Jive Data and Store Model

Norbert Schirmer
TU München
schirmer@informatik.tu-muenchen.de

Nicole Rauch
TU Kaiserslautern
rauch@informatik.uni-kl.de

Abstract

This document presents the formalization of an object-oriented data and store model in ISABELLE/HOL. This model is being used in the Java Interactive Verification Environment, JIVE.
## Contents

1 Introduction 5

2 Theory Dependencies 7

3 The Example Program 8

4 TypeIds 9

5 Java-Type 9

6 The Direct Subtype Relation of Java Types 11

7 Widening the Direct Subtype Relation 13  
   7.1 Auxiliary lemmas .................................................. 13  
   7.2 The Widening (Subtype) Relation of Javatypes .................. 15  
   7.3 The Subtype Relation as Partial Order .......................... 15  
   7.4 Javatype Ordering Properties .................................... 16  
   7.5 Enhancing the Simplifier ........................................... 17  
   7.6 Properties of the Subtype Relation .............................. 17

8 Attributes 19

9 Program-Independent Lemmas on Attributes 22

10 Value 23  
   10.1 Discriminator Functions .......................................... 24  
   10.2 Selector Functions ................................................ 27  
   10.3 Determining the Type of a Value ................................ 29  
   10.4 Default Initialization Values for Types ......................... 30

11 Location 32

12 Store 34  
   12.1 New ........................................................................ 34  
   12.2 The Definition of the Store ........................................ 35  
   12.3 The Store Interface .................................................. 36  
   12.4 Derived Properties of the Store ................................... 37

13 Store Properties 51  
   13.1 Reachability of a Location from a Reference .................... 51  
   13.2 Reachability of a Reference from a Reference .................. 59  
   13.3 Disjointness of Reachable Locations .............................. 59  
   13.4 X-Equivalence .......................................................... 61  
   13.5 T-Equivalence .......................................................... 64  
   13.6 Less Alive ............................................................... 64  
   13.7 Reachability of Types from Types ................................. 67

14 The Formalization of JML Operators 70
15 The Universal Specification
1 Introduction

*

**Jive** [MPH00, Jiv] is a verification system that is being developed at the University of Kaiserslautern and at the ETH Zürich. It is an interactive special-purpose theorem prover for the verification of object-oriented programs on the basis of a partial-correctness Hoare-style programming logic. **Jive** operates on Java-KE [PHGR05], a desugared subset of sequential Java which contains all important features of object-oriented languages (subtyping, exceptions, static and dynamic method invocation, etc.). **Jive** is written in Java and currently has a size of about 40,000 lines of code.

**Jive** is able to operate on completely unannotated programs, allowing the user to dynamically add specifications. It is also possible to preliminarily annotate programs with invariants, pre- and postconditions using the specification language JML [LBR99]. In practice, a mixture of both techniques is employed, in which the user extends and refines the pre-annotated specifications during the verification process. The program to be verified, together with the specifications, is translated to Hoare sequents. Program and pre-annotated specifications are translated during startup, while the dynamically added specifications are translated whenever they are entered by the user. Hoare sequents have the shape $A \triangleright \{ P \} \text{pp} \{ Q \}$ and express that for all states $S$ that fulfill $P$, if the execution of the program part $\text{pp}$ terminates, the state that is reached when $\text{pp}$ has been evaluated in $S$ must fulfill $Q$. The so-called assumptions $A$ are used to prove recursive methods.

**Jive**’s logic contains so-called Hoare rules and axioms. The rules consist of one or more Hoare sequents that represent the assumptions of the rule, and a Hoare sequent which is the conclusion of the rule. Axioms consist of only one Hoare sequent; they do not have assumptions. Therefore, axioms represent the known facts of the Hoare logic.

To prove a program specification, the user directly works on the program source code. Proofs can be performed in backward direction and in forward direction. In backward direction, an initial open proof goal is reduced to new, smaller open subgoals by applying a rule. This process is repeated for the smaller subgoals until eventually each open subgoal can be closed by the application of an axiom. If all open subgoals are proven by axioms, the initial goal is proven as well.

In forward direction, the axioms can be used to establish known facts about the statements of a given program. The rules are then used to produce new facts from these already known facts. This way, facts can be constructed for parts of the program.

A large number of the rules and axioms of the Hoare logic is related to the structure of the program part that is currently being examined. Besides these, the logic also contains rules that manipulate the pre- or postcondition of the examined subgoal without affecting the current program part selection. A prominent member of this kind of rules is the rule of consequence:\footnote{In **Jive**, the rule of consequence is part of a larger rule which serves several purposes at once. Since we want to focus on the rule of consequence, we left out the parts that are irrelevant in this context.}

\[
\begin{align*}
\text{PP} \Rightarrow P & \quad A \triangleright \{ P \} \text{pp} \{ Q \} & \quad Q \Rightarrow QQ \\
\hline
A \triangleright \{ PP \} \text{pp} \{ QQ \}
\end{align*}
\]

It plays a special role in the Hoare logic because it additionally requires implications between stronger and weaker conditions to be proven. If a **Jive** proof contains an application of the rule of consequence, the implication is attached to the proof tree node that documents this rule application; these attachments are called lemmas. **Jive** sends these lemmas to an associated...
general purpose theorem prover where the user is required to prove them. Currently, JIVE supports ISABELLE/HOL as associated prover. It is required that all lemmas that are attached to any node of a proof tree are proven before the initial goal of the proof tree is accepted as being proven.

In order to prove these logical predicates, ISABELLE/HOL needs a data and store model of JAVA-KE. This model acts as an interface between JIVE and ISABELLE/HOL.

The first paper-and-pencil formalization of the data and store model was given in Arnd Poetzsch-Heffter’s habilitation thesis [PH97, Sect. 3.1.2]. The first machine-supported formalization was performed in PVS by Peter Müller, by translating the axioms given in [PH97] to axioms in PVS. The formalization presented in this report extends the PVS formalization. The axioms have been replaced by conservative extensions and proven lemmas, thus there is no longer any possibility to accidentally introduce unsoundness.

Some changes were made to the PVS theories during the conversion. Some were caused due to the differences in the tools ISABELLE/HOL and PVS, but some are more conceptional. Here is a list of the major changes.

- In PVS, function arguments were sometimes restricted to subtypes. In ISABELLE/HOL, unintended usage of functions is left unspecified.

- In PVS, the program-independent theories were parameterized by the datatypes that were generated for the program to be verified. In ISABELLE/HOL, we just build on the generated theories. This makes the whole setting easier. The drawback is that we have to run the theories for each program we want to verify. But the proof scripts are designed in a way that they will work if the basic program-dependent theories are generated in the proper way. Since we can create an image of a proof session before starting actual verification we do not run into time problems either.

- The subtype relation is based on the direct subtype relation between classes and interfaces. We prove that subtyping forms a partial order. In the PVS version subtyping was expressed by axioms that described the subtype relation for the types appearing in the Java program to be verified.

Besides these changes we also added new concepts to the model. We can now deal with static fields and arrays. This way, the model supports programming languages that are much richer than JAVA-KE to allow for future extensions of JIVE.

Please note that although the typographic conventions in Isabelle suggest that constructors start with a capital letter while types do not, we kept the capitalization as it was before (which means that types start with a capital letter while constructors usually do not) to keep the naming more uniform across the various JIVE-related publications.

The theories presented in this report require the use of ISABELLE 2005. The proofs of lemmas are skipped in the presentation to keep it compact. The full proofs can be found in the original ISABELLE theories.
2 Theory Dependencies

The theories “TypeIds”, “DirectSubtypes”, “Attributes” and “UnivSpec” are program-dependent and are generated by the Jive tool. The program-dependent theories presented in this report are just examples and act as placeholders. The theories are stored in four different directories:

Isabelle:
- JavaType.thy
- Subtype.thy
- Value.thy
- JML.thy

Isabelle_Store:
- AttributesIndep.thy
- Location.thy
- Store.thy
- StoreProperties.thy

Isa_(Prog):
- TypeIds.thy
- DirectSubtypes.thy
- UnivSpec.thy

Isa_(Prog)_Store:
- Attributes.thy
In this naming convention, the suffix “_Store” denotes those theories that depend on the actual realization of the Store. They have been separated in order to allow for easy exchanging of the Store realization. The midfix “⟨Prog⟩” denotes the name of the program for which the program-dependent theories have been generated. This way, different program-dependent theories can reside side-by-side without conflicts.

These four directories have to be added to the ML path before loading UnivSpec. This can be done in a setup theory with the following command (here applied to a program called Counter):

```ml
ML {*
  add_path "<PATH_TO_THEORIES>/Isabelle";
  add_path "<PATH_TO_THEORIES>/Isabelle_Store";
  add_path "<PATH_TO_THEORIES>/Isa.Counter";
  add_path "<PATH_TO_THEORIES>/Isa.Counter.Store";
  *}
```

This way, one can select the program-dependent theories for the program that currently is to be proven.

### 3 The Example Program

The program-dependent theories are generated for the following example program:

```java
interface Counter {
    public int incr();
    public int reset();
}

class CounterImpl implements Counter {
    protected int value;

    public int incr()
    {
        int dummy;
        res = this.value;
        res = (int) res + 1;
        this.value = res;
    }

    public int reset()
    {
        int dummy;
        this.value=0;
        res = (int) 0;
    }
}

class UndoCounter extends CounterImpl {
    private int save;
```
public int incr()
{
    int dummy;
    res = this.value;
    this.save = res;
    res = res + 1;
    this.value = res;
}

public int un_do()
{
    int res2;
    res = this.save;
    res2 = this.value;
    this.value = res;
    this.save = res2;
}

4 TypeIds

theory TypeIds imports Main begin

This theory contains the program specific names of abstract and concrete classes and interfaces. It has to be generated for each program we want to verify. The following classes are an example taken from the program given in Sect. 3. They are complemented by the classes that are known to exist in each Java program implicitly, namely Object, Exception, ClassCastException and NullPointerException. The example program does not contain any abstract classes, but since we cannot formalize datatypes without constructors, we have to insert a dummy class which we call Dummy.

The datatype CTypeId must contain a constructor called Object because subsequent proofs in the Subtype theory rely on it.

datatype CTypeId = CounterImpl | UndoCounter
             | Object | Exception | ClassCastException | NullPointerExce
— The last line contains the classes that exist in every program by default.

datatype ITypeId = Counter

datatype CTypeId = Dummy
— we cannot have an empty type.

Why do we need different datatypes for the different type identifiers? Because we want to be able to distinguish the different identifier kinds. This has a practical reason: If we formalize objects as "ObjectId × TypeId" and if we quantify over all objects, we get a lot of objects that do not exist, namely all objects that bear an interface type identifier or abstract class identifier. This is not very helpful. Therefore, we separate the three identifier kinds from each other.

end

5 Java-Type

theory JavaType imports ../Isa.Counter/TypeIds begin
This theory formalizes the types that appear in a Java program. Note that the types defined by
the classes and interfaces are formalized via their identifiers. This way, this theory is program-

independent.

We only want to formalize one-dimensional arrays. Therefore, we describe the types that can
be used as element types of arrays. This excludes the null type and array types themselves.

This way, we get a finite number of types in our type hierarchy, and the subtype relations can
be given explicitly (see Sec. 6). If desired, this can be extended in the future by using Javatype
as argument type of the ArrT type constructor. This will yield infinitely many types.

\[-\]
\[
\text{datatype } \text{Arraytype} = \text{BoolAT} \mid \text{IntgAT} \mid \text{ShortAT} \mid \text{ByteAT} \\
| \text{CClassAT CTypeId} \mid \text{AClassAT ATypeId} \\
| \text{InterfaceAT ITypeId}
\]

\[
\text{datatype } \text{Javatype} = \text{BoolT} \mid \text{IntgT} \mid \text{ShortT} \mid \text{ByteT} \mid \text{NullT} \mid \text{ArrT Arraytype} \\
| \text{CClassT CTypeId} \mid \text{AClassT ATypeId} \\
| \text{InterfaceT ITypeId}
\]

We need a function that widens Arraytype to Javatype.

**definition**

\[
\text{at2jt :: Arraytype } \Rightarrow \text{ Javatype}
\]

**where**

\[
\text{at2jt at } = \begin{cases} 
\text{BoolAT } & \Rightarrow \text{BoolT} \\
\text{IntgAT } & \Rightarrow \text{IntgT} \\
\text{ShortAT } & \Rightarrow \text{ShortT} \\
\text{ByteAT } & \Rightarrow \text{ByteT} \\
\text{CClassAT CTypeId } & \Rightarrow \text{CClassT CTypeId} \\
\text{AClassAT ATypeId } & \Rightarrow \text{AClassT ATypeId} \\
\text{InterfaceAT ITypeId } & \Rightarrow \text{InterfaceT ITypeId}
\end{cases}
\]

We define two predicates that separate the primitive types and the class types.

**primrec** isprimitive:: Javatype \(\Rightarrow\) bool

**where**

\[
\text{isprimitive BoolT } = \text{True} \\
\text{isprimitive IntgT } = \text{True} \\
\text{isprimitive ShortT } = \text{True} \\
\text{isprimitive ByteT } = \text{True} \\
\text{isprimitive NullT } = \text{False} \\
\text{isprimitive (ArrT T) } = \text{False} \\
\text{isprimitive (CClassT c) } = \text{False} \\
\text{isprimitive (AClassT c) } = \text{False} \\
\text{isprimitive (InterfaceT i) } = \text{False}
\]

**primrec** isclass:: Javatype \(\Rightarrow\) bool

**where**

\[
\text{isclass BoolT } = \text{False} \\
\text{isclass IntgT } = \text{False} \\
\text{isclass ShortT } = \text{False} \\
\text{isclass ByteT } = \text{False} \\
\text{isclass NullT } = \text{False} \\
\text{isclass (ArrT T) } = \text{False} \\
\text{isclass (CClassT c) } = \text{True} \\
\text{isclass (AClassT c) } = \text{True}
\]
isclass (InterfaceT i) = False

end

6 The Direct Subtype Relation of Java Types

theory DirectSubtypes
  imports ../Isabelle/JavaType
begin

In this theory, we formalize the direct subtype relations of the Java types (as defined in Sec. 4) that appear in the program to be verified. Thus, this theory has to be generated for each program.

We have the following type hierarchy:

We need to describe all direct subtype relations of this type hierarchy. As you can see in the picture, all unnecessary direct subtype relations can be ignored, e.g. the subclass relation between CounterImpl and Object, because it is added transitively by the widening relation of types (see Sec. 7.2).

We have to specify the direct subtype relation between

- each “leaf” class or interface and its subtype Nullable
- each “root” class or interface and its supertype Object
- each two types that are direct subtypes as specified in the code by extends or implements
- each array type of a primitive type and its subtype Nullable
- each array type of a primitive type and its supertype Object
- each array type of a “leaf” class or interface and its subtype Nullable
- the array type Object[] and its supertype Object
- two array types if their element types are in a subtype hierarchy

**definition** `direct-subtype :: (Javatype * Javatype) set where**

direct-subtype =

{(NullT, AClassT Dummy),
 (NullT, CClassT UndoCounter),
 (NullT, CClassT NullPointerException),
 (NullT, CClassT ClassCastException),

(AClassT Dummy, CClassT Object),
 (InterfaceT Counter, CClassT Object),
 (CClassT Exception, CClassT Object),

(CClassT UndoCounter, CClassT CounterImpl),
 (CClassT CounterImpl, InterfaceT Counter),
 (CClassT NullPointerException, CClassT Exception),
 (CClassT ClassCastException, CClassT Exception),

(NullT, ArrT BoolAT),
 (NullT, ArrT IntgAT),
 (NullT, ArrT ShortAT),
 (NullT, ArrT ByteAT),
 (ArrT BoolAT, CClassT Object),
 (ArrT IntgAT, CClassT Object),
 (ArrT ShortAT, CClassT Object),
 (ArrT ByteAT, CClassT Object),

(NullT, ArrT (AClassAT Dummy)),
 (NullT, ArrT (CClassAT UndoCounter)),
 (NullT, ArrT (CClassAT NullPointerException)),
 (NullT, ArrT (CClassAT ClassCastException)),

(ArrT (CClassAT Object), CClassT Object),

(ArrT (AClassAT Dummy), ArrT (CClassAT Object)),
 (ArrT (CClassAT CounterImpl), ArrT (InterfaceAT Counter)),
 (ArrT (InterfaceAT Counter), ArrT (CClassAT Object)),
 (ArrT (CClassAT Exception), ArrT (CClassAT Object)),
 (ArrT (CClassAT UndoCounter), ArrT (CClassAT CounterImpl)),
 (ArrT (CClassAT NullPointerException), ArrT (CClassAT Exception)),
 (ArrT (CClassAT ClassCastException), ArrT (CClassAT Exception))
}

This lemma is used later in the Simplifier.

