  fixes b::int
  assumes "b≠0"
  shows "a = (jDiv a b)*b + jMod a b"
  by simp

fun moduloInt::"int⇒int"
  where "moduloInt a = int_MIN + ((int_HALFRANGE + a) mod (int_RANGE))"

fun jAdd::"int⇒int⇒int"
  where "jAdd a b = moduloInt (a+b)"

fun jSub:: "int⇒int⇒int" where
    "jSub a b = moduloInt (a-b)"

fun jMul:: "int⇒int⇒int" where
```

```
    "jMul a b = moduloInt (a*b)"

lemma euclMod_spec:
  fixes a::int
  fixes b::int
  assumes "b≠0"
  shows "0≤euclMod a b ∧ euclMod a b < abs(b)"
proof -
  consider (mod_neg) "a mod b < 0" | (mod_nonneg) "a mod b≥0" by linarith
  then show ?thesis
  proof cases
    case (mod_neg)
    then have "0≤euclMod a b ∧ euclMod a b < abs(b) ≡ 0≤a mod b + abs(b) ∧ a mod b
+ abs(b) < abs(b)" using assms
      by auto
    also have "... ≡ -abs(b)≤a mod b ∧ a mod b + abs(b) < abs(b)"
      by linarith
    also have "... ≡ abs(b) ≥ abs(a mod b) ∧ a mod b + abs(b) < abs(b)"
      using mod_neg by linarith
    also have "... ≡ a mod b + abs(b) < abs(b)"
      by (simp add: abs_mod_less assms dual_order.order_iff_strict)
    finally show ?thesis
      using mod_neg by auto
  next
    case (mod_nonneg)
    then have "0≤euclMod a b ∧ euclMod a b < abs(b) ≡ 0≤a mod b ∧ a mod b < abs(b)"
using assms
      by auto

    then show ?thesis
      by (metis abs_mod_less abs_of_nonneg assms mod_nonneg)
  qed
qed

fun euclDiv::"int⇒int⇒int" where
"(euclDiv k l) = (k - euclMod k l) div l"

lemma euclMod_euclDiv_eq:
  fixes a::int
  fixes b::int
  assumes "b≠0"
  shows "a = euclDiv a b * b + euclMod a b"
proof -
  consider (mod_le0) "a mod b<0" | (mod_geq0) "a mod b≥0" by linarith
  then show ?thesis
proof cases
  case mod_le0
```

```
  then have "euclMod a b = a mod b + abs(b)" using assms
    by simp
  then have "euclMod a b = a - ((a div b) * b) + abs(b)"
    by (metis minus_div_mult_eq_mod)
  then have "(euclDiv a b) = (a div b * b) div b - (abs(b)) div b"
    by simp
  then have "euclDiv a b = (a div b) - sgn(b)"
    by (metis div_by_0 linordered_idom_class.abs_sgn nonzero_mult_div_cancel_left nonzero_mult_div
sgn_0)
  then have "euclDiv a b * b = (a div b)*b - abs b"
    by (metis linordered_idom_class.abs_sgn mult.commute right_diff_distrib')
  then show ?thesis using assms
    by auto
next
  case mod_geq0
  then have euclMod_eq_mod:"euclMod a b = a mod b" using assms
    by simp
  then have "euclDiv a b = a div b"
    by (simp add: minus_mod_eq_mult_div)
  then show ?thesis using euclMod_eq_mod
    by auto
  qed
qed
end


declare [[coercion_enabled]]
declare [[coercion_map image]]
typedecl any

consts
  bottom::"any"

specification (bottom) "bottom = bottom"
  by simp

lemma bottom_in_any:"bottom ∈ (UNIV::any set)"
  by simp

typedef javaDL_type = "(UNIV::any set set)"
  by auto

setup_lifting type_definition_javaDL_type
lift_definition typeof::"any⇒javaDL_type⇒bool" is Set.member.
lift_definition subtype::"javaDL_type⇒javaDL_type⇒bool" is Set.subset_eq.
lift_definition strict_subtype::"javaDL_type⇒javaDL_type⇒bool" is Set.subset.
lift_definition disjointTypes::"javaDL_type⇒javaDL_type⇒bool" is Set.disjnt.
```