**lemma** `direct-subtype`:

(NullT, AClassT Dummy) ∈ direct-subtype
 (NullT, CClassT UndoCounter) ∈ direct-subtype
 (NullT, CClassT NullPointerException) ∈ direct-subtype
 (NullT, CClassT ClassCastException) ∈ direct-subtype

(AClassT Dummy, CClassT Object) ∈ direct-subtype
 (InterfaceT Counter, CClassT Object) ∈ direct-subtype
 (CClassT Exception, CClassT Object) ∈ direct-subtype
(\text{CClassT\ }\text{UndoCounter}, \ \text{CClassT\ }\text{CounterImpl}) \in \text{direct-subtype}
\(\text{CClassT\ }\text{CounterImpl}, \ \text{InterfaceT\ }\text{Counter}) \in \text{direct-subtype}
(\text{CClassT\ }\text{NullPointerException}, \ \text{CClassT\ }\text{Exception}) \in \text{direct-subtype}
(\text{CClassT\ }\text{ClassCastException}, \ \text{CClassT\ }\text{Exception}) \in \text{direct-subtype}

(\text{NullT, ArrT\ }\text{BoolAT}) \in \text{direct-subtype}
(\text{NullT, ArrT\ }\text{IntgAT}) \in \text{direct-subtype}
(\text{NullT, ArrT\ }\text{ShortAT}) \in \text{direct-subtype}
(\text{NullT, ArrT\ }\text{ByteAT}) \in \text{direct-subtype}
(\text{ArrT\ }\text{BoolAT,\ }\text{CClassT\ }\text{Object}) \in \text{direct-subtype}
(\text{ArrT\ }\text{IntgAT,\ }\text{CClassT\ }\text{Object}) \in \text{direct-subtype}
(\text{ArrT\ }\text{ShortAT,\ }\text{CClassT\ }\text{Object}) \in \text{direct-subtype}
(\text{ArrT\ }\text{ByteAT,\ }\text{CClassT\ }\text{Object}) \in \text{direct-subtype}

(\text{NullT, ArrT\ }\text{AClassAT\ Dummy}) \in \text{direct-subtype}
(\text{NullT, ArrT\ }\text{CClassAT\ UndoCounter}) \in \text{direct-subtype}
(\text{NullT, ArrT\ }\text{CClassAT\ NullPointerException}) \in \text{direct-subtype}
(\text{NullT, ArrT\ }\text{CClassAT\ ClassCastException}) \in \text{direct-subtype}

(\text{ArrT\ }\text{CClassAT\ Object),\ }\text{CClassT\ }\text{Object}) \in \text{direct-subtype}
(\text{ArrT\ }\text{AClassAT\ Dummy),\ }\text{ArrT\ }\text{CClassAT\ Object}) \in \text{direct-subtype}
(\text{ArrT\ }\text{CClassAT\ CounterImpl),\ }\text{ArrT\ }\text{InterfaceAT\ Counter}) \in \text{direct-subtype}
(\text{ArrT\ }\text{InterfaceAT\ Counter),\ }\text{ArrT\ }\text{CClassAT\ Object}) \in \text{direct-subtype}
(\text{ArrT\ }\text{CClassAT\ Exception),\ }\text{ArrT\ }\text{CClassAT\ Object}) \in \text{direct-subtype}
(\text{ArrT\ }\text{CClassAT\ UndoCounter),\ }\text{ArrT\ }\text{CClassAT\ CounterImpl}) \in \text{direct-subtype}
(\text{ArrT\ }\text{CClassAT\ NullPointerException),\ }\text{ArrT\ }\text{CClassAT\ Exception}) \in \text{direct-subtype}
(\text{ArrT\ }\text{CClassAT\ ClassCastException),\ }\text{ArrT\ }\text{CClassAT\ Exception}) \in \text{direct-subtype}

\text{by (simp-all add: direct-subtype-def)}

end

7 Widening the Direct Subtype Relation

theory Subtype
imports ../Isa-Counter/DirectSubtypes
begin

In this theory, we define the widening subtype relation of types and prove that it is a partial order.

7.1 Auxiliary lemmas

These general lemmas are not especially related to Jive. They capture some useful properties of general relations.

\text{lemma distinct-rtrancl-into-trancl:}
\text{assumes neq-x-y: } x \neq y
\text{assumes x-y-rtrancl: } (x, y) \in r^*
\text{shows } (x, y) \in r^+
\text{using x-y-rtrancl neq-x-y}
\text{proof (induct)}
\text{assume } x \neq x \text{ thus } (x, x) \in r^+ \text{ by simp}
next
fix y z
assume \( x\cdot y\cdot \text{rtrancl} \): \((x, y) \in r^*\)
assume \( y\cdot z\cdot r \): \((y, z) \in r\)
assume \( x \neq y \implies (x, y) \in r^+\)
assume \( x \neq z \)
assume \( x\cdot y\cdot \text{rtrancl} \):
show \((x, z) \in r^+\)
proof (cases)
assume \( x = y \)
with \( y\cdot z\cdot r \) have \((x, z) \in r\) by simp
thus \((x, z) \in r^+\).
next
fix w
assume \((x, w) \in r^*\)
moreover assume \((w, y) \in r\)
ultimately have \((x, y) \in r^+\)
by (rule \text{rtrancl-into-trancl1})
from this \( y\cdot z\cdot r \)
show \((x, z) \in r^+\).
qed
qed

lemma \text{acyclic-imp-antisym-rtrancl}: \text{acyclic } r \implies \text{antisym } (r^*)
proof (clar simp simp only: \text{acyclic-def antisym-def})
fix x y
assume \text{acyclic}: \( \forall x. (x, x) \notin r^+\)
assume \( x\cdot y\cdot r \): \((x, y) \in r^*\)
assume \( y\cdot x\cdot r \): \((y, x) \in r^*\)
show \( x = y \)
proof (cases \( x = y \))
 case \( \text{True} \) thus \(?\text{thesis}\) .
next
 case \( \text{False} \)
from \( \text{False} \) \( x\cdot y \)
show \((x, y) \in r^+\)
by (rule \text{distinct-rtrancl-into-trancl})
also
from \( \text{False} \) \( y\cdot x \)
show \((y, x) \in r^+\)
by (fastforce intro: \text{distinct-rtrancl-into-trancl})
finally have \((x, x) \in r^+\).
with \text{acyclic} show \(?\text{thesis}\) by simp
qed
qed

lemma \text{acyclic-rtrancl-rtrancl}:
assumes \text{acyclic}: \text{acyclic } r
shows \((x, y) \in r^+ = ((x, y) \in r^* \land x \neq y)\)
proof
assume \( x\cdot y\cdot \text{rtrancl} \): \((x, y) \in r^*\)
show \((x, y) \in r^* \land x \neq y\)
proof
from \( x\cdot y\cdot \text{rtrancl} \)
show \((x, y) \in r^+\).
next
from \( x\cdot y\cdot \text{rtrancl} \) \text{acyclic} show \( x \neq y \) by (auto simp add: \text{acyclic-def})
7.2 The Widening (Subtype) Relation of Javatypes

In this section we widen the direct subtype relations specified in Sec. 6. It is done by a calculation of the transitive closure of the direct subtype relation.

This is the concrete syntax that expresses the subtype relations between all types.

abbreviation
direct-subtype-syntax :: Javatype \Rightarrow Javatype \Rightarrow bool (- ≺ - [71, 71] 70)
where — direct subtype relation
A ≺ B == (A, B) ∈ direct-subtype

abbreviation
widen-syntax :: Javatype \Rightarrow Javatype \Rightarrow bool (- ⪯ - [71, 71] 70)
where — reflexive transitive closure of direct subtype relation
A ⪯ B == (A, B) ∈ direct-subtype*

abbreviation
widen-strict-syntax :: Javatype \Rightarrow Javatype \Rightarrow bool (- ≺ - [71, 71] 70)
where — transitive closure of direct subtype relation
A ≺ B == (A, B) ∈ direct-subtype+

7.3 The Subtype Relation as Partial Order

We prove the axioms required for partial orders, i.e. reflexivity, transitivity and antisymmetry, for the widened subtype relation. The direct subtype relation has been defined in Sec. 6. The reflexivity lemma is added to the Simplifier and to the Classical reasoner (via the attribute iff), and the transitivity and antisymmetry lemmas are made known as transitivity rules (via the attribute trans). This way, these lemmas will be automatically used in subsequent proofs.

lemma acyclic-direct-subtype: acyclic direct-subtype
proof (clarsimp simp add: acyclic-def)
  fix x show x ≺ x ⇒ False
  by (cases x) (fastforce elim: tranclE simp add: direct-subtype-def)+
qed

lemma antisym-rtrancl-direct-subtype: antisym (direct-subtype*)
using acyclic-direct-subtype by (rule acyclic-imp-antisym-rtrancl)

lemma widen-strict-to-widen: C ≺ D = (C ⪯ D ∧ C≠D)
using acyclic-direct-subtype by (rule acyclic-trancl-rtrancl)

The widening relation on Javatype is reflexive.

lemma widen-refl [iff]: X ⪯ X ..
The widening relation on Javatype is transitive.
lemma widen-trans [trans]:
assumes a-b: \( a \preceq b \)
shows \( \forall c. \ b \preceq c \Longrightarrow a \preceq c \)
by (insert a-b, rule rtrancl-trans)

The widening relation on Javatype is antisymmetric.

lemma widen-antisym [trans]:
assumes a-b: \( a \preceq b \)
assumes b-c: \( b \preceq a \)
shows \( a = b \)
using a-b b-c antisym-rtrancl-direct-subtype
by (unfold antisym-def) blast

7.4 Javatype Ordering Properties

The type class \textit{ord} allows us to overwrite the two comparison operators < and \( \leq \). These are the two comparison operators on \textit{Javatype} that we want to use subsequently.

We can also prove that \textit{Javatype} is in the type class \textit{order}. For this we have to prove reflexivity, transitivity, antisymmetry and that \( < \) and \( \leq \) are defined in such a way that \( (x < y) = (x \leq y \land x \neq y) \) holds. This proof can easily be achieved by using the lemmas proved above and the definition of less-Javatype-def.

instantiation Javatype:: order

begin

definition le-Javatype-def: \( A \leq B \equiv A \preceq B \)

definition less-Javatype-def: \( A < B \equiv A \leq B \land \neg B \leq (A::Javatype) \)

instance proof

fix \( x \ y \ z :: Javatype \)
{
  show \( x \leq x \)
    by (simp add: le-Javatype-def )

  next
  assume \( x \leq y \ y \leq z \)
  then show \( x \leq z \)
    by (unfold le-Javatype-def) (rule rtrancl-trans)

  next
  assume \( x \leq y \ y \leq x \)
  then show \( x = y \)
    apply (unfold le-Javatype-def)
    apply (rule widen-antisym)
    apply assumption +
    done

  next
  show \( (x < y) = (x \leq y \land \neg y \leq x) \)
    by (simp add: less-Javatype-def)

} qed
7.5 Enhancing the Simplifier

lemmas subtype-defs = le-Javatype-def less-Javatype-def
direct-subtype-def

lemmas subtype-ok-simps = subtype-defs
lemmas subtype-wrong-elims = rtranclE

During verification we will often have to solve the goal that one type widens to the other. So we equip the simplifier with a special solver-tactic.

lemma widen-asm: (a::Javatype) ≤ b ⇒ a ≤ b
  by simp

lemmas direct-subtype-widened = direct-subtype[THEN r-into-rtrancl]

ML ⟨⟨
  local val ss = simpset-of ⟨theory-context Transitive-Closure⟩ in
  fun widen-tac ctxt =
    rtac ⟨theory-context Transitive-Closure⟩ THEN'
    simp-tac (put-simpset ss ctxt addsimps ⟨theory-context Transitive-Closure⟩) THEN'
    Method.insert-tac ⟨theory-context Transitive-Closure⟩ THEN'
    simp-tac (put-simpset (simpset-of ⟨theory-context Transitive-Closure⟩) ctxt)
end ⟩⟩

declaration ⟨⟨ fn - =>
  Simplifier.map-ss (fn ss => ss addSolver (mk-solver widen widen-tac)) ⟩⟩

In this solver-tactic, we first try the trivial resolution with widen-asm to check if the actual subgoal really is a request to solve a subtyping problem. If so, we unfold the comparison operator, insert the direct subtype relations and call the simplifier.

7.6 Properties of the Subtype Relation

The class Object has to be the root of the class hierarchy, i.e. it is supertype of each concrete class, abstract class, interface and array type. The proof scripts should run on every correctly generated type hierarchy.

lemma Object-root: CClassT C ≤ CClassT Object
  by (cases C, simp-all)

lemma Object-root-abs: AClassT C ≤ CClassT Object
  by (cases C, simp-all)

lemma Object-root-int: InterfaceT C ≤ CClassT Object
  by (cases C, simp-all)

lemma Object-root-array: ArrT C ≤ CClassT Object
proof (cases C)
  fix x
  assume c: C = CClassAT x
  show ArrT C ≤ CClassT Object
    using c by (cases x, simp-all)
next
  fix x
  assume c: C = AClassAT x
  show ArrT C ≤ CClassT Object
    using c by (cases x, simp-all)
next
  fix x
  assume c: C = InterfaceAT x
  show ArrT C ≤ CClassT Object
    using c by (cases x, simp-all)
next
  assume c: C = BoolAT
  show ArrT C ≤ CClassT Object
    using c by simp
next
  assume c: C = IntgAT
  show ArrT C ≤ CClassT Object
    using c by simp
next
  assume c: C = ShortAT
  show ArrT C ≤ CClassT Object
    using c by simp
next
  assume c: C = ByteAT
  show ArrT C ≤ CClassT Object
    using c by simp
qed

If another type is (non-strict) supertype of Object, then it must be the type Object itself.

lemma Object-rootD:
  assumes p: CClassT Object ≤ c
  shows CClassT Object = c
  using p
  apply (cases c)
  apply (fastforce elim: subtype-wrong-elims simp add: subtype-defs) +
  — In this lemma, we only get contradictory cases except for Object itself.
done

The type NullT has to be the leaf of each branch of the class hierarchy, i.e. it is subtype of each type.

lemma NullT-leaf [simp]: NullT ≤ CClassT C
  by (cases C, simp-all)

lemma NullT-leaf-abs [simp]: NullT ≤ AClassT C
  by (cases C, simp-all)

lemma NullT-leaf-int [simp]: NullT ≤ InterfaceT C
  by (cases C, simp-all)
lemma NullT-leaf-array: NullT ≤ ArrT C
proof (cases C)
  fix x
  assume c: C = CClassAT x
  show NullT ≤ ArrT C
  using c by (cases x, simp-all)
next
  fix x
  assume c: C = AClassAT x
  show NullT ≤ ArrT C
  using c by (cases x, simp-all)
next
  fix x
  assume c: C = InterfaceAT x
  show NullT ≤ ArrT C
  using c by (cases x, simp-all)
next
  assume c: C = BoolAT
  show NullT ≤ ArrT C
  using c by simp
next
  assume c: C = IntgAT
  show NullT ≤ ArrT C
  using c by simp
next
  assume c: C = ShortAT
  show NullT ≤ ArrT C
  using c by simp
next
  assume c: C = ByteAT
  show NullT ≤ ArrT C
  using c by simp
qed

end

8 Attributes

theory Attributes
imports ../Isabelle/Subtype
begin

This theory has to be generated as well for each program under verification. It defines the attributes of the classes and various functions on them.

datatype AttId = CounterImpl'value | UndoCounter'save
| Dummy'dummy | Counter'dummy

The last two entries are only added to demonstrate what is to happen with attributes of abstract classes and interfaces.

It would be nice if attribute names were generated in a way that keeps them short, so that the proof state does not get unreadable because of fancy long names. The generation of attribute names that is performed by the Jive tool should only add the definition class if necessary,
i.e. if there would be a name clash otherwise. For the example above, the class names are not necessary. One must be careful, though, not to generate names that might clash with names of free variables that are used subsequently.

The domain type of an attribute is the definition class (or interface) of the attribute.

definition dtype :: AttId ⇒ Javatype where
dtype f = (case f of
  CounterImpl’value ⇒ CClass T CounterImpl
  UndoCounter’save ⇒ CClass T UndoCounter
  Dummy’dummy ⇒ AClass T Dummy
  (Counter’dummy ⇒ Interface T Counter)
)

lemma dtype-simps [simp]:
dtype CounterImpl’value = CClass T CounterImpl
dtype UndoCounter’save = CClass T UndoCounter
dtype Dummy’dummy = AClass T Dummy
dtype Counter’dummy = Interface T Counter
  by (simp-all add: dtype-def dtype-def dtype-def)

For convenience, we add some functions that directly apply the selectors of the datatype Javatype.

definition cDTypeId :: AttId ⇒ CTypeId where
cDTypeId f = (case f of
  CounterImpl’value ⇒ CounterImpl
  UndoCounter’save ⇒ UndoCounter
  Dummy’dummy ⇒ undefined
  (Counter’dummy ⇒ undefined )
)

definition aDTypeId :: AttId ⇒ ATypeId where
aDTypeId f = (case f of
  CounterImpl’value ⇒ undefined
  UndoCounter’save ⇒ undefined
  Dummy’dummy ⇒ Dummy
  (Counter’dummy ⇒ undefined )
)

definition iDTypeId :: AttId ⇒ ITypeId where
iDTypeId f = (case f of
  CounterImpl’value ⇒ undefined
  UndoCounter’save ⇒ undefined
  Dummy’dummy ⇒ undefined
  (Counter’dummy ⇒ Counter )
)

lemma DTypeId-simps [simp]:
cDTypeId CounterImpl’value = CounterImpl
cDTypeId UndoCounter’save = UndoCounter
aDTypeId Dummy’dummy = Dummy
iDTypeId Counter’dummy = Counter
  by (simp-all add: cDTypeId-def aDTypeId-def iDTypeId-def)

The range type of an attribute is the type of the value stored in that attribute.

definition rtype :: AttId ⇒ Javatype where
rtype f = (case f of
  CounterImpl’value ⇒ Intg T
)
lemma rtype-simps [simp]:
  rtype CounterImpl'value = IntgT
  rtype UndoCounter'save = IntgT
  rtype Dummy'dummy = NullT
  rtype Counter'dummy = NullT
by (simp-all add: rtype-def rtype-def rtype-def)

With the datatype \texttt{CAttId} we describe the possible locations in memory for instance fields. We rule out the impossible combinations of class names and field names. For example, a \textit{CounterImpl} cannot have a \textit{save} field. A store model which provides locations for all possible combinations of the Cartesian product of class name and field name works out fine as well, because we cannot express modification of such “wrong” locations in a Java program. So we can only prove useful properties about reasonable combinations. The only drawback in such a model is that we cannot prove a property like \texttt{not-treach-ref-impl-not-reach} in theory \texttt{StoreProperties}. If the store provides locations for every combination of class name and field name, we cannot rule out reachability of certain pointer chains that go through “wrong” locations. That is why we decided to introduce the new type \texttt{CAttId}.

While \textit{AttId} describes which fields are declared in which classes and interfaces, \texttt{CAttId} describes which objects of which classes may contain which fields at run-time. Thus, \texttt{CAttId} makes the inheritance of fields visible in the formalization. There is only one such datatype because only objects of concrete classes can be created at run-time, thus only instance fields of concrete classes can occupy memory.

\textbf{definition} \texttt{catt} :: \texttt{CTypeId} ⇒ \texttt{AttId} ⇒ \texttt{CAttId} where
\begin{align*}
catt C f &= \text{(case } C \text{ of} \\
& \quad \text{CounterImpl} ⇒ \text{(case } f \text{ of} \\
& \quad \quad \text{CounterImpl'value} ⇒ \text{CounterImpl.CounterImpl'value} \\
& \quad \quad \quad \text{UndoCounter'save} ⇒ \text{undefined} \\
& \quad \quad \quad \text{Dummy'dummy} ⇒ \text{undefined} \\
& \quad \quad \quad \text{Counter'dummy} ⇒ \text{CounterImpl.Counter'dummy} \\
& \quad \quad \text{UndoCounter} ⇒ \text{(case } f \text{ of} \\
& \quad \quad \quad \text{CounterImpl'value} ⇒ \text{UndoCounter.CounterImpl'value} \\
& \quad \quad \quad \text{UndoCounter'save} ⇒ \text{UndoCounter.UndoCounter'save} \\
& \quad \quad \quad \text{Dummy'dummy} ⇒ \text{undefined} \\
& \quad \quad \quad \text{Counter'dummy} ⇒ \text{UndoCounter.Counter'dummy} \\
& \quad \text{Object} ⇒ \text{undefined} \\
& \quad \text{Exception} ⇒ \text{undefined} \\
& \quad \text{ClassCastException} ⇒ \text{undefined} \\
& \quad \text{NullPointerException} ⇒ \text{undefined} \\
& \text{undefined}
\end{align*}
9 Program-Independent Lemmas on Attributes

theory AttributesIndep
imports ../Isa-Counter-Store/Attributes
begin

lemma catt-simps [simp]:
catt CounterImpl CounterImpl'value = CounterImpl'CounterImpl'value
catt UndoCounter CounterImpl'value = UndoCounter'CounterImpl'value
catt UndoCounter UndoCounter'save = UndoCounter'UndoCounter'save
catt CounterImpl Counter'dummy = CounterImpl'Counter'dummy
catt UndoCounter Counter'dummy = UndoCounter'Counter'dummy

by (simp-all add: catt-def)

Selection of the class name of the type of the object in which the field lives. The field can only be located in a concrete class.

definition cls:: CAttId ⇒ CTypeId where
cls cf = (case cf of
    CounterImpl'CounterImpl'value ⇒ CounterImpl
   | UndoCounter'CounterImpl'value ⇒ UndoCounter
   | UndoCounter'UndoCounter'save ⇒ UndoCounter
   | CounterImpl'Counter'dummy ⇒ CounterImpl
   | UndoCounter'Counter'dummy ⇒ UndoCounter
)

lemma cls-simps [simp]:
cls CounterImpl'CounterImpl'value = CounterImpl
cls UndoCounter'CounterImpl'value = UndoCounter
cls UndoCounter'UndoCounter'save = UndoCounter
cls CounterImpl'Counter'dummy = CounterImpl
cls UndoCounter'Counter'dummy = UndoCounter
by (simp-all add: cls-def)

Selection of the field name.

definition att:: CAttId ⇒ AttId where
att cf = (case cf of
  CounterImpl'CounterImpl'value ⇒ CounterImpl'value
  | UndoCounter'CounterImpl'value ⇒ CounterImpl'value
  | UndoCounter'UndoCounter'save ⇒ UndoCounter'save
  | CounterImpl'Counter'dummy ⇒ Counter'dummy
  | UndoCounter'Counter'dummy ⇒ Counter'dummy
)

lemma att-simps [simp]:
att CounterImpl'CounterImpl'value = CounterImpl'value
att UndoCounter'CounterImpl'value = CounterImpl'value
att UndoCounter'UndoCounter'save = UndoCounter'save
att CounterImpl'Counter'dummy = Counter'dummy
att UndoCounter'Counter'dummy = Counter'dummy
by (simp-all add: att-def)

end
The following lemmas validate the functions defined in the Attributes theory. They also aid in subsequent proving tasks. Since they are program-independent, it is of no use to add them to the generation process of Attributes.thy. Therefore, they have been extracted to this theory.

**lemma cls-catt [simp]:**
\[ CClassT c \leq dtype f \implies cls (catt c f) = c \]
apply (case-tac c)
apply (case-tac ![f])
apply simp-all
— solves all goals where \( CClassT c \leq dtype f \)
apply (fastforce elim: subtype-wrong-elims simp add: subtype-defs)+
— solves all the rest where \( \neg CClassT c \leq dtype f \) can be derived

**done**

**lemma att-catt [simp]:**
\[ CClassT c \leq dtype f \implies att (catt c f) = f \]
apply (case-tac c)
apply (case-tac ![f])
apply simp-all
— solves all goals where \( CClassT c \leq dtype f \)
apply (fastforce elim: subtype-wrong-elims simp add: subtype-defs)+
— solves all the rest where \( \neg CClassT c \leq dtype f \) can be derived

**done**

The following lemmas are just a demonstration of simplification.

**lemma rtype-att-catt:**
\[ CClassT c \leq dtype f \implies rtype (att (catt c f)) = rtype f \]
by simp

**lemma widen-cls-dtype-att [simp,intro]:**
\[ (CClassT (cls cf) \leq dtype (att cf)) \]
by (cases cf, simp-all)

**end**

## 10 Value

**theory Value imports Subtype begin**

This theory contains our model of the values in the store. The store is untyped, therefore all types that exist in Java are wrapped into one type \emph{Value}.

In a first approach, the primitive Java types supported in this formalization are mapped to similar Isabelle types. Later, we will have proper formalizations of the Java types in Isabelle, which will then be used here.

**type-synonym JavaInt = int**
**type-synonym JavaShort = int**
**type-synonym JavaByte = int**
**type-synonym JavaBoolean = bool**

The objects of each class are identified by a unique ID. We use elements of type \emph{nat} here, but in general it is sufficient to use an infinite type with a successor function and a comparison predicate.
The definition of the datatype `Value`. Values can be of the Java types boolean, int, short and byte. Additionally, they can be an object reference, an array reference or the value null.

```plaintext
datatype Value = boolV JavaBoolean
                   | intgV JavaInt
                   | shortV JavaShort
                   | byteV JavaByte
                   | objV CTypeId ObjectId — typed object reference
                   | arrV Arraytype ObjectId — typed array reference
                   | nullV
```

Arrays are modeled as references just like objects. So they can be viewed as special kinds of objects, like in Java.

### 10.1 Discriminator Functions

To test values, we define the following discriminator functions.

**definition** `isBoolV :: Value ⇒ bool where`

```plaintext
isBoolV v = (case v of
   boolV b ⇒ True
   | intgV i ⇒ False
   | shortV s ⇒ False
   | byteV by ⇒ False
   | objV C a ⇒ False
   | arrV T a ⇒ False
   | nullV ⇒ False)
```

**lemma** `isBoolV-simps [simp]:`

```plaintext
isBoolV (boolV b) = True
isBoolV (intgV i) = False
isBoolV (shortV s) = False
isBoolV (byteV by) = False
isBoolV (objV C a) = False
isBoolV (arrV T a) = False
isBoolV (nullV) = False
```

**definition** `isIntgV :: Value ⇒ bool where`

```plaintext
isIntgV v = (case v of
   boolV b ⇒ False
   | intgV i ⇒ True
   | shortV s ⇒ False
   | byteV by ⇒ False
   | objV C a ⇒ False
   | arrV T a ⇒ False
   | nullV ⇒ False)
```

**lemma** `isIntgV-simps [simp]:`

```plaintext
isIntgV (boolV b) = False
isIntgV (intgV i) = True
isIntgV (shortV s) = False
```
isIntgV (byteV by) = False
isIntgV (objV C a) = False
isIntgV (arrV T a) = False
isIntgV (nullV) = False

by (simp-all add: isIntgV-def)

definition isShortV :: Value ⇒ bool where
isShortV v = (case v of
  boolV b ⇒ False
| intgV i ⇒ False
| shortV s ⇒ True
| byteV by ⇒ False
| objV C a ⇒ False
| arrV T a ⇒ False
| nullV ⇒ False)

lemma isShortV-simps [simp]:
  isShortV (boolV b) = False
  isShortV (intgV i) = False
  isShortV (shortV s) = True
  isShortV (byteV by) = False
  isShortV (objV C a) = False
  isShortV (arrV T a) = False
  isShortV (nullV) = False
by (simp-all add: isShortV-def)

definition isByteV :: Value ⇒ bool where
isByteV v = (case v of
  boolV b ⇒ False
| intgV i ⇒ False
| shortV s ⇒ False
| byteV by ⇒ True
| objV C a ⇒ False
| arrV T a ⇒ False
| nullV ⇒ False)

lemma isByteV-simps [simp]:
  isByteV (boolV b) = False
  isByteV (intgV i) = False
  isByteV (shortV s) = False
  isByteV (byteV by) = True
  isByteV (objV C a) = False
  isByteV (arrV T a) = False
  isByteV (nullV) = False
by (simp-all add: isByteV-def)

definition isRefV :: Value ⇒ bool where
isRefV v = (case v of
  boolV b ⇒ False
| intgV i ⇒ False
| shortV s ⇒ False

lemma isRefV-simps [simp]:
  isRefV (boolV b) = False
  isRefV (intgV i) = False
  isRefV (shortV s) = False
  isRefV (byteV by) = False
  isRefV (objV C a) = True
  isRefV (arrV T a) = True
  isRefV (nullV) = True
by (simp-all add: isRefV-def)

definition isObjV :: Value ⇒ bool where
isObjV v = (case v of
  boolV b ⇒ False
  intgV i ⇒ False
  shortV s ⇒ False
  byteV by ⇒ False
  objV C a ⇒ True
  arrV T a ⇒ False
  nullV ⇒ False)

lemma isObjV-simps [simp]:
  isObjV (boolV b) = False
  isObjV (intgV i) = False
  isObjV (shortV s) = False
  isObjV (byteV by) = False
  isObjV (objV c a) = True
  isObjV (arrV T a) = False
  isObjV nullV = False
by (simp-all add: isObjV-def)

definition isArrV :: Value ⇒ bool where
isArrV v = (case v of
  boolV b ⇒ False
  intgV i ⇒ False
  shortV s ⇒ False
  byteV by ⇒ False
  objV C a ⇒ False
  arrV T a ⇒ True
  nullV ⇒ False)

lemma isArrV-simps [simp]:
  isArrV (boolV b) = False
  isArrV (intgV i) = False
  isArrV (shortV s) = False
  isArrV (byteV by) = False
  isArrV (objV c a) = False
  isArrV (arrV T a) = True
isArrV nullV = False
by (simp-all add: isArrV-def)

definition isNullV :: Value ⇒ bool where
isNullV v = (case v of
  boolV b ⇒ False
| intgV i ⇒ False
| shortV s ⇒ False
| byteV by ⇒ False
| objV C a ⇒ False
| arrV T a ⇒ False
| nullV ⇒ True)

lemma isNullV-simps [simp]:
isNullV (boolV b) = False
isNullV (intgV i) = False
isNullV (shortV s) = False
isNullV (byteV by) = False
isNullV (objV c a) = False
isNullV (arrV T a) = False
isNullV nullV = True
by (simp-all add: isNullV-def)

10.2 Selector Functions

definition aI :: Value ⇒ JavaInt where
aI v = (case v of
  boolV b ⇒ undefined
| intgV i ⇒ i
| shortV sh ⇒ undefined
| byteV by ⇒ undefined
| objV C a ⇒ undefined
| arrV T a ⇒ undefined
| nullV ⇒ undefined)

lemma aI-simps [simp]:
aI (intgV i) = i
by (simp add: aI-def)

definition aB :: Value ⇒ JavaBoolean where
aB v = (case v of
  boolV b ⇒ b
| intgV i ⇒ undefined
| shortV sh ⇒ undefined
| byteV by ⇒ undefined
| objV C a ⇒ undefined
| arrV T a ⇒ undefined
| nullV ⇒ undefined)

lemma aB-simps [simp]:
aB (boolV b) = b
by (simp add: aB-def)
definition \( aSh :: \text{Value} \Rightarrow \text{JavaShort} \) where
\[
aSh v = \begin{cases} 
\text{boolV} b & \Rightarrow \text{undefined} \\
\text{intgV} i & \Rightarrow \text{undefined} \\
\text{shortV} sh & \Rightarrow sh \\
\text{byteV} by & \Rightarrow \text{undefined} \\
\text{objV} C a & \Rightarrow \text{undefined} \\
\text{arrV} T a & \Rightarrow \text{undefined} \\
\text{nullV} & \Rightarrow \text{undefined}
\end{cases}
\]

lemma \( aSh\text{-simps} \) [simp]:
\( aSh (\text{shortV} sh) = sh \)
by (simp add: \( aSh\text{-def} \))

definition \( aBy :: \text{Value} \Rightarrow \text{JavaByte} \) where
\[
aBy v = \begin{cases} 
\text{boolV} b & \Rightarrow \text{undefined} \\
\text{intgV} i & \Rightarrow \text{undefined} \\
\text{shortV} s & \Rightarrow \text{undefined} \\
\text{byteV} by & \Rightarrow by \\
\text{objV} C a & \Rightarrow \text{undefined} \\
\text{arrV} T a & \Rightarrow \text{undefined} \\
\text{nullV} & \Rightarrow \text{undefined}
\end{cases}
\]

lemma \( aBy\text{-simps} \) [simp]:
\( aBy (\text{byteV} by) = by \)
by (simp add: \( aBy\text{-def} \))

definition \( tid :: \text{Value} \Rightarrow \text{CTypeId} \) where
\[
tid v = \begin{cases} 
\text{boolV} b & \Rightarrow \text{undefined} \\
\text{intgV} i & \Rightarrow \text{undefined} \\
\text{shortV} s & \Rightarrow \text{undefined} \\
\text{byteV} by & \Rightarrow \text{undefined} \\
\text{objV} C a & \Rightarrow C \\
\text{arrV} T a & \Rightarrow \text{undefined} \\
\text{nullV} & \Rightarrow \text{undefined}
\end{cases}
\]

lemma \( tid\text{-simps} \) [simp]:
\( tid (\text{objV} C a) = C \)
by (simp add: \( tid\text{-def} \))

definition \( oid :: \text{Value} \Rightarrow \text{ObjectId} \) where
\[
oid v = \begin{cases} 
\text{boolV} b & \Rightarrow \text{undefined} \\
\text{intgV} i & \Rightarrow \text{undefined} \\
\text{shortV} s & \Rightarrow \text{undefined} \\
\text{byteV} by & \Rightarrow \text{undefined} \\
\text{objV} C a & \Rightarrow a \\
\text{arrV} T a & \Rightarrow \text{undefined} \\
\text{nullV} & \Rightarrow \text{undefined}
\end{cases}
\]

lemma \( oid\text{-simps} \) [simp]:
oid (objV C a) = a
by (simp add: oid-def)

definition jt :: Value ⇒ Javatype where
jt v = (case v of
  boolV b ⇒ undefined
| intgV i ⇒ undefined
| shortV s ⇒ undefined
| byteV by ⇒ undefined
| objV C a ⇒ undefined
| arrV T a ⇒ at2jt T
| nullV ⇒ undefined)

lemma jt-simps [simp]:
jt (arrV T a) = at2jt T
by (simp add: jt-def)

definition aid :: Value ⇒ ObjectId where
aid v = (case v of
  boolV b ⇒ undefined
| intgV i ⇒ undefined
| shortV s ⇒ undefined
| byteV by ⇒ undefined
| objV C a ⇒ undefined
| arrV T a ⇒ a
| nullV ⇒ undefined)

lemma aid-simps [simp]:
aid (arrV T a) = a
by (simp add: aid-def)

10.3 Determining the Type of a Value

To determine the type of a value, we define the function typeof. This function is often written as \(\tau\) in theoretical texts, therefore we add the appropriate syntax support.

definition typeof :: Value ⇒ Javatype where
typeof v = (case v of
  boolV b ⇒ BoolT
| intgV i ⇒ IntgT
| shortV s ⇒ ShortT
| byteV by ⇒ ByteT
| objV C a ⇒ CClassT C
| arrV T a ⇒ ArrT T
| nullV ⇒ NullT)

abbreviation tau-syntax :: Value ⇒ Javatype (\(\tau\) -
where \(\tau\) v = typeof v

lemma typeof-simps [simp]:
(\(\tau\) (boolV b)) = BoolT
(\(\tau\) (intgV i)) = IntgT
10.4 Default Initialization Values for Types

The function init yields the default initialization values for each type. For boolean, the default value is False, for the integral types, it is 0, and for the reference types, it is nullV.