```
consts
  int_UNIV::"any set"
  int2any::"int⇒any"
  any2int::"any⇒int"

axiomatization where int_sub_any[simp]:"type_definition int2any any2int (int_UNIV)"
declare [[coercion int2any]]

interpretation int:type_definition int2any any2int int_UNIV
  by simp


definition int_type::"javaDL_type" where "int_type ≡ Abs_javaDL_type (UNIV::int set)"

consts
  bool_UNIV::"any set"
  bool2any::"bool⇒any"
  any2bool::"any⇒bool"

axiomatization where bool_sub_any[simp]:"type_definition bool2any any2bool (bool_UNIV)"
declare [[coercion bool2any]]

interpretation bool:type_definition bool2any any2bool bool_UNIV
  by simp


definition bool_type::"javaDL_type" where "bool_type ≡ Abs_javaDL_type (UNIV::bool set)"



consts
  java_lang_Object_UNIV::"any set"


specification (java_lang_Object_UNIV) "java_lang_Object_UNIV ⊆ (UNIV::any set)"
  "bottom:java_lang_Object_UNIV"
  by auto

lemma java_lang_Object_UNIV_specification:"java_lang_Object_UNIV ⊆ (UNIV::any set) ∧
      bottom:java_lang_Object_UNIV"
  by (metis (mono_tags, lifting) java_lang_Object_UNIV_def UNIV_I subset_UNIV verit_sko_ex_indirect)


typedef java_lang_Object = "java_lang_Object_UNIV"
  morphisms java_lang_Object2any any2java_lang_Object
  using java_lang_Object_UNIV_specification by auto
```

```
declare [[coercion java_lang_Object2any]]

definition java_lang_Object_type::"javaDL_type" where "java_lang_Object_type ≡ Abs_javaDL_type
(UNIV::java_lang_Object set)"

lemma java_lang_Object_subset_any[simp]:"(UNIV::java_lang_Object set) ⊆ (UNIV::any
set)"
  by simp

lemma bottom_in_java_lang_Object[simp] :"bottom ∈ (UNIV::java_lang_Object set)"
  using java_lang_Object_UNIV_specification
  using type_definition.Rep_range type_definition_java_lang_Object by blast


consts
  Field_UNIV::"any set"

specification (Field_UNIV) "Field_UNIV ⊆ (UNIV::any set)"
  "Field_UNIV ≠ {}"
  by auto

lemma Field_UNIV_specification:"Field_UNIV ⊆ (UNIV::any set) ∧
  Field_UNIV ≠ {}"
  by (metis (mono_tags, lifting) Field_UNIV_def empty_not_UNIV someI_ex top_greatest)

typedef Field = Field_UNIV
  morphisms Field2any any2Field
  using Field_UNIV_specification by auto

declare [[coercion Field2any]]

consts
  created::"Field"
  fieldType::"Field⇒javaDL_type"

axiomatization arr::"int⇒Field" where arr_inject[simp]:"(arr x = arr y) = (x = y)"

definition Field_type::"javaDL_type" where "Field_type ≡ Abs_javaDL_type (UNIV::Field
set)"


typedef LocSet = "UNIV::(java_lang_Object × Field) set set"
  by simp


setup_lifting  type_definition_LocSet
```

lift_definition *elementOf::"java_lang_Object ⇒ Field ⇒LocSet⇒bool"* **is** *"λobj f s. (obj, f) ∈ s"*.
lift_definition *empty::"LocSet"* **is** *Set.empty*.
lift_definition *allLocs::"LocSet"* **is** *Set.UNIV*.
lift_definition *singleton::"java_lang_Object⇒Field⇒LocSet"* **is** *"λobj f. {(obj, f)}"*.
lift_definition *disjoint::"LocSet⇒LocSet⇒bool"* **is** *Set.disjnt*.
lift_definition *union::"LocSet⇒LocSet⇒LocSet"* **is** *Set.union*.
lift_definition *intersect::"LocSet⇒LocSet⇒LocSet"* **is** *Set.inter*.
lift_definition *setMinus::"LocSet⇒LocSet⇒LocSet"* **is** *minus*.
lift_definition *allFields::"java_lang_Object⇒LocSet"* **is** *"λx. {x} × (UNIV::Field set)"*.
lift_definition *allObjects::"Field⇒LocSet"* **is** *"λx. (UNIV::java_lang_Object set) × {x}"*.
lift_definition *arrayRange::"java_lang_Object⇒int⇒int⇒LocSet"* **is** *"λobj x y. {obj} × (image arr {x..y})"*.
lift_definition *subset::"LocSet⇒LocSet⇒bool"* **is** *Set.subset*.
lift_definition *infiniteUnion::"LocSet set⇒LocSet"* **is** *Complete_Lattices.Union*.

consts
  LocSet_Image::"any set"
  LocSet2any::"LocSet⇒any"
  any2LocSet::"any⇒LocSet"

axiomatization where *LocSet_sub_any:"type_definition LocSet2any any2LocSet LocSet_Image"*

declare *[[coercion LocSet2any]]*

interpretation *LocSet:type_definition LocSet2any any2LocSet LocSet_Image*
  by *(rule LocSet_sub_any)*

definition *LocSet_type::"javaDL_type"* **where** *"LocSet_type ≡ Abs_javaDL_type (UNIV::LocSet set)"*

typedef *Heap = "UNIV::(java_lang_Object ⇒ Field ⇒ any) set"*
  by *simp*

declare *[[coercion Rep_Heap]]*

consts
  Heap_Image::"any set"
  Heap2any::"Heap⇒any"
  any2Heap::"any⇒Heap"

axiomatization where *Heap_sub_any:"type_definition Heap2any any2Heap Heap_Image"*

declare *[[coercion Heap2any]]*

```
interpretation Heap:type_definition Heap2any any2Heap Heap_Image
  by (rule Heap_sub_any)


definition Heap_type::"javaDL_type" where "Heap_type ≡ Abs_javaDL_type (UNIV::Heap set)"


class any =
  fixes to_any::"'a⇒any"
  fixes cast::"any⇒'a"

instantiation any::any
begin
fun to_any_any where "to_any_any x = (id::any⇒any) x"
fun cast_any where "cast_any x = (id::any⇒any) x"
instance by standard
end

instantiation int::any
begin
fun to_any_int where "to_any_int x = int2any x"
fun cast_int where "cast_int x = any2int x"
instance by standard
end

instantiation bool::any
begin
fun to_any_bool where "to_any_bool x = bool2any x"
fun cast_bool where "cast_bool x = any2bool x"
instance by standard
end

instantiation Field::any
begin
fun to_any_Field where "to_any_Field x = Field2any x"
fun cast_Field where "cast_Field x = any2Field x"
instance by standard
end

instantiation LocSet::any
begin
fun to_any_LocSet where "to_any_LocSet x = LocSet2any x"
fun cast_LocSet where "cast_LocSet x = any2LocSet x"
instance by standard
end
```

```
instantiation Heap::any
begin
fun to_any_Heap where "to_any_Heap x = Heap2any x"
fun cast_Heap where "cast_Heap x = any2Heap x"
instance by standard
end

instantiation java_lang_Object::any
begin
fun cast_java_lang_Object where "cast_java_lang_Object x = any2java_lang_Object x"
fun to_any_java_lang_Object where "to_any_java_lang_Object x = java_lang_Object2any
x"
instance by standard
end

typedef (overloaded) Null = "{bottom}"
  morphisms Null2any any2Null
  by simp

declare [[coercion Null2any]]

lemma bottom_Null_set:"(UNIV::Null set) = {bottom}"
  using type_definition.Rep_range type_definition_Null by blast

lemma Null_sub_java_lang_Object_Types: "(UNIV::Null set) ⊆ (UNIV::java_lang_Object
set)"
  using bottom_Null_set bottom_in_java_lang_Object by auto

definition "null ≡ any2Null bottom"

instantiation Null::any
begin
fun to_any_Null where "to_any_Null (x::Null) = Null2any x"
fun cast_Null where "cast_Null x = any2Null x"
instance by standard
end

abbreviation "Null2java_lang_Object≡any2java_lang_Object ∘ Null2any"