**definition** init :: Javatype ⇒ Value where
init T = (case T of
  | BoolT ⇒ boolV False
  | IntgT ⇒ intgV 0
  | ShortT ⇒ shortV 0
  | ByteT ⇒ byteV 0
  | NullT ⇒ nullV
  | ArrT T ⇒ nullV
  | CClassT C ⇒ nullV
  | AClassT a ⇒ nullV
  | InterfaceT i ⇒ nullV)

**lemma** init-simps [simp]:
init BoolT = boolV False
init IntgT = intgV 0
init ShortT = shortV 0
init ByteT = byteV 0
init NullT = nullV
init (ArrT T) = nullV
init (CClassT c) = nullV
init (AClassT a) = nullV
init (InterfaceT i) = nullV
by (simp-all add: init-def)

**lemma** typeof-init-widen [simp,intro]: typeof (init T) ≤ T
proof (cases T)
  assume c: T = BoolT
  show (τ (init T)) ≤ T
    using c by simp
  next
  assume c: T = IntgT
  show (τ (init T)) ≤ T
    using c by simp
  next
  assume c: T = ShortT
  show (τ (init T)) ≤ T
    using c by simp
  next
  assume c: T = ByteT
  show (τ (init T)) ≤ T
    using c by simp
  next
assume \( c: T = \text{NullT} \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) by simp

next
fix \( x \)
assume \( c: T = \text{CClassT } x \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) by (cases \( x \), simp-all)

next
fix \( x \)
assume \( c: T = \text{AClassT } x \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) by (cases \( x \), simp-all)

next
fix \( x \)
assume \( c: T = \text{InterfaceT } x \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) by (cases \( x \), simp-all)

next
fix \( y \)
assume \( c2: x = \text{CClassAT } y \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) \( c2 \) by (cases \( y \), simp-all)

next
fix \( y \)
assume \( c2: x = \text{AClassAT } y \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) \( c2 \) by (cases \( y \), simp-all)

next
fix \( y \)
assume \( c2: x = \text{InterfaceAT } y \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) \( c2 \) by (cases \( y \), simp-all)

next
assume \( c2: x = \text{BoolAT} \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) \( c2 \) by simp

next
assume \( c2: x = \text{IntgAT} \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) \( c2 \) by simp

next
assume \( c2: x = \text{ShortAT} \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) \( c2 \) by simp

next
assume \( c2: x = \text{ByteAT} \)
show \((\tau (\text{init } T)) \leq T\)
using \( c \) \( c2 \) by simp
A storage location can be a field of an object, a static field, the length of an array, or the contents of an array.

datatype Location =
  objLoc CAttId ObjectId — field in object
  | staticLoc AttId — static field in concrete class
  | arrLenLoc Arraytype ObjectId — length of an array
  | arrLoc Arraytype ObjectId nat — contents of an array

We only directly support one-dimensional arrays. Multidimensional arrays can be simulated by arrays of references to arrays.

The function \( ltype \) yields the content type of a location.

definition \( ltype:: Location \Rightarrow Javatype \)
\[
\begin{align*}
ltype l &= \text{case } l \text{ of} \\
  &\quad \text{objLoc } cf \ a \Rightarrow rtype (att \ cf) \\
  &\quad \text{staticLoc } f \Rightarrow rtype f \\
  &\quad \text{arrLenLoc } T \ a \Rightarrow \text{IntgT} \\
  &\quad \text{arrLoc } T \ a \ i \Rightarrow \text{at2jt } T
\end{align*}
\]

lemma \( ltype\text{-}simps \) [simp]:
\[
\begin{align*}
ltype (\text{objLoc } cf \ a) &= rtype (\text{att } cf) \\
ltype (\text{staticLoc } f) &= rtype f \\
ltype (\text{arrLenLoc } T \ a) &= \text{IntgT} \\
ltype (\text{arrLoc } T \ a \ i) &= \text{at2jt } T
\end{align*}
\]
\by\ (simp-all add: \( ltype\text{-}def \))

Discriminator functions to test whether a location denotes an array length or whether it denotes a static object. Currently, the discriminator functions for object and array locations are not specified. They can be added if they are needed.

definition \( isArrLenLoc:: Location \Rightarrow bool \)
\[
\begin{align*}
isArrLenLoc l &= \text{case } l \text{ of} \\
  &\quad \text{objLoc } cf \ a \Rightarrow \text{False} \\
  &\quad \text{staticLoc } f \Rightarrow \text{False} \\
  &\quad \text{arrLenLoc } T \ a \Rightarrow \text{True} \\
  &\quad \text{arrLoc } T \ a \ i \Rightarrow \text{False}
\end{align*}
\]

lemma \( isArrLenLoc\text{-}simps \) [simp]:
\[
\begin{align*}
isArrLenLoc (\text{objLoc } cf \ a) &= \text{False} \\
isArrLenLoc (\text{staticLoc } f) &= \text{False} \\
isArrLenLoc (\text{arrLenLoc } T \ a) &= \text{True} \\
isArrLenLoc (\text{arrLoc } T \ a \ i) &= \text{False}
\end{align*}
\]
\by\ (simp-all add: \( isArrLenLoc\text{-}def \))
**definition** isStaticLoc:: Location ⇒ bool where

isStaticLoc l = (case l of

  objLoc cf a ⇒ False
|
  staticLoc f ⇒ True
|
  arrLenLoc T a ⇒ False
|
  arrLoc T a i ⇒ False)

**lemma** isStaticLoc-simps [simp]:

isStaticLoc (objLoc cf a) = False

isStaticLoc (staticLoc f) = True

isStaticLoc (arrLenLoc T a) = False

isStaticLoc (arrLoc T a i) = False

by (simp-all add: isStaticLoc-def)

The function ref yields the object or array containing the location that is passed as argument (see the function obj in [PH97, p. 43 f.]). Note that for static locations the result is nullV since static locations are not associated to any object.

**definition** ref:: Location ⇒ Value where

ref l = (case l of

  objLoc cf a ⇒ objV (cls cf) a
|
  staticLoc f ⇒ nullV
|
  arrLenLoc T a ⇒ arrV T a
|
  arrLoc T a i ⇒ arrV T a)

**lemma** ref-simps [simp]:

ref (objLoc cf a) = objV (cls cf) a

ref (staticLoc f) = nullV

ref (arrLenLoc T a) = arrV T a

ref (arrLoc T a i) = arrV T a

by (simp-all add: ref-def)

The function loc denotes the subscription of an object reference with an attribute.

**primrec** loc:: Value ⇒ AttId ⇒ Location ...

where

loc (objV c a) f = objLoc (catt c f) a

Note that we only define subscription properly for object references. For all other values we do not provide any defining equation, so they will internally be mapped to arbitrary.

The length of an array can be selected with the function arr-len.

**primrec** arr-len:: Value ⇒ Location ...

where

arr-len (arrV T a) = arrLenLoc T a

Arrays can be indexed by the function arr-loc.

**primrec** arr-loc:: Value ⇒ nat ⇒ Location ...

where

arr-loc (arrV T a) i = arrLoc T a i

The functions loc, arr-len and arr-loc define the interface between the basic store model (based on locations) and the programming language Java. Instance field access obj.x is modelled as obj..x or loc obj x (without the syntactic sugar), array length a.length with arr-len a, array indexing a[i] with a.[i] or arr-loc a i. The accessing of a static field C.f can be expressed by the location itself staticLoc C.f. Of course one can build more infrastructure to make access to instance fields and static fields more uniform. We could for example define a function static
which indicates whether a field is static or not and based on that create an \textit{objLoc} location or a \textit{staticLoc} location. But this will only complicate the actual proofs and we can already easily perform the distinction whether a field is static or not in the JIVE-frontend and therefore keep the verification simpler.

\textbf{lemma} \textit{ref-loc} [simp]: \[\text{isObjV } r; \text{typeof } r \leq \text{dtype } f \implies \text{ref } (r..f) = r\]
apply (case-tac \( r \))
apply (case-tac \(! f \))
apply (simp-all)
done

\textbf{lemma} \textit{obj-arr-loc} [simp]: \( \text{isArrV } r \implies \text{ref } (r.[i]) = r \)
by (cases \( r \)) simp-all

\textbf{lemma} \textit{obj-arr-len} [simp]: \( \text{isArrV } r \implies \text{ref } (arr-len \ r) = r \)
by (cases \( r \)) simp-all

\textbf{end}

\section{12 Store}

\textbf{theory} \textit{Store}
\textbf{imports} \textit{Location}
\textbf{begin}

\subsection{12.1 New}

The store provides a uniform interface to allocate new objects and new arrays. The constructors of this datatype distinguish both cases.

\textbf{datatype} \( \textit{New} = \textit{new-instance } C \textit{TypeId} \) — New object, can only be of a concrete class type
\( \mid \textit{new-array } \textit{Arrtype } \textit{nat} \) — New array with given size

The discriminator \textit{isNewArr} can be used to distinguish both kinds of newly created elements.

\textbf{definition} \textit{isNewArr} :: \( \textit{New} \Rightarrow \textit{bool} \) \textbf{where}
\( \textit{isNewArr } t = (\text{case } t \text{ of}
\text{ new-instance } C \Rightarrow \text{False}
\text{ new-array } T \text{ l } \Rightarrow \text{True}) \)

\textbf{lemma} \textit{isNewArr-simps} [simp]:
\( \text{isNewArr } (\text{new-instance } C) = \text{False} \)
\( \text{isNewArr } (\text{new-array } T \text{ l}) = \text{True} \)
by (simp-all add: isNewArr-def)

The function \textit{typeofNew} yields the type of the newly created created element.

\textbf{definition} \textit{typeofNew} :: \( \textit{New} \Rightarrow \textit{Javatype} \) \textbf{where}
\( \textit{typeofNew } n = (\text{case } n \text{ of}
\text{ new-instance } C \Rightarrow \text{CClassT } C
\text{ new-array } T \text{ l } \Rightarrow \text{ArrT } T) \)

\textbf{lemma} \textit{typeofNew-simps}:
\( \text{typeofNew } (\text{new-instance } C) = \text{CClassT } C \)
\( \text{typeofNew } (\text{new-array } T \text{ l}) = \text{ArrT } T \)
by (simp-all add: typeofNew-def)
12.2 The Definition of the Store

In our store model, all objects of all classes exist at all times, but only those objects that have already been allocated are alive. Objects cannot be deallocated, thus an object that once gained the aliveness status cannot lose it later on.

To model the store, we need two functions that give us fresh object Id’s for the allocation of new objects (function $newOID$) and arrays (function $newAID$) as well as a function that maps locations to their contents (function $vals$).

\begin{verbatim}
record StoreImpl = newOID :: CTypeId ⇒ ObjectId
newAID :: Arraytype ⇒ ObjectId
vals :: Location ⇒ Value
\end{verbatim}

The function $aliveImpl$ determines for a given value whether it is alive in a given store.

\begin{verbatim}
definition aliveImpl :: Value ⇒ StoreImpl ⇒ bool where
aliveImpl x s = (case x of
  boolV b ⇒ True
  | intgV i ⇒ True
  | shortV s ⇒ True
  | byteV by ⇒ True
  | objV C a ⇒ (a < newOID s C)
  | arrV T a ⇒ (a < newAID s T)
  | nullV ⇒ True)
\end{verbatim}

The store itself is defined as new type. The store ensures and maintains the following properties: All stored values are alive; for all locations whose values are not alive, the store yields the location type’s init value; and all stored values are of the correct type (i.e. of the type of the location they are stored in).

\begin{verbatim}
definition Store = { s. (∀ l. aliveImpl (vals s l) s) ∧
  (∀ l. ¬ aliveImpl (ref l) s → vals s l = init (ltype l)) ∧
  (∀ l. typeof (vals s l) ≤ ltype l)}
\end{verbatim}

One might also model the Store as axiomatic type class and prove that the type $StoreImpl$ belongs to this type class. This way, a clearer separation between the axiomatic description of the store and its properties on the one hand and the realization that has been chosen in this formalization on the other hand could be achieved. Additionally, it would be easier to make use of different store implementations that might have different additional features. This separation remains to be performed as future work.

---

2In the following, the term “objects” includes arrays. This keeps the explanations compact.
12.3 The Store Interface

The Store interface consists of five functions: \textit{access} to read the value that is stored at a location; \textit{alive} to test whether a value is alive in the store; \textit{alloc} to allocate a new element in the store; \textit{new} to read the value of a newly allocated element; \textit{update} to change the value that is stored at a location.

\textbf{consts access::} \textit{Store} $\Rightarrow$ \textit{Location} $\Rightarrow$ \textit{Value} \hspace{0.5cm} (at [71,71])
\textit{alive::} \textit{Value} $\Rightarrow$ \textit{Store} $\Rightarrow$ \textit{bool}
\textit{alloc::} \textit{Store} $\Rightarrow$ \textit{New} $\Rightarrow$ \textit{Store}
\textit{new::} \textit{Store} $\Rightarrow$ \textit{New} $\Rightarrow$ \textit{Value}
\textit{update::} \textit{Store} $\Rightarrow$ \textit{Location} $\Rightarrow$ \textit{Value} $\Rightarrow$ \textit{Store}

\textbf{nonterminal} smodifybinds \hspace{0.5cm} \textbf{and} \hspace{0.5cm} smodifybind

\textbf{syntax}
- \textit{-smodifybind} $:: [\textquoteleft a, \textquoteleft a] \Rightarrow \textit{smodifybind}$ (((2::=-/-))
  $:: \textit{smodifybind} \Rightarrow \textit{smodifybinds}$ (-)
  $:: \textit{CTypeId} \Rightarrow \textit{smodifybind}$ (-)
- \textit{-smodifybinds::} $[\textit{smodifybind}, \textit{smodifybinds}] \Rightarrow \textit{smodifybinds}$ (-/ -)
- \textit{sModify} $:: [\textquoteleft a, \textit{smodifybinds}] \Rightarrow \textquoteleft a$ ((-/(-)) [900,0] 900)

\textbf{translations}
- \textit{-sModify} \textit{s} (-\textit{smodifybinds} \textit{b} \textit{bs}) $==$ \textit{-sModify} (-\textit{sModify} \textit{s} \textit{b} \textit{bs})
\textit{s(x:=y)} $==$ \textit{CONST update} \textit{s} \textit{x} \textit{y}
\textit{s(c)} $==$ \textit{CONST alloc} \textit{s} \textit{c}

With this syntactic setup we can write chains of (array) updates and allocations like in the following term \textit{s(new-instance Node, x := y, z := intgV 3, new-array IntgAT 3, a.[i] := intgV 4, k := boolV True)}.

In the following, the definitions of the five store interface functions and some lemmas about them are given.

\textbf{defs alive-def:}
\textit{alive x s} $\equiv$ \textit{aliveImpl x} (\textit{Rep-Store s})

\textbf{lemma alive-trivial-simps} [\textit{simp, intro}:]
\textit{alive} (\textit{boolV b}) \textit{s}
\textit{alive} (\textit{intgV i}) \textit{s}
\textit{alive} (\textit{shortV sh}) \textit{s}
\textit{alive} (\textit{byteV by}) \textit{s}
\textit{alive} \textit{nullV} \textit{s}
\textit{by} (\textit{simp-all add: alive-def aliveImpl-def})

\textbf{defs access-def:}
\textit{access s l} $\equiv$ \textit{vals} (\textit{Rep-Store s}) \textit{l}

\textbf{defs update-def:}
\textit{update s l v} $\equiv$ \textit{if alive} (\textit{ref l}) \textit{s} $\wedge$ \textit{alive v} \textit{s} $\wedge$ \textit{typeof} \textit{v} $\leq$ \textit{ltypes} \textit{l}
  then \textit{Abs-Store} ((\textit{Rep-Store s})\{\textit{vals:=}\textit{(vals} (\textit{Rep-Store s})\{(l:=v))})
  else \textit{s}

\textbf{defs alloc-def:}
\textit{alloc s t} $\equiv$
\textit{(case t of}
  \textit{new-instance C}
12.4 Derived Properties of the Store

In this subsection, a number of lemmas formalize various properties of the Store. Especially the 13 axioms are proven that must hold for a modelling of a Store (see [PH97, p. 45]). They are labeled with Store1 to Store13.

**Lemma alive-init** [simp, intro]: alive (init T) s
by (cases T) (simp-all add: alive-def aliveImpl-def)

**Lemma alive-loc** [simp]:
\[\text{isObjV} \, x \land \text{typeof} \, x \leq \text{dtype} \, f \implies \text{alive} \, (\text{ref} \, (x..f)) \, s = \text{alive} \, x \, s\]
by (cases x) (simp-all)

**Lemma alive-arr-loc** [simp]:
\[\text{isArrV} \, x \implies \text{alive} \, (\text{ref} \, (x.[i])) \, s = \text{alive} \, x \, s\]
by (cases x) (simp-all)

**Lemma alive-arr-len** [simp]:
\[\text{isArrV} \, x \implies \text{alive} \, (\text{ref} \, (\text{arr-len} \, x)) \, s = \text{alive} \, x \, s\]
by (cases x) (simp-all)

**Lemma ref-arr-len-new** [simp]:
\[\text{ref} \, (\text{arr-len} \, \text{new} \, s \, (\text{new-array} \, T \, n)) = \text{new} \, s \, (\text{new-array} \, T \, n)\]
by (simp add: new-def)

**Lemma ref-arr-loc-new** [simp]:
\[\text{ref} \, ((\text{new} \, s \, (\text{new-array} \, T \, n)).[i]) = \text{new} \, s \, (\text{new-array} \, T \, n)\]
by (simp add: new-def)

**Lemma ref-loc-new** [simp]: \[\text{CClass} \, T \, C \leq \text{dtype} \, f \implies \text{ref} \, ((\text{new} \, s \, (\text{new-instance} \, C)).f) = \text{new} \, s \, (\text{new-instance} \, C)\]
by (simp add: new-def)

lemma access-type-safe [simp, intro]: typeof (s@l) ≤ ltype l
proof –
have Rep-Store s ∈ Store
  by (rule Rep-Store)
thus ?thesis
  by (auto simp add: access-def Store-def)
qed

The store is well-typed by construction.

lemma always-welltyped-store: wts OS
by (simp add: wts-def access-type-safe)

proof
  have Rep-Store s ∈ Store
    by (rule Rep-Store)
  thus ?thesis
    by (auto simp add: access-def Store-def alive-def aliveImpl-def)
qed

lemma access-unalive [simp]:
  assumes unalive: ∼ alive (ref l) s
  shows s@l = init (ltype l)
proof –
have Rep-Store s ∈ Store
  by (rule Rep-Store)
with unalive show ?thesis
  by (simp add: access-def Store-def alive-def aliveImpl-def)
qed

lemma update-induct:
  assumes skip: P s
  assumes update: [alive (ref l) s; alive v s; typeof v ≤ ltype l] ⟹
  P (Abs-Store ((Rep-Store s)(vals:=(vals (Rep-Store s))(l:=v))))
  shows P (s⟨l:=v⟩)
using update skip
by (simp add: update-def)

lemma vals-update-in-Store:
  assumes alive-l: alive (ref l) s
  assumes alive-y: alive y s
  assumes type-conform: typeof y ≤ ltype l
  shows (Rep-Store s vals := (vals (Rep-Store s))(l := y)) ∈ Store
(is ?s-upd ∈ Store)
proof –
have s: Rep-Store s ∈ Store
  by (rule Rep-Store)
  have alloc-eq: newOID ?s-upd = newOID (Rep-Store s)
by \( \text{simp} \)

have \( \forall \ l. \text{aliveImpl} (\text{vals} \ ?s-upd \ l) \ ?s-upd \)
proof
  fix \( k \)
  show \( \text{aliveImpl} (\text{vals} \ ?s-upd \ k) \ ?s-upd \)
  proof (cases \( k=\text{l} \))
    case \( \text{True} \)
    with \( \text{alive-y} \) show \( \text{thesis} \)
    by (simp add: \text{alloc-eq alive-def aliveImpl-def split: Value.splits})
  next
    case \( \text{False} \)
    from \( s \) have \( \forall \ l. \text{aliveImpl} (\text{vals} (\text{Rep-Store s} \) \ l) (\text{Rep-Store s}) \)
    by (simp add: \text{Store-def})
    with \( \text{False} \) show \( \text{thesis} \)
    by (simp add: \text{aliveImpl-def split: Value.splits})
  qed
qed

moreover
have \( \forall \ l. \neg \text{aliveImpl} (\text{ref l}) \ ?s-upd \rightarrow \text{vals} \ ?s-upd \ l = \text{init} (\text{ltype l}) \)
proof (intro allI impl)
  fix \( k \)
  assume \( \text{unalive} : \neg \text{aliveImpl} (\text{ref k}) \ ?s-upd \)
  show \( \text{vals} \ ?s-upd \ k = \text{init} (\text{ltype k}) \)
  proof
    from \( \text{unalive alive-l} \)
    have \( k \neq l \)
    by (auto simp add: \text{alive-def aliveImpl-def split: Value.splits})
    hence \( \text{vals} \ ?s-upd \ k = \text{vals} (\text{Rep-Store s} \) \ k \)
    by \( \text{simp} \)
    moreover from \( \text{unalive} \)
    have \( \neg \text{aliveImpl} (\text{ref k}) (\text{Rep-Store s}) \)
    by (simp add: \text{aliveImpl-def split: Value.splits})
    ultimately show \( \text{thesis} \)
    using \( s \) by (simp add: \text{Store-def})
  qed
qed

moreover
have \( \forall \ l. \text{typeof} (\text{vals} \ ?s-upd \ l) \leq \text{ltype l} \)
proof
  fix \( k \) show \( \text{typeof} (\text{vals} \ ?s-upd \ k) \leq \text{ltype k} \)
  proof (cases \( k=\text{l} \))
    case \( \text{True} \)
    with \( \text{type-conform} \) show \( \text{thesis} \)
    by \( \text{simp} \)
  next
    case \( \text{False} \)
    hence \( \text{vals} \ ?s-upd \ k = \text{vals} (\text{Rep-Store s} \) \ k \)
    by \( \text{simp} \)
    with \( s \) show \( \text{thesis} \)
    by (simp add: \text{Store-def})
  qed
ultimately show \( \text{thesis} \)
  by (simp add: \text{Store-def})
lemma alive-update-invariant [simp]: alive x (s(l:=y)) = alive x s
proof (rule update-induct)
  show alive x s = alive x s ..
next
  assume alive (ref l) s alive y s typeof y ≤ ltype l
  hence Rep-Store
    (Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := y))))
    = Rep-Store s(vals := (vals (Rep-Store s))(l := y))
    by (rule vals-update-in-Store [THEN Abs-Store-inverse])
  hence alive x
    (Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := y))))
    = alive x s
    by (simp add: alive-def aliveImpl-def split: Value.split)
qed

lemma access-update-other [simp]:
  assumes neq-l-m: l ≠ m
  shows s(l:=x)@@m = s@@m
proof (rule update-induct)
  show s@@m = s@@m ..
next
  assume alive (ref l) s alive x s typeof x ≤ ltype l
  hence Rep-Store
    (Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := x))))
    = Rep-Store s(vals := (vals (Rep-Store s))(l := x))
    by (rule vals-update-in-Store [THEN Abs-Store-inverse])
  with neq-l-m
  show Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := x)))@@m = s@@m
    by (auto simp add: access-def)
qed

lemma update-access-same [simp]:
  assumes alive-l: alive (ref l) s
  assumes alive-x: alive x s
  assumes widen-x-l: typeof x ≤ ltype l
  shows s(l:=x)@@l = x
proof
  from alive-l alive-x widen-x-l
  have Rep-Store
    (Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := x))))
    = Rep-Store s(vals := (vals (Rep-Store s))(l := x))
    by (rule vals-update-in-Store [THEN Abs-Store-inverse])
  hence Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := x)))@@l = x
    by (simp add: access-def)
  with alive-l alive-x widen-x-l
  show ?thesis
    by (simp add: update-def)
qed
12.4 Derived Properties of the Store

**Store4**

*lemma* `update-unalive-val` [simp,intro]: ¬ alive x s ⇒ s(l:=x) = s
by (simp add: update-def)

*lemma* `update-unalive-loc` [simp,intro]: ¬ alive (ref l) s ⇒ s(l:=x) = s
by (simp add: update-def)

*lemma* `update-type-mismatch` [simp,intro]: ¬ typeof x ≤ ltype l ⇒ s(l:=x) = s
by (simp add: update-def)

Store9

*lemma* `alive-primitive` [simp.intro]: isprimitive (typeof x) =⇒ alive x s
by (cases x) (simp-all)

Store10

*lemma* `new-unalive-old-Store` [simp]: ¬ alive (new s t) s
by (cases t) (simp-all add: alive-def aliveImpl-def new-def)

*lemma* `alloc-new-instance-in-Store`:
(Rep-Store s[(newOID := λD. if C = D then Suc (newOID (Rep-Store s) C) else newOID (Rep-Store s) D)] ∈ Store
(is ?s-alloc ∈ Store)

proof –
  have s: Rep-Store s ∈ Store
  by (rule Rep-Store)
  hence ∀ l. aliveImpl (vals (Rep-Store s) l) (Rep-Store s)
  by (simp add: Store-def)
  then
  have ∀ l. aliveImpl (vals ?s-alloc l) ?s-alloc
  by (auto intro: less-SucI simp add: aliveImpl-def split: Value.splits)
  moreover
  have ∀ l. ¬ aliveImpl (ref l) ?s-alloc ⇒ vals ?s-alloc l = init (ltype l)
  proof (intro allI impI)
    fix l
    assume ¬ aliveImpl (ref l) ?s-alloc
    hence ¬ aliveImpl (ref l) (Rep-Store s)
    by (simp add: aliveImpl-def split: Value.splits split-if_asm)
    with s have vals (Rep-Store s) l = init (ltype l)
    by (simp add: Store-def)
    thus vals ?s-alloc l = init (ltype l)
    by simp
  qed
  moreover
  from s have ∀ l. typeof vals ?s-alloc l ≤ ltype l
  by (simp add: Store-def)
  ultimately
  show ?thesis
  by (simp add: Store-def)
  qed

*lemma* `alloc-new-array-in-Store`:
(Rep-Store s [(newAID :=
\[ \lambda S. \text{if } T = S \]
\[ \quad \text{then Suc (\text{newAID (Rep-Store } S \text{) } T)} \]
\[ \quad \text{else newAID (Rep-Store } S \text{) } S, \]
\[ \quad \text{vals := (vals (Rep-Store } S)) \]
\[ \quad (\text{arrLenLoc } T) \]
\[ \quad (\text{newAID (Rep-Store } S \text{) } T) := \]
\[ \quad \text{intgV (int } n)) \] ∈ Store \]

(is \( ?s\text{-alloc } \in \text{Store} \))

proof –

have \( s \text{: Rep-Store } s \in \text{Store} \)
by (rule Rep-Store)

have \( \forall \ l. \text{aliveImpl (vals } ?s\text{-alloc } l?) \text{-alloc} \)
proof
fix \( l \) show \( \text{aliveImpl (vals } ?s\text{-alloc } l?) \text{-alloc} \)
proof (cases \( l = \text{arrLenLoc } T \text{(newAID (Rep-Store } s \text{) } T) \))
  case True
  thus \( \text{?thesis} \)
  by (simp add: aliveImpl-def split: Value.splits)
next
case False
from \( s \) have \( \forall \ l. \text{aliveImpl (vals (Rep-Store } s \text{) } l?) \text{(Rep-Store } s \text{)} \)
by (simp add: Store-def)
with False show \( \text{?thesis} \)
by (auto intro: less-SucI simp add: aliveImpl-def split: Value.splits)
qed

qed

moreover

have \( \forall \ l. \neg \text{aliveImpl (ref } l?) \text{-alloc } \rightarrow \text{vals } ?s\text{-alloc } l = \text{init (ltype } l) \)
proof (intro allI impI)
fix \( l \)
assume \( \text{unalive: } \neg \text{aliveImpl (ref } l?) \text{-alloc} \)
show \( \text{vals } ?s\text{-alloc } l = \text{init (ltype } l) \)
proof (cases \( l = \text{arrLenLoc } T \text{(newAID (Rep-Store } s \text{) } T) \))
  case True
  with \( \text{unalive show } \text{?thesis by (simp add: aliveImpl-def)} \)
next
  case False
  from \( \text{unalive} \)
  have \( \neg \text{aliveImpl (ref } l?) \text{(Rep-Store } s \text{)} \)
  by (simp add: aliveImpl-def split: Value.splits split-if-asm)
  with \( s \) have \( \text{vals (Rep-Store } s \text{) } l = \text{init (ltype } l) \)
  by (simp add: Store-def)
  with False show \( \text{?thesis} \)
  by simp
qed

qed

moreover

from \( s \) have \( \forall \ l. \text{typeof (vals } ?s\text{-alloc } l?) \leq \text{ltype } l \)
by (simp add: Store-def)
ultimately
show \( \text{?thesis} \)
by (simp add: Store-def)

qed
lemma new-alive-alloc [simp,intro]: alive (new s t) (s(t))
proof (cases t)
  case new-instance thus ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
         alloc-new-instance-in-Store [THEN Abs-Store-inverse])
next
  case new-array thus ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
         alloc-new-array-in-Store [THEN Abs-Store-inverse])
qed

lemma value-class-inhabitants:
(∀ x. typeof x = CClassT typeId −→ P x) = (∀ a. P (objV typeId a))
(is (∀ x. ?A x) = ?B)
proof
  assume ∀ x. ?A x thus ?B
    by simp
next
  assume B: ?B thus ∀ x. ?A x
    proof
      fix x from B show ?A x
        by (cases x) auto
    qed
  qed

lemma value-array-inhabitants:
(∀ x. typeof x = ArrT typeId −→ P x) = (∀ a. P (arrV typeId a))
(is (∀ x. ?A x) = ?B)
proof
  assume ∀ x. ?A x thus ?B
    by simp
next
  assume B: ?B thus ∀ x. ?A x
    proof
      fix x from B show ?A x
        by (cases x) auto
    qed
  qed

The following three lemmas are helper lemmas that are not related to the store theory. They
might as well be stored in a separate helper theory.

lemma le-Suc-eq: (∀ a. (a < Suc n) = (a < Suc m)) = (∀ a. (a < n) = (a < m))
(is (∀ a. ?A a) = (∀ a. ?B a))
proof
  assume ∀ a. ?A a thus ∀ a. ?B a
    by fastforce
next
  assume B: ∀ a. ?B a
  show ∀ a. ?A a
    proof
      fix a
      from B show ?A a
        by (cases a) simp-all
lemma all-le-eq-imp-eq: (∀ c::nat. (∀ a. (a < d) = (a < c)) −→ (d = c))
proof (induct d)
case 0 thus ?case by fastforce
next
case (Suc n c)
thus ?case
  by (cases c) (auto simp add: le-Suc-eq)
qed

lemma all-le-eq: (∀ a::nat. (a < d) = (a < c)) = (d = c)
using all-le-eq-imp-eq by auto

lemma typeof-new: typeof (new s t) = typeofNew t
  by (cases t) (simp-all add: new-def typeofNew-def)

lemma new-eq: (new s1 t = new s2 t) = 
  (∀ x. typeof x = typeofNew t −→ alive x s1 = alive x s2)
by (cases t)
  (auto simp add: new-def typeofNew-def alive-def aliveImpl-def
       value-class-inhabitants value-array-inhabitants all-le-eq)

lemma new-update [simp]: new (s ⟨l:=x⟩) t = new s t
  by (simp add: new-eq)

lemma alive-alloc-propagation:
  assumes alive-s: alive x s shows alive x (s⟨t⟩)
proof (cases t)
case new-instance with alive-s show ?thesis
  by (cases x)
    (simp-all add: alive-def aliveImpl-def alloc-def
         alloc-new-instance-in-Store [THEN Abs-Store-inverse])
next
case new-array with alive-s show ?thesis
  by (cases x)
    (simp-all add: alive-def aliveImpl-def alloc-def
         alloc-new-array-in-Store [THEN Abs-Store-inverse])
qed

lemma alive-alloc-exhaust: alive x (s⟨t⟩) = (alive x s ∨ (x = new s t))
proof
  assume alive-alloc: alive x (s⟨t⟩)
  show alive x s ∨ x = new s t
  proof (cases t)
case (new-instance C)
  with alive-alloc show ?thesis
    by (cases x) (auto split: split-if_asm
              simp add: alive-def new-def alloc-def aliveImpl-def)
12.4 Derived Properties of the Store

Next case (new-array T l)
  with alive-alloc show ?thesis
      by (cases x) (auto split: split-if-asm
          simp add: alive-def new-def alloc-def aliveImpl-def
          alloc-new-array-in-Store [THEN Abs-Store-inverse])

qed

Next

assume alive x s ∨ x = new s t
then show alive x (s(t))

proof
  assume alive x s thus ?thesis by (rule alive-alloc-propagation)

next
  assume new: x = new s t show ?thesis
  proof (cases t)
    case new-instance with new show ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
          alloc-new-instance-in-Store [THEN Abs-Store-inverse])

next
  case new-array with new show ?thesis
  by (simp add: alive-def aliveImpl-def new-def alloc-def
          alloc-new-array-in-Store [THEN Abs-Store-inverse])

qed

qed

lemma alive-alloc-cases [consumes 1]:

[ [alive x (s(s(t))); alive x s → P; x = new s t → P] → P

by (auto simp add: alive-alloc-exhaust)

lemma aliveImpl-vals-independent: aliveImpl x (s(vals := z)) = aliveImpl x s
by (cases x) (simp-all add: aliveImpl-def)

lemma access-arr-len-new-alloc [simp]:

s(new-array T l) @@ arr-len (new s (new-array T l)) = intgV (int l)

by (subst access-def)

  (simp add: new-def alloc-def alive-def
          alloc-new-array-in-Store [THEN Abs-Store-inverse] access-def)

lemma access-new [simp]:

  assumes ref-new: ref l = new s t
  assumes no-arr-len: isNewArr t → l ≠ arr-len (new s t)
  shows s(t) @@ l = init (ltype l)

proof
  from ref-new
  have ¬ alive (ref l) s
    by simp
  hence s(t) @@ l = init (ltype l)
    by simp
  moreover
  from ref-new
  have alive (ref l) (s(t))
by simp
moreover
from no-arr-len
have vals (Rep-Store (s(t))) l = s@@l
by (cases t)
  (simp-all add: alloc-def new-def access-def
   alloc-new-instance-in-Store [THEN Abs-Store-inverse]
   alloc-new-array-in-Store [THEN Abs-Store-inverse] )
ultimately show s⟨t⟩@@l = init (ltype l)
by (subst access-def) (simp)
qed