declare [[coercion Null2java_lang_Object]]

fun instanceof::"any⇒javaDL_type⇒bool"
  where "instanceof x type = typeof x type"

typedef Seq = "UNIV::any list set"
  by auto
```

```
axiomatization Seq2any any2Seq Seq_UNIV
  where Seq_sub_any:"type_definition (Seq2any::Seq⇒any) (any2Seq::any⇒Seq) (Seq_UNIV::any
set)"

declare [[coercion Seq2any]]

interpretation Seq:type_definition Seq2any any2Seq Seq_UNIV
  by (rule Seq_sub_any)


instantiation Seq::any
begin
fun to_any_Seq where "to_any_Seq (x::Seq) = Seq2any x"
fun cast_Seq where "cast_Seq (x::any) = any2Seq x"
instance by standard
end

definition Seq_type::"javaDL_type" where "Seq_type ≡ Abs_javaDL_type (UNIV::Seq set)"

consts
  seqGetOutside::any

setup_lifting type_definition_Seq
lift_definition seqLen::"Seq⇒int" is "int ∘ List.length".
lift_definition seqGet::"Seq⇒int⇒'a::any" is "λs i. (if (0::int)≤i∧i<(int (length s))
then cast (s ! (nat i)) else cast seqGetOutside)".
lift_definition seqDef::"int⇒int⇒(int⇒any)⇒Seq" is "λle ri e. map e [le..ri - 1]".
lift_definition seqEmpty::"Seq" is "[]".
lift_definition seqSingleton::"any⇒Seq" is "λx. [x]".
lift_definition seqConcat::"Seq⇒Seq⇒Seq" is List.append.
lift_definition seqReverse::"Seq⇒Seq" is List.rev.
lift_definition seqPerm::"Seq⇒Seq⇒bool" is List_Permutation.perm.

fun seqNPerm::"Seq⇒bool"
  where "seqNPerm s = seqPerm s (seqDef 0 (seqLen s - 1) (to_any))"

fun seqSub::"Seq⇒int⇒int⇒Seq" where
  "seqSub s i j = seqDef i j (λx. seqGet s x)"

primrec (nonexhaustive) listIndexOf::"'a list⇒'a⇒int" where
  "listIndexOf (x#xs) a = (if (x=a) then 0 else 1+(listIndexOf xs a))"

lift_definition seqIndexOf::"Seq⇒any⇒int" is "listIndexOf".

fun listSwap::"'a list⇒int⇒int⇒'a list"
  where "listSwap l i j =
(if ¬(0≤i ∧ i<int (length l) ∧ 0≤j ∧ i<int (length l))
```

```
then l
else list_update (list_update l (nat i) (l ! (nat j))) (nat j) (l ! (nat i)))"
```

lift_definition *seqSwap::"Seq⇒int⇒int⇒Seq"* **is** *listSwap*.

fun *listRemove::"'a list⇒nat⇒'a list"*
  where *"listRemove [] _ = []"*
  | "listRemove (x#xs) 0 = xs"
  | "listRemove (x#xs) (Suc k) = x # (listRemove xs k)"

lift_definition *seqRemove::"Seq⇒int⇒Seq"* **is** *"λs (i::int). (if ¬(0≤i ∧ i<int (length s)) then s else listRemove s (nat i))"*.

consts
  exactInstance::"any⇒javaDL_type⇒bool"

axiomatization *obj_length::"java_lang_Object⇒int"* **where** *length_nonneg[simp]:"obj_length obj ≥ 0"*

fun *unusedLocs* **where** *"unusedLocs (h::Heap) = Abs_LocSet {((obj::java_lang_Object), (f::Field)). (h obj created=False)∧ obj≠null}"*

fun *select::"Heap⇒java_lang_Object⇒Field⇒'a::any"* **where**
  "select h obj f = cast (h obj f)"

fun *anon::"Heap⇒LocSet⇒Heap⇒Heap"* **where**
  "anon h1 s h2 = Abs_Heap (λ(obj::java_lang_Object) (f::Field). (if elementOf obj f s ∧ f≠created ∨ elementOf obj f (unusedLocs h1)
    then select h2 obj f else select h1 obj f))"

fun *store::"Heap⇒java_lang_Object⇒Field⇒any⇒Heap"* **where**
  "store h obj f x = Abs_Heap (λ(obj'::java_lang_Object) (f'::Field). (if obj'=obj ∧ f'=f ∧ f≠created then x else h obj' f'))"

fun *create::"Heap⇒java_lang_Object⇒Heap"* **where**
  "create h obj = Abs_Heap (λ(obj'::java_lang_Object) (f'::Field). (if obj'=obj ∧ f'=created ∧ obj≠null then cast True else h obj' f'))"

class *array = any +*
  fixes *element_type::"'a⇒javaDL_type"*

axiomatization *wellFormed::"Heap⇒bool"* **where**
  onlyCreatedjava_lang_ObjecteAreReferenced:"wellFormed h ⟹ select h obj f = null ∨
    ((select h (select h obj f) created)::bool)"