Store5. We have to take into account that the length of an array is changed during allocation.

lemma access-alloc [simp]:
  assumes no-arr-len-new: isNewArr t → l ≠ arr-len (new s t)
  shows s⟨t⟩@@l = s@@l
proof –
show ?thesis
proof (cases alive (ref l) (s⟨t⟩))
  case True
  then
  have access-alloc-vals: s⟨t⟩@@l = vals (Rep-Store (s⟨t⟩)) l
  by (simp add: access-def alloc-def)
  from True show ?thesis
proof (cases rule: alive-alloc-cases)
  assume alive-l-s: alive (ref l) s
  with new-unalive-old-Store
  have l-not-new: ref l ≠ new s t
  by fastforce
  hence vals (Rep-Store (s⟨t⟩)) l = s@@l
  by (cases t)
  (auto simp add: alloc-def new-def access-def
   alloc-new-instance-in-Store [THEN Abs-Store-inverse]
   alloc-new-array-in-Store [THEN Abs-Store-inverse])
  with access-alloc-vals
  show ?thesis
  by simp
next
  assume ref-new: ref l = new s t
  with no-arr-len-new
  have s⟨t⟩@@l = init (ltype l)
  by (simp add: access-new)
  moreover
  from ref-new have s@@l = init (ltype l)
  by simp
  ultimately
  show ?thesis by simp
qed

next
  case False
  hence s⟨t⟩@@l = init (ltype l)
  by (simp)
  moreover
  from False have ¬ alive (ref l) s
by (auto simp add: alive-alloc-propagation)

hence \( s \sigma \Gamma = \text{init} \ (\text{ltype} \ l) \)
by simp
ultimately show \( \text{thesis} \) by simp
qed
qed

Store13

lemma \( \text{Store-eqI} \): assumes \( \text{eq-alive} : \forall \ x. \ \text{alive} \ x \ s_1 = \text{alive} \ x \ s_2 \)
assumes \( \text{eq-access} : \forall \ l. \ s_1 \sigma \Gamma l = s_2 \sigma \Gamma l \)
shows \( s_1 = s_2 \)
proof (cases \( s_1 = s_2 \))
case True thus \( \text{thesis} \).
case False note neq-s1-s2 = this show \( \text{thesis} \) proof (cases newOID (Rep-Store s1) = newOID (Rep-Store s2))
case False have \( \exists \ C. \ \text{newOID} \ (\text{Rep-Store} \ s_1) \ C \neq \text{newOID} \ (\text{Rep-Store} \ s_2) \ C \)
proof (rule ccontr)
assume \( \neg \ (\exists \ C. \ \text{newOID} \ (\text{Rep-Store} \ s_1) \ C \neq \text{newOID} \ (\text{Rep-Store} \ s_2) \ C) \)
then have \( \text{newOID} \ (\text{Rep-Store} \ s_1) = \text{newOID} \ (\text{Rep-Store} \ s_2) \)
by (blast intro: ext)
with False show False ..
qed
with eq-alive obtain C
where \( \text{newOID} \ (\text{Rep-Store} \ s_1) \ C \neq \text{newOID} \ (\text{Rep-Store} \ s_2) \ C \)
\( \forall \ a. \ \text{alive} \ (\text{objV} \ C \ a) \ s_1 = \text{alive} \ (\text{objV} \ C \ a) \ s_2 \) by auto
then show \( \text{thesis} \) by (simp add: all-le-eq alive-def aliveImpl-def)
next
case True note eq-newOID = this
show \( \text{thesis} \) proof (cases newAID (Rep-Store s1) = newAID (Rep-Store s2))
case False have \( \exists \ T. \ \text{newAID} \ (\text{Rep-Store} \ s_1) \ T \neq \text{newAID} \ (\text{Rep-Store} \ s_2) \ T \)
proof (rule ccontr)
assume \( \neg \ (\exists \ T. \ \text{newAID} \ (\text{Rep-Store} \ s_1) \ T \neq \text{newAID} \ (\text{Rep-Store} \ s_2) \ T) \)
then have \( \text{newAID} \ (\text{Rep-Store} \ s_1) = \text{newAID} \ (\text{Rep-Store} \ s_2) \)
by (blast intro: ext)
with False show False ..
qed
with eq-newAID obtain T
where \( \text{newAID} \ (\text{Rep-Store} \ s_1) \ T \neq \text{newAID} \ (\text{Rep-Store} \ s_2) \ T \)
\( \forall \ a. \ \text{alive} \ (\text{arrV} \ T \ a) \ s_1 = \text{alive} \ (\text{arrV} \ T \ a) \ s_2 \) by auto
then show \( \text{thesis} \) by (simp add: all-le-eq alive-def aliveImpl-def)
next
case True note eq-newAID = this
show \( \text{thesis} \) proof (cases vals (Rep-Store s1) = vals (Rep-Store s2))
case True with eq-newOID eq-newAID
Lemma 3.1 in [Poetzsch-Heffter97]. The proof of this lemma is quite an impressive demonstration of readable Isar proofs since it closely follows the textual proof.

lemma comm:
  assumes neq-l-new: \( r e f \ l \neq new \ s \ t \)  
  assumes neq-x-new: \( x \neq new \ s \ t \)  
  shows \( s(\langle t \rangle \langle l := x \rangle) = s(\langle l := x \rangle \langle t \rangle) \)  
proof (rule Store-eqI \[ rule-format \])  
  fix \( y \)  
  show alive y (s\((\langle t \rangle \langle l := x \rangle)\)) = alive y (s\((\langle l := x \rangle \langle t \rangle)\))  
  proof  
    have alive y (s\((\langle t \rangle \langle l := x \rangle)\)) = alive y (s\((\langle t \rangle)\))  
      by (rule alive-update-invariant)  
    also have \( \ldots = (alive y s \lor (y = new s t)) \)  
      by (rule alive-alloc-exhaust)  
    also have \( \ldots = (alive y (s(\langle l := x \rangle)) \lor y = new s t) \)  
      by (simp only: alive-update-invariant)  
    also have \( \ldots = (alive y (s(\langle l := x \rangle)) \lor y = new (s(\langle l := x \rangle)) t) \)  
      proof  
        have new s t = new (s\((\langle l := x \rangle)\))\ t  
          by simp  
        thus \( \_\_ \_ \) by simp  
      qed  
    qed  
  qed  
next
  fix \( k \)
12.4 Derived Properties of the Store

show \( s(t)(l := x) @@ k = s(l := x)(t) @@ k \)

proof (cases \( l = k \))
  case False note neq-l-k = this
  show \(?thesis\)
  proof (cases isNewArr \( t \longrightarrow k \neq \text{arr-len} \ (\text{new} \ s \ t) \))
    case True
    from neq-l-k
    have \( s(t)(l := x) @@ k = s(t) @@ k \) by simp
    also from True
    have \( \ldots = s @@ k \) by simp
    also from neq-l-k
    have \( \ldots = s(l := x) @@ k \) by simp
    also from True
    have \( \ldots = s(l := x)(t) @@ k \) by simp
    finally show \(?thesis\).
  qed
  next
  case False
  then obtain \( T \ n \) where
    \( t := \text{new-array} \ T \ n \) and \( k := \text{arr-len} \ (\text{new} \ s \ t) \)
  by (cases \( t \)) auto
  from \( k \) have \( k': \text{arr-len} \ (\text{new} \ (s(l := x)) \ (\text{new-array} \ T \ n)) \)
  by simp
  from neq-l-k
  have \( s(t)(l := x) @@ k = s(t) @@ k \) by simp
  also from \( t \ k \)
  have \( \ldots = \text{intgV} \ (\text{int} \ n) \)
  by simp
  also from \( t \ k' \)
  have \( \ldots = s(l := x)(t) @@ k \)
  by (simp del: new-update)
  finally show \(?thesis\).
  qed

next
  case True note eq-l-k = this
  have lemma-3-1:
    \( \text{ref} \ l \neq \text{new} \ s \ t \Longrightarrow \text{alive} \ (\text{ref} \ l) \ (s(t)) = \text{alive} \ (\text{ref} \ l) \ s \)
  by (simp add: alive-alloc-exhaust)
  have lemma-3-2:
    \( x \neq \text{new} \ s \ t \Longrightarrow \text{alive} \ x \ (s(t)) = \text{alive} \ x \ s \)
  by (simp add: alive-alloc-exhaust)
  have lemma-3-3: \( s(l := x, t) @@ l = s(l := x) @@ l \)
  proof
    from neq-l-new have \( \text{ref} \ l \neq \text{new} \ (s(l := x)) \ t \)
    by simp
    hence isNewArr \( t \longrightarrow l \neq \text{arr-len} \ (\text{new} \ (s(l := x)) \ t) \)
    by (cases \( t \)) auto
    thus \(?thesis\)
    by (simp)
  qed
  show \(?thesis\)
  proof (cases alive \( x \ s \))
    case True note alive-x = this
    show \(?thesis\)
    proof (cases alive \( \text{ref} \ l \) \( s \))
case True note alive-l = this
  show ?thesis
  proof (cases typeof x ≤ ltype l)
    case True
    with alive-l alive-x
    have s(l:=x)@@l = x
      by (rule update-accesssame)
    moreover
    have s(t)l:=x)@@l = x
      proof
        from alive-l neq-l-new have alive (ref l) (s(t))
          by (simp add: lemma-3-1)
        moreover
        from alive-x neq-x-new have alive x (s(t))
          by (simp add: lemma-3-2)
        ultimately
        show s(t)l:=x)@@l = x
          using True by (rule update-accesssame)
      qed
    ultimately show ?thesis
      using eq-l-k lemma-3-3 by simp
  next
    case False
    thus ?thesis by simp
  qed
next
  case False note not-alive-l = this
  from not-alive-l neq-l-new have ¬ alive (ref l) (s(t))
    by (simp add: lemma-3-1)
  then have s(t)l:=x)@@l = init (ltype l)
    by simp
  also from not-alive-l have ... = s(l:=x)@@l
    by simp
  also have ... = s(l:=x)(t)@@l
    by (simp add: lemma-3-3)
  finally show ?thesis
    by (simp add: eq-l-k)
  qed
next
  case False note not-alive-x = this
  from not-alive-x neq-x-new have ¬ alive x (s(t))
    by (simp add: lemma-3-2)
  then have s(t)l:=x)@@l = s(t)@@l
    by (simp)
  also have ... = s@@l
    proof
      from neq-l-new
      have isNewArr t → l ≠ arr-len (new s t)
        by (cases t) auto
      thus ?thesis
        by (simp)
    qed
  also from not-alive-x have ... = s(l:=x)@@l
    by (simp)
  also have ... = s(l:=x)(t)@@l
theory StoreProperties
imports Store
begin

This theory formalizes advanced concepts and properties of stores.

13.1 Reachability of a Location from a Reference

For a given store, the function \( \text{reach}\) yields the set of all pairs \((l,v)\) where \(l\) is a location that is reachable from the value \(v\) (which must be a reference) in the given store. The predicate \(\text{reach}\) decides whether a location is reachable from a value in a store.

\[
\text{inductive} \quad \text{reach} :: \text{Store} \Rightarrow \text{Location} \Rightarrow \text{Value} \Rightarrow \text{bool}
\]

\[
\begin{align*}
\text{Immediate:} & \quad \text{ref} \, l \neq \text{nullV} \implies s \vdash l \text{ reachable-from (ref} \, l) \\
\text{Indirect:} \quad & \text{[| s \vdash l \text{ reachable-from (s@@k); ref} \, k \neq \text{nullV}] |} \\
& \implies s \vdash l \text{ reachable-from (ref} \, k) 
\end{align*}
\]

Note that we explicitly exclude \text{nullV} as legal reference for reachability. Keep in mind that static fields are not associated to any object, therefore \text{ref} yields \text{nullV} if invoked on static fields (see the definition of the function \text{ref}, Sect. 11). Reachability only describes the locations directly reachable from the object or array by following the pointers and should not include the static fields if we encounter a \text{nullV} reference in the pointer chain.

We formalize some properties of reachability. Especially, Lemma 3.2 as given in [PH97, p. 53] is proven.

\begin{itemize}
\item \textbf{lemma} \text{unreachable-Null:}
\item \textbf{assumes} \text{reach:} \quad s \vdash l \text{ reachable-from x} \quad \text{shows} \quad x \neq \text{nullV}
\item using \text{reach by (induct) auto}
\end{itemize}

\begin{itemize}
\item \textbf{corollary} \text{unreachable-Null-simp [simp]}:
\item \text{\neg} \quad s \vdash l \text{ reachable-from nullV}
\item by \text{(iprover dest: unreachable-Null)}
\end{itemize}

\begin{itemize}
\item \textbf{corollary} \text{unreachable-NullE [elim]}:
\item \text{s \vdash l reachable-from nullV \implies P}
\item by \text{(simp)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \text{reachObjLoc [simp,intro]}:
\end{itemize}
\[ \text{Lemma } \text{reachArrLoc} \]
\[ \text{simp.intro} : s \vdash \text{arrLoc } T \ a \ i \ \text{reachable-from } \text{arrV } T \ a \]
\[ \text{by (rule reach.Indirect [OF arrLoc T a i,simplified])} \]

\[ \text{Lemma } \text{reachArrLen} \]
\[ \text{simp.intro} : s \vdash \text{arrLenLoc } T \ a \ \text{reachable-from } \text{arrV } T \ a \]
\[ \text{by (rule reach.Indirect [OF arrLenLoc T a,simplified])} \]

\[ \text{Lemma } \text{unreachStatic} \]
\[ \text{simp} : \neg s \vdash \text{staticLoc } f \ \text{reachable-from } x \]
\[ \text{proof} \]
\[ \{\]
\[ \text{fix } y \text{ assume } s \vdash y \ \text{reachable-from } x \ y=\text{staticLoc } f \]
\[ \text{then have False} \]
\[ \text{by induct auto} \]
\[ \} \]
\[ \text{thus } ?\text{thesis} \]
\[ \text{by auto} \]
\[ \text{qed} \]

\[ \text{Lemma } \text{unreachStaticE} \]
\[ \text{simp add: unreachStatic} \]

\[ \text{Lemma } \text{reachable-from-ArrLoc-impl-Arr} \]
\[ \text{simp.intro} : \]
\[ \text{assumes reach-loc} : s \vdash l \ \text{reachable-from} \ (s@@\text{arrLoc } T \ a \ i) \]
\[ \text{shows } s \vdash l \ \text{reachable-from} \ (\text{arrV } T \ a) \]
\[ \text{using reach.Indirect [OF reach-loc]} \]
\[ \text{by simp} \]

\[ \text{Lemma } \text{reachable-from-ObjLoc-impl-Obj} \]
\[ \text{simp.intro} : \]
\[ \text{assumes reach-loc} : s \vdash l \ \text{reachable-from} \ (s@@\text{objLoc } cf \ a) \]
\[ \text{assumes } C : C=\text{cls } cf \]
\[ \text{shows } s \vdash l \ \text{reachable-from} \ (\text{objV } C \ a) \]
\[ \text{using } C \text{ reach.Indirect [OF reach-loc]} \]
\[ \text{by simp} \]

**Lemma 3.2 (i)**

\[ \text{Lemma } \text{reach-update} \]
\[ \text{simp} : \]
\[ \text{assumes unreachable-l-x} : \neg s \vdash l \ \text{reachable-from } x \]
\[ \text{shows } s(l:=y)\vdash k \ \text{reachable-from} \ x = s \vdash k \ \text{reachable-from } x \]
\[ \text{proof} \]
\[ \text{assume } s \vdash k \ \text{reachable-from } x \]
\[ \text{from this unreachable-l-x} \]
\[ \text{show } s(l:=y)\vdash k \ \text{reachable-from } x \]
\[ \text{proof (induct)} \]
\[ \text{case (Immediate } k) \]
\[ \text{have ref } k \neq \text{nullV by fact} \]
\[ \text{then show } s(l:=y)\vdash k \ \text{reachable-from} \ (\text{ref } k) \]
\[ \text{by (rule reach.Indirect)} \]
\[ \text{next} \]
\[ \text{case (Indirect } k \ m) \]
\[ \text{have hyp: } \neg s \vdash l \ \text{reachable-from} \ (s@@m) \]
\[ \quad \implies s(l:=y)\vdash k \ \text{reachable-from} \ (s@@m) \text{ by fact} \]
\[ \text{have ref } m \neq \text{nullV and } \neg s \vdash l \ \text{reachable-from} \ (\text{ref } m) \text{ by fact} \]
hence \( l \neq m \leadsto s \vdash l \text{ reachable-from } (s@@m) \)
by (auto intro: reach.intros)
with hyp have \( s(l := y) \vdash k \text{ reachable-from } (s(l := y)@@m) \)
by simp
then show \( s(l := y)\vdash k \text{ reachable-from } (\text{ref } m) \)
by (rule reach.Indirect) (rule Indirect.hyps)
qed

next
assume \( s(l := y)\vdash k \text{ reachable-from } x \)
from this unreachable-l-x
show \( s \vdash k \text{ reachable-from } x \)
proof (induct)
  case (Immediate k)
  have ref k \( \neq \text{nullV} \) by fact
  then show \( s \vdash k \text{ reachable-from } (\text{ref } k) \)
  by (rule reach.Immediate)
next
  case (Indirect k m)
  with Indirect.hyps
  have hyp: \( s \vDash l \text{ reachable-from } (s(l := y)@@m) \)
      \( \rightarrow \) \( s \vdash k \text{ reachable-from } (s(l := y)@@m) \) by simp
  have ref m \( \neq \text{nullV} \) and \( s \vdash l \text{ reachable-from } (\text{ref } m) \) by fact+
  hence \( l \neq m \rightarrow s \vdash l \text{ reachable-from } (s@@m) \)
  by (auto intro: reach.intros)
  with hyp have \( s \vdash k \text{ reachable-from } (s@@m) \)
  by simp
  thus \( s \vdash k \text{ reachable-from } (\text{ref } m) \)
  by (rule reach.Indirect) (rule Indirect.hyps)
qed
qed

Lemma 3.2 (ii)

lemma reach2:
\( \neg s \vdash l \text{ reachable-from } x \rightleftharpoons \neg s(l := y)\vdash l \text{ reachable-from } x \)
by (simp)

Lemma 3.2 (iv)

lemma reach4: \( \neg s \vdash l \text{ reachable-from } (\text{ref } k) \rightleftharpoons k \neq l \lor (\text{ref } k) = \text{nullV} \)
by (auto intro: reach.intros)

lemma reachable-isRef:
assumes reach: \( s \vdash l \text{ reachable-from } x \)
shows isRefV x
using reach
proof (induct)
  case (Immediate l)
  show \( \text{isRefV } (\text{ref } l) \)
  by (cases l) simp-all
next
  case (Indirect l k)
  show \( \text{isRefV } (\text{ref } k) \)
  by (cases k) simp-all
qed
lemma val-ArrLen-IntgT: isArrLenLoc $l \implies$ typeof $(s @@ l) = \text{IntgT}$
proof –
  assume isArrLen: isArrLenLoc $l$
  have $T$: typeof $(s @@ l) \leq \text{ltype } l$
    by (simp)
  also from isArrLen have $I$: \text{ltype } $l = \text{IntgT}$
    by (cases $l$) simp-all
  finally show $\text{thesis}$
    by (auto elim: rtranclE simp add: le-Javatype-def subtype-defs)
qed

lemma access-alloc':[simp]:
  assumes no-arr-len: $\neg$ isArrLenLoc $l$
  shows $(s(t)) @@ l = s @@ l$
proof –
  from no-arr-len
  have isNewArr $t \implies l \neq \text{arr-len (new } s t)$
    by (cases $t$) (auto simp add: new-def isArrLenLoc-def split: Location.splits)
  thus $\text{thesis}$
    by (rule access-alloc)
qed

Lemma 3.2 (v)

lemma reach-allocl [simp]: $s(t) \vdash l \text{ reachable-from } x = s \vdash l \text{ reachable-from } x$
proof
  assume $s(t) \vdash l \text{ reachable-from } x$
  thus $s \vdash l \text{ reachable-from } x$
  proof (induct)
    case (Immediate $l$)
    thus $s \vdash \text{ref } l$
      by (rule reach.intros)
  next
    case (Indirect $l$ $k$)
    have reach-$k$: $s \vdash l \text{ reachable-from } (s(t) @@ k)$ by fact
    moreover
    have $s(t) @@ k = s @@ k$
    proof –
      from reach-$k$ have isRef: isRefV $(s(t) @@ k)$
        by (rule reachable-isRef)
      have $\neg$ isArrLenLoc $k$
        by (rule ccontr, simp)
      proof
        assume isArrLenLoc $k$
        then have typeof $(s(t) @@ k) = \text{IntgT}$
          by (rule val-ArrLen-IntgT)
        with isRef
        show $\text{False}$
          by (cases $(s(t) @@ k)$) simp-all
      qed
      thus $\text{thesis}$
        by (rule access-alloc')
    qed
    ultimately have $s \vdash l \text{ reachable-from } (s @@ k)$
      by simp
thus $s \vdash l \text{ reachable-from ref } k$
    by (rule reach.intros) (rule Indirect.hyps)
qed

next
assume $s \vdash l \text{ reachable-from } x$
thus $s(t) \vdash l \text{ reachable-from } x$
proof (induct)
case (Immediate $l$)
thus $s(t) \vdash l \text{ reachable-from ref } l$
    by (rule reach.intros)
next
case (Indirect $l \ k$)
have reach-$k$: $s \langle t \rangle \vdash l \text{ reachable-from } (s @@ k)$ by fact
moreover
have $s(t) @@ k = s @@ k$
proof −
from reach-$k$ have isRef: isRefV $(s @@ k)$
    by (rule reachable-isRef)
have $\neg \text{ArrLenLoc } k$
proof (rule ccontr, simp)
assume $\text{ArrLenLoc } k$
then have typeof $(s @@ k) = \text{IntgT}$
    by (rule val-ArrLen-IntgT)
with isRef
show $\text{False}$
    by (cases $(s @@ k)$) simp-all
qed
thus $?\text{thesis}$
    by (rule access-alloc')
qed
ultimately have $s(t) \vdash l \text{ reachable-from } (s(t) @@ k)$
    by simp
thus $s(t) \vdash l \text{ reachable-from ref } k$
    by (rule reach.intros) (rule Indirect.hyps)
qed

Lemma 3.2 (vi)

lemma reach6: isprimitive(typeof $x$) $\Rightarrow$ $\neg s \vdash l \text{ reachable-from } x$
proof
assume prim: isprimitive(typeof $x$)
assume $s \vdash l \text{ reachable-from } x$
hence isRefV $x$
    by (rule reachable-isRef)
with prim show $\text{False}$
    by (cases $x$) simp-all
qed

Lemma 3.2 (iii)

lemma reach3:
assumes $k$-$y$: $\neg s \vdash k \text{ reachable-from } y$
assumes $k$-$x$: $\neg s \vdash k \text{ reachable-from } x$
shows $\neg s(t := y) \vdash k \text{ reachable-from } x$
proof
assume $s(l := y) \vdash k \text{ reachable-from } x$
from this $k \text{-y} k \text{-x}$
show False

proof (induct)
case (Immediate l)
have $\neg s \vdash l \text{ reachable-from ref } l$ and $\text{ref } l \neq \text{null}$V by fact+
thus False
by (iprover intro: reach.intros)

next
case (Indirect $m \text{-} k$)
have $k \text{-not-Null}: \text{ref } k \neq \text{null}$V by fact
have not-m-y: $\neg s \vdash m \text{ reachable-from } y$ by fact
have hyp: $\neg s \vdash m \text{ reachable-from } y; \neg s \vdash m \text{ reachable-from } (s(l := y) @@ k)$

$\Rightarrow$ False by fact
have m-upd-k: $s(l := y) \vdash m \text{ reachable-from } (s(l := y) @@ k)$ by fact
show False

proof (cases $l = k$)
case False
then have $s(l := y) @@ k = s @@ k$ by simp
moreover
from not-m-k $k \text{-not-Null}$ have $\neg s \vdash m \text{ reachable-from } (s @@ k)$
by (iprover intro: reach.intros)
ultimately show False
using not-m-y hyp by simp

case True
note eq-l-k = this
show ?thesis

proof (cases alive (ref l) $s \land alive y s \land typeaf y \leq ltype l$)
case True
with eq-l-k have $s(l := y) @@ k = y$
by simp
with not-m-y hyp show False by simp

next
case False
hence $s(l := y) = s$
by auto
moreover
from not-m-k $k \text{-not-Null}$ have $\neg s \vdash m \text{ reachable-from } (s @@ k)$
by (iprover intro: reach.intros)
ultimately show False
using not-m-y hyp by simp

qed

Lemma 3.2 (vii).

lemma unreachable-from-init [simp,intro]: $\neg s \vdash l \text{ reachable-from } (\text{init } T)$
using reach6 by (cases $T$) simp-all

lemma ref-reach-unalive:
assumes unalive-x: $\neg alive x s$
assumes l-x: $s \vdash l \text{ reachable-from } x$
shows \( x = \text{ref} \ l \)

using \( l \cdot x \ \text{unalive-x} \)

proof induct
  case (Immediate \( l \))
  show \( \text{ref} \ l = \text{ref} \ l \)
    by simp
next
case (Indirect \( l \ k \))
  have \( \text{ref} \ k \neq \text{nullV} \) by fact
  have \( \neg \ \text{alive} \ (\text{ref} \ k) \ s \) by fact
  hence \( s @@ k = \text{init} \ (l \text{type} \ k) \) by simp
  moreover have \( s \vdash l \ \text{reachable-from} \ (s @@ k) \) by fact
ultimately have \( \text{False} \) by simp
thus ?case ..
qed

lemma loc-new-reach:
  assumes \( l \): \( \text{ref} \ l = \text{new} \ s \ t \)
  assumes \( l \cdot x \): \( s \vdash l \ \text{reachable-from} \ x \)
  shows \( x = \text{new} \ s \ t \)

proof induct
  case (Immediate \( l \))
  show ?case by fact
next
case (Indirect \( l \ k \))
  hence \( s @@ k = \text{new} \ s \ t \) by iprover
  moreover have \( \neg \ \text{alive} \ (\text{new} \ s \ t) \ s \)
    by simp
  moreover
  have \( \text{alive} \ (s @@ k) \ s \)
    by simp
ultimately have \( \text{False} \) by simp
thus ?case ..
qed

Lemma 3.2 (viii)

lemma alive-reach-alive:
  assumes \( \text{alive-x} \): \( \text{alive} \ x \ s \)
  assumes \( \text{reach-l} \): \( s \vdash l \ \text{reachable-from} \ x \)
  shows \( \text{alive} \ (\text{ref} \ l) \ s \)

proof (induct)
case (Immediate \( l \))
  show ?case by fact
next
case (Indirect \( l \ k \))
  have \( \text{hgp: alive} \ (s @@ k) \ s \vdash \text{alive} \ (\text{ref} \ l) \ s \) by fact
  moreover have \( \text{alive} \ (s @@ k) \ s \) by simp
ultimately
  show \( \text{alive} \ (\text{ref} \ l) \ s \)
    by iprover
qed
Lemma 3.2 (ix)

**Lemma** reach\(_9\):

- **assumes** reach-impl-access-eq: \(\forall l. s1 \vdash l \text{ reachable-from } x \rightarrow (s1 @@ l = s2 @@ l)\)
- **shows** \(s1 \vdash l \text{ reachable-from } x = s2 \vdash l \text{ reachable-from } x\)

**proof**

- **assume** \(s1 \vdash l \text{ reachable-from } x\)
- **from** this reach-impl-access-eq
- **show** \(s2 \vdash l \text{ reachable-from } x\)

**proof** (induct)

- **case** (Immediate \(l\))
- **show** \(s2 \vdash l \text{ reachable-from } \text{ref } l\)
  - **by** (rule reach.intros) (rule Immediate.hyps)

**next**

- **case** (Indirect \(l \ k\))
- **have** hyp: \(\forall l. s1 \vdash l \text{ reachable-from } (s1 @@ k) \rightarrow s1 @@ l = s2 @@ l\)
- **by** fact
- **have** k-not-Null: ref \(k \neq \text{nullV}\) **by** fact
- **have** reach-impl-access-eq:
  - \(\forall l. s1 \vdash l \text{ reachable-from ref } k \rightarrow s1 @@ l = s2 @@ l\) **by** fact
- **have** \(s1 \vdash l \text{ reachable-from } (s1 @@ k)\) **by** fact
- **with** k-not-Null
- **have** \(s1 @@ k = s2 @@ k\)
- **by** (iprover intro: reach-impl-access-eq [rule-format] reach.intros)
- **moreover from** reach-impl-access-eq k-not-Null
- **have** \(\forall l. s1 \vdash l \text{ reachable-from } (s1 @@ k) \rightarrow s1 @@ l = s2 @@ l\)
- **by** (iprover intro: reach.intros)
- **then have** \(s2 \vdash l \text{ reachable-from } (s1 @@ k)\)
   - **by** (rule hyp)
- **ultimately have** \(s2 \vdash l \text{ reachable-from } (s1 @@ k)\)
  - **by** simp
  - thus \(s2 \vdash l \text{ reachable-from } \text{ref } k\)
  - **by** (rule reach.intros) (rule Indirect.hyps)

**qed**

**next**

- **assume** \(s2 \vdash l \text{ reachable-from } x\)
- **from** this reach-impl-access-eq
- **show** \(s1 \vdash l \text{ reachable-from } x\)

**proof** (induct)

- **case** (Immediate \(l\))
- **show** \(s1 \vdash l \text{ reachable-from } \text{ref } l\)
  - **by** (rule reach.intros) (rule Immediate.hyps)

**next**

- **case** (Indirect \(l \ k\))
- **have** hyp: \(\forall l. s1 \vdash l \text{ reachable-from } (s2 @@ k) \rightarrow s1 @@ l = s2 @@ l\)
- **by** fact
- **have** k-not-Null: ref \(k \neq \text{nullV}\) **by** fact
- **have** reach-impl-access-eq:
  - \(\forall l. s1 \vdash l \text{ reachable-from ref } k \rightarrow s1 @@ l = s2 @@ l\) **by** fact
- **have** \(s1 \vdash k \text{ reachable-from ref } k\)
  - **by** (rule reach.intros) (rule Indirect.hyps)
- **with** reach-impl-access-eq
- **have** eq-k: \(s1 @@ k = s2 @@ k\)
  - **by** simp
- **from** reach-impl-access-eq k-not-Null
13.2 Reachability of a Reference from a Reference

The predicate \( rreach \) tests whether a value is reachable from another value. This is an extension of the predicate \(oreach\) as described in [PH97, p. 54] because now arrays are handled as well.

**Definition**

\[
\begin{align*}
\text{rreach} : & \text{Store} \Rightarrow \text{Value} \Rightarrow \text{Value} \Rightarrow \text{bool} \\
&s|\neg \text{Ref \ reachable-from \ x} = (\exists \ l. \ s|\neg \text{reachable-from \ x} \land y = \text{ref \ l})
\end{align*}
\]

**Notation**

\[
\begin{align*}
\text{rreach} (\leftrightarrow \text{Ref \ reachable-from \ x})
\end{align*}
\]

13.3 Disjointness of Reachable Locations

The predicate \(disj\) tests whether two values are disjoint in a given store. Its properties as given in [PH97, Lemma 3.3, p. 54] are then proven.

**Definition**

\[
\begin{align*}
\text{disj} : & \text{Value} \Rightarrow \text{Value} \Rightarrow \text{Store} \Rightarrow \text{bool} \\
&\text{where} \\
&\text{disj \ x \ y \ s = (\forall \ l. \ s|\neg \text{reachable-from \ x} \lor \neg \text{reachable-from \ y})}
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{disjI1} : \forall \ l. \ s|\neg \text{reachable-from \ x} \Rightarrow \neg \text{reachable-from \ y} \\
\Rightarrow \disj \ x \ y \ s
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{disjI2} : \forall \ l. \ s|\neg \text{reachable-from \ y} \Rightarrow \neg \text{reachable-from \ x} \\
\Rightarrow \disj \ x \ y \ s
\end{align*}
\]

**Lemma**

\[
\begin{align*}
\text{disj-cases} \ [\text{consumes 1}]: \\
\text{assumes} \ disj \ x \ y \ s \\
\text{assumes} \ \forall \ l. \ \neg \ s|\neg \text{reachable-from \ x} \Rightarrow P \\
\text{assumes} \ \forall \ l. \ \neg \ s|\neg \text{reachable-from \ y} \Rightarrow P \\
\text{shows} \ P \\
\text{using} \ assms \ by \ (\text{auto \ simp \ add: \ disj-def})
\end{align*}
\]

Lemma 3.3 (i) in [PH97]

**Lemma**

\[
\begin{align*}
\text{disjI1} : \ [\text{disj \ x \ y \ s}; \ \neg \ s|\neg \text{reachable-from \ x}; \ \neg \ s|\neg \text{reachable-from \ y}] \\
\Rightarrow \text{disj \ x \ y \ (s(l:=z))}
\end{align*}
\]

**Lemma**

Lemma 3.3 (ii)
lemma disj2:
  assumes disj-x-y: disj x y s
  assumes disj-x-z: disj x z s
  assumes unreach-l-x: ¬ s l reachable-from x
  shows disj x y (s(l:=z))
proof (rule disjI1)
  fix k
  assume reach-k-x: s(l := z) k reachable-from x
  show ¬ s(l := z) k reachable-from y
  proof
    assume reach-l-new: s l reachable-from new s t
    have unalive-new: ¬ alive (new s t) s by simp
    from this reach-l-new
    have new s t = ref l
      by (rule ref-reach-unalive)
    moreover from alive-x-s reach-l-x
    have alive (ref l) s
      by (rule alive-reach-alive)
    ultimately show False
      using unalive-new
      by simp
  qed
qed

Lemma 3.3 (iii)
lemma disj3: assumes alive-x-s: alive x s
  shows disj x (new s t) (s(t))
proof (rule disjI1, simp only: reach-alloc)
  fix l
  assume reach-l-x: s l reachable-from x
  show ¬ s l reachable-from new s t
  proof
    assume reach-l-new: s l reachable-from new s t
    have unalive-new: ¬ alive (new s t) s by simp
    from this reach-l-new
    have new s t = ref l
      by (rule ref-reach-unalive)
    moreover from alive-x-s reach-l-x
    have alive (ref l) s
      by (rule alive-reach-alive)
    ultimately show False
      using unalive-new
      by simp
  qed
qed

Lemma 3.3 (iv)
lemma disj4: [disj (objV C a) y s; CClassT C ≤ dtype f ]
  ⇒ disj (s@@(objV C a).f) y s
by (auto simp add: disj-def)

lemma disj4': [disj (arrV T a) y s ]
  ⇒ disj (s@@(arrV T a).[i]) y s
by (auto simp add: disj-def)
13.4 X-Equivalence

We call two stores $s_1$ and $s_2$ equivalent wrt. a given value $X$ (which is called X-equivalence) iff $X$ and all values reachable from $X$ in $s_1$ or $s_2$ have the same state [PH97, p. 55]. This is tested by the predicate $xeq$. Lemma 3.4 of [PH97] is then proven for $xeq$.

definition $xeq$: $Value \Rightarrow Store \Rightarrow Store \Rightarrow bool$ where
$xeq\ x\ s\ t = (alive\ x\ s = alive\ x\ t \land \forall l. s \vdash l\ reachable-from\ x \rightarrow s\@l = t\@l)$

abbreviation $xeq-syntax$: $Store \Rightarrow Value \Rightarrow Store \Rightarrow bool$
$(-/(==[[-]])/ -[900,0,900] 900)$

where $s ==[x] t == xeq\ x\ s\ t$

notation $(xsymbols)$ $xeq-syntax (-/(\equiv[[-]])/ -[900,0,900] 900)$

lemma $xeq1$: $[alive\ x\ s = alive\ x\ t; l. s \vdash l\ reachable-from\ x \rightarrow s\@l = t\@l] \Longrightarrow s \equiv[x] t$
by (auto simp add: $xeq$-def)

Lemma 3.4 (i) in [PH97].

lemma $xeq1-refl$: $s \equiv[x] s$
by (simp add: $xeq$-def)

Lemma 3.4 (i)

lemma $xeq1-sym'$:
assumes $s-t$: $s \equiv[x] t$
shows $t \equiv[x] s$
proof -
from $s-t$ have $alive\ x\ s = alive\ x\ t$ by (simp add: $xeq$-def)
moreover
from $s-t$ have $\forall l. s \vdash l\ reachable-from\ x \rightarrow s\@l = t\@l$
by (simp add: $xeq$-def)
with $reach9$ [OF this]
have $\forall l. t \vdash l\ reachable-from\ x \rightarrow t\@l = s\@l$
by simp
ultimately show $?thesis$
by (simp add: $xeq$-def)
qed

lemma $xeq1-sym$: $s \equiv[x] t \equiv t \equiv[x] s$
by (auto intro: $xeq1-sym'$)

Lemma 3.4 (i)

lemma $xeq1-trans$ [trans]:
assumes $s-t$: $s \equiv[x] t$
assumes $t-r$: $t \equiv[x] r$
shows $s \equiv[x] r$
proof -
from $s-t\ t-r$ have $alive\ x\ s = alive\ x\ r$
by (simp add: $xeq$-def)
moreover
have $\forall\ l.\ s \vdash l \text{ reachable-from } x \rightarrow s \circ \circ l = t \circ \circ l$
proof (intro allI impI)
fix $l$
assume reach-$l$: $s \vdash l \text{ reachable-from } x$
show $s \circ \circ l = r \circ \circ l$
proof
from reach-$l$ s-t have $s \circ \circ l = t \circ \circ l$
by (simp add: xeq-def)
also have $t \circ \circ l = r \circ \circ l$
proof
from s-t have $\forall\ l.\ s \vdash l \text{ reachable-from } x \rightarrow s \circ \circ l = t \circ \circ l$
by (simp add: xeq-def)
from reach9 [OF this] reach-$l$ have $t \vdash l \text{ reachable-from } x$
by simp
with t-r show ?thesis
by (simp add: xeq-def)
qed
finally show ?thesis.
qed
qed
ultimately show ?thesis
by (simp add: xeq-def)
qed

Lemma 3.4 (ii)

lemma xeq2:
assumes xeq: $\forall\ x.\ s \equiv [x] t$
assumes static-eq: $\forall\ f.\ s \circ \circ (\text{staticLoc } f) = t \circ \circ (\text{staticLoc } f)$
shows $s = t$
proof (rule Store-eqI)
from xeq
show $\forall\ x.\ \text{alive } x\ s = \text{alive } x\ t$
by (simp add: xeq-def)
next
show $\forall\ l.\ s \circ \circ l = t \circ \circ l$
proof
fix $l$
show $s \circ \circ l = t \circ \circ l$
proof (cases $l$)
case (objLoc $cf\ a$
have $l = \text{objLoc } cf\ a$ by fact
hence $s \vdash l \text{ reachable-from } (\text{objV } (\text{cls } cf)\ a)$
by simp
with xeq show ?thesis
by (simp add: xeq-def)
next
case (staticLoc $f$
have $l = \text{staticLoc } f$ by fact
with static-eq show ?thesis
by (simp add: xeq-def)
next
case (arrLenLoc $T\ a$
have $l = \text{arrLenLoc } T\ a$ by fact
hence $s \vdash l \text{ reachable-from } (\text{arr}V T a)$
  by simp
with $xeq$ show $\text{thesis}$
  by (simp add: $xeq$-def)

next
  case (arrLoc $T a i$)
  have $l = \text{arrLoc} T a i$ by fact
  hence $s \vdash l \text{ reachable-from } (\text{arr}V T a)$
  by simp
with $xeq$ show $\text{thesis}$
  by (simp add: $xeq$-def)

qed

Lemma 3.4 (iii)

lemma $xeq$3:
  assumes $\text{unreach-l}$: $\neg s \vdash l \text{ reachable-from } x$
  shows $s \equiv[x] s(l:=y)$
proof (rule $xeq$I)
  show alive $x s = alive x (s(l:=y))$
    by simp
next
  fix $k$
  assume $\text{reach-k}$: $s \vdash k \text{ reachable-from } x$
  with $\text{unreach-l}$ have $l \neq k$ by auto
  then show $s \llcorner k = s(l:=y) \llcorner k$
    by simp

qed

Lemma 3.4 (iv)

lemma $xeq$4: assumes $\text{not-new}$: $x \neq \text{new } s t$
  shows $s \equiv t s(t)$
proof (rule $xeq$I)
  from $\text{not-new}$ show alive $x s = alive x (s(t))$
    by (simp add: alive-alloc-exhaust)
next
  fix $l$
  assume $\text{reach-l}$: $s \vdash l \text{ reachable-from } x$
  show $s \llcorner l = s(t) \llcorner l$
proof (cases $\text{isNewArr } t \rightarrow l \neq \text{arr-len } (\text{new } s t)$)
  case True
  with $\text{reach-l}$ show $\text{thesis}$
    by simp
next
  case False
  then obtain $T n$ where $t = \text{new-array } T n$ and
    $l : l = \text{arr-len } (\text{new } s t)$
  by (cases $t$) auto
  hence $\text{ref } l = \text{new } s t$
  by simp
  from this $\text{reach-l}$
  have $x = \text{new } s t$
Lemma 3.4 (v)

\textbf{lemma} xeq5: s \equiv [x] t \implies s \downarrow l \text{ reachable-from } x = t \downarrow l \text{ reachable-from } x
\text{by (rule reach9) (simp add: xeq-def)}

13.5 T-Equivalence

T-equivalence is the extension of X-equivalence from values to types. Two stores are T-equivalent iff they are X-equivalent for all values of type T. This is formalized by the predicate \textit{teq} \cite[p. 55]{PH97}.

\textbf{definition} teq:: Javatype \Rightarrow Store \Rightarrow Store \Rightarrow bool \text{ where}
\text{teq } t \ s1 \ s2 = (\forall x. \text{typeof } x \leq t \implies s1 \equiv [x] s2)

13.6 Less Alive

To specify that methods have no side-effects, the following binary relation on stores plays a prominent role. It expresses that the two stores differ only in values that are alive in the store passed as first argument. This is formalized by the predicate \textit{lessalive} \cite[p. 55]{PH97}. The stores have to be X-equivalent for the references of the first store that are alive, and the values of the static fields have to be the same in both stores.

\textbf{definition} lessalive:: Store \Rightarrow Store \Rightarrow bool \text{ (-/ << - [70,71] 70)}
\text{where lessalive } s \ t = (\forall x. \text{alive } x \ s \implies s \equiv [x] t) \land (\forall f. s@@staticLoc f = t@@staticLoc f))

\textbf{abbreviation} (xsymbols)
lessalive-syntax :: Store \Rightarrow Store \Rightarrow bool \text{ (-/ <= - [70,71] 70)}
\text{where } s \leq t == lessalive s t

We define an introduction rule for the new operator.

\textbf{lemma} lessaliveI:
[\forall x. \text{alive } x \ s \implies s \equiv [x] t; \land f. s@@staticLoc f = t@@staticLoc f]
\implies s \leq t
\text{by (simp add: lessalive-def)}

It can be shown that \textit{lessalive} is reflexive, transitive and antisymmetric.

\textbf{lemma} lessalive-refl: s \leq s
\text{by (simp add: lessalive-def xeq1-refl)}

\textbf{lemma} lessalive-trans [trans]:
\text{assumes } s\rightleftharpoons t: s \leq t
\text{assumes } t\rightleftharpoons w: t \leq w
\text{shows } s \leq w
\text{proof (rule lessaliveI)}
\text{fix } x
\text{assume alive-x-s: alive } x \ s
\text{with } s\rightleftharpoons t \text{ have } s \equiv [x] t
\text{by (simp add: lessalive-def)}
\text{also}
have \( t \equiv [x] \ w \)

proof 
  from alive-x-s s\( t \) have alive x t by (simp add: lessalive-def xeq-def)
  with t-w show ?thesis 
  by (simp add: lessalive-def)
qed 

finally show s \( \equiv [x] \ w \).

next

fix \( f \)

from s\( t \) t-w show \( s@@statLoc f = w@@statLoc f \)
  by (simp add: lessalive-def)
qed

lemma lessalive-antisym:
  assumes s\( t \): s \( \ll \) t
  assumes t\( s \): t \( \ll \) s
  shows s = t
proof (rule xeq2)
  show \( \forall x. \ s \equiv [x] \ t \)
  proof
    fix \( x \)
    show s \( \equiv [x] \ t \)
    proof (cases alive x s)
      case True 
      with s\( t \) show ?thesis by (simp add: lessalive-def)
    next
      case False 
      note unalive-x-s = this
      show ?thesis 
      proof (cases alive x t)
        case True 
        with t\( s \) show ?thesis 
        by (subst xeq1-sym) (simp add: lessalive-def)
      next
      case False 
      show ?thesis 
      proof (rule xeqI)
        from False unalive-x-s show alive x s = alive x t by simp
      next
      fix \( l \)
      assume reach-s-x: \( s@ l \) reachable-from x
      with unalive-x-s have x: \( x = ref l \)
      by (rule ref-reach-unalive)
      with unalive-x-s have s@@l = init (ltype l)
      by simp
      also from reach-s-x x have \( t@ l \) reachable-from x
      by (auto intro: reach.Immediate unreachable-Null)
      with False x have t@@l = init (ltype l)
      by simp
      finally show s@@l = t@@l
      by simp
      qed
      qed
      qed
      qed
next

from s\( t \) show \( \forall f. s@@statLoc f = t@@statLoc f \)
This gives us a partial ordering on the store. Thus, the type \textit{Store} can be added to the appropriate type class \textit{ord} which lets us define the < and ≤ symbols, and to the type class \textit{order} which axiomatizes partial orderings.

\textbf{instantiation} \textit{Store :: order}

\begin{verbatim}
definition le-Store-def: \textit{s ≤ t} ↔ \textit{s ≪ t}
definition less-Store-def: (\textit{s::Store}) < \textit{t} ↔ \textit{s ≤ t} ∧ ¬ \textit{t ≤ s}
\end{verbatim}

We prove Lemma 3.5 of [PH97, p. 56] for this relation.

\textbf{Lemma 3.5 (i)}

\textbf{instance} \textbf{proof}

\begin{verbatim}
fix \textit{s t w:: Store}
{  
  show \textit{s ≤ s} by (simp add: le-Store-def lessalive-refl)  
  next  
  assume \textit{s ≤ t t ≤ w}  
  then show \textit{s ≤ w} by (unfold le-Store-def) (rule lessalive-trans)  
  next  
  assume \textit{s ≤ t t ≤ s}  
  then show \textit{s = t} by (unfold le-Store-def) (rule lessalive-antisym)  
  next  
  show (\textit{s < t}) = (\textit{s ≤ t} ∧ ¬ \textit{t ≤ s}) by (simp add: less-Store-def)
}
qed
\end{verbatim}

\textbf{Lemma 3.5 (ii)}

\textbf{lemma} \textit{lessalive2}: \([s ≪ t; alive x s] \implies alive x t\)

\begin{verbatim}
by (simp add: lessalive-def xeq-def)
\end{verbatim}

\textbf{Lemma 3.5 (iii)}

\textbf{lemma} \textit{lessalive3}:

\begin{verbatim}
assumes s-t: \textit{s ≪ t}
assumes alive: alive x s ∨ ¬ alive x t
shows \textit{s ≡[x] t}
proof (cases alive x s)
case True with s-t show \textit{?thesis} by (simp add: lessalive-def)
next
\end{verbatim}
案 False
note unalive-x-s = this
with alive have unalive-x-t: ¬ alive x t
  by simp
show thesis
proof (rule xeqI)
  from False alive show alive x s = alive x t
  by simp
next
fix l assume reach-s-x: s ⊆ l reachable-from x
with unalive-x-s have x: x = ref l
  by (rule ref-reach-unalive)
with unalive-x-s have s@@l = init (ltype l)
  by simp
also from reach-s-x x have l ⊆ l reachable-from x
  by (auto intro: reach.Immediate unreachable-Null)
with unalive-x-t x have t@@l = init (ltype l)
  by simp
finally show s@@l = t@@l
  by simp
qed

Lemma 3.5 (iv)
lemma lessalive-update [simp,intro]:
  assumes s-l: s ≪ t
  assumes unalive-l: ¬ alive (ref l) t
  shows s ≪ t(l:=x)
proof –
  from unalive-l have t(l:=x) = t
  by simp
  with s-l show thesis by simp
qed

lemma Xequ4':
  assumes alive: alive x s
  shows s ≡ x s(t)
proof –
  from alive have x ≠ new s t
  by auto
  thus thesis by (rule xeq4)
qed

Lemma 3.5 (v)
lemma lessalive-alloc [simp,intro]: s ≪ s(t)
  by (simp add: lessalive-def Xequ4')

13.7 Reachability of Types from Types

The predicate treach denotes the fact that the first type reaches the second type by stepping finitely many times from a type to the range type of one of its fields. This formalization diverges from [PH97, p. 106] in that it does not include the number of steps that are allowed to reach
the second type. Reachability of types is a static approximation of reachability in the store. If I cannot reach the type of a location from the type of a reference, I cannot reach the location from the reference. See lemma \textit{not-treach-ref-impl-not-reach} below.

\begin{verbatim}
inductive treach :: Javatype \Rightarrow Javatype \Rightarrow bool
where
  Subtype:     U \leq T \implies treach T U
  Attribute:  \begin{array}{l}
  \text{treach T S}; S \leq dtpe f; U \leq rtype f \end{array} \implies treach T U
  \| ArrLength: \begin{array}{l}
  \text{treach (ArrT AT) IntgT}
  \end{array}
  \| ArrElem: \begin{array}{l}
  \text{treach (ArrT AT) (at2jt AT)}
  \end{array}
  \| Trans [trans]: \begin{array}{l}
  \text{treach T U; treach U V} \implies treach T V
  \end{array}

lemma treach-ref-l [simp,intro]:
  assumes not-Null: \text{ref l \neq nullV}
  shows treach (typeof (ref l)) (ltype l)
proof (cases l)
  case (objLoc cf a)
  have l = objLoc cf a by fact
  moreover
  have treach (CClassT (cls cf)) (rtype (att cf))
    by (rule treach.Attribute [where ?f = att cf and ?S = CClassT (cls cf)])
  (auto intro: treach.Subtype)
  ultimately show ?thesis
    by simp
  next
  case (staticLoc f)
  have l = staticLoc f by fact
  hence ref l = nullV by simp
  with not-Null show ?thesis
    by simp
next
  case (arrLenLoc T a)
  have l = arrLenLoc T a by fact
  then show ?thesis
    by (auto intro: treach.ArrLength)
next
  case (arrLoc T a i)
  have l = arrLoc T a i by fact
  then show ?thesis
    by (auto intro: treach.ArrElem)
qed

lemma treach-ref-l' [simp,intro]:
  assumes not-Null: \text{ref l \neq nullV}
  shows treach (typeof (ref l)) (typeof (s@@l))
proof --
  from not-Null have treach (typeof (ref l)) (ltype l) by (rule treach-ref-l)
  also have typeof (s@@l) \leq ltype l
    by simp
  hence treach (ltype l) (typeof (s@@l))
    by (rule treach.intros)
  finally show ?thesis .
\end{verbatim}
lemma reach-impl-treach:
  assumes reach-l: s |- l reachable-from x
  shows treach (typeof x) (ltype l)
using reach-l
proof (induct)
  case (Immediate l)
  have ref l ≠ nullV by fact
  then show treach (typeof (ref l)) (ltype l)
    by (rule treach-ref-l)
next
  case (Indirect l k)
  have treach (typeof (s@@k)) (ltype l) by fact
  moreover
  have ref