```
  and onlyCreatedjava_lang_ObjectsAreInLocSets:"wellFormed h ∧ elementOf (o2::java_lang_Object)
f2 ((select
      h obj f)::LocSet) ⟹ Null2java_lang_Object null=o2 ∨ ((select h o2
      created)::bool)"
  and wellFormedStorejava_lang_Object:"wellFormed h ∧ ((x::java_lang_Object)=null ∨
((select
      h x created) ∧ instanceof x (fieldType f))) ⟹ wellFormed (store h obj f x)"
  and wellFormedStoreLocSet:"wellFormed h ∧ (∀ov fv. (elementOf ov fv y ⟶ ov = null
∨ select h ov created))
      ⟹ wellFormed (store h obj f y)"
  and wellFormedStorePrimitive:"(typeof x (fieldType f) ⟹ ¬typeof x java_lang_Object_type
⟹ ¬typeof x LocSet_type  ⟹  wellFormed h
    ⟹ wellFormed (store h obj f x))"
  and wellFormedCreate:"wellFormed h ⟹ wellFormed (create h obj)"
  and wellFormedAnon:"wellFormed h ∧ wellFormed h2 ⟹ wellFormed (anon h y h2)"

axiomatization where wellFormedStoreArray:"wellFormed h ∧ ((x::java_lang_Object)=null
∨ (select h x created ∧ (typeof x (element_type obj))))
      ⟹ wellFormed (store h (cast (to_any (obj::'a::{array, any}))) (arr idx) x)"

definition "setOfStandardAnySubtypes≡{int_type, bool_type, java_lang_Object_type, Field_type,
Heap_type, LocSet_type, Seq_type}"


axiomatization where distinctStandardTypes[simp]:"∀x∈setOfStandardAnySubtypes. (∀y∈setOfStandarc
disjointTypes x y)"

lemma induct_sum_upper_limit:
  fixes f::"int⇒int"
  fixes lower::int
  fixes upper::int
  assumes "lower<upper"
  shows "(∑(i::int) = lower..<upper. f i) = (∑(i::int) = lower..<upper - 1. f i) +
f (upper - 1)"
proof -
  have "{lower..<upper} = {lower..<upper-1} ∪ {upper-1..<upper}"
    using assms by auto
  have "{upper-1..<upper} = {upper - 1}"
    by auto
  then have "sum f ({lower..<upper-1} ∪ {upper-1..<upper}) = (∑(i::int) = lower..<upper-1.
f i) + (∑(i::int) = upper-1..<upper. f i) - sum f ({lower..<upper-1} ∩ {upper-1..<upper})"
    by (subst sum.union_inter [symmetric]) (auto simp add: algebra_simps)
  then have "sum f {lower..<upper} = (∑(i::int) = lower..<upper-1. f i) + (∑(i::int)
= upper-1..<upper. f i) - sum f ({lower..<upper-1} ∩ {upper-1..<upper})"
    using ‹{lower..<upper} = {lower..<upper-1} ∪ {upper-1..<upper}› by presburger
  also have "... = (∑(i::int) = lower..<upper-1. f i) + (∑(i::int) = upper-1..<upper.
f i)"
```

```
    by simp
  finally show ?thesis
    using ‹{upper-1..<upper} = {upper - 1}› by auto
qed

end
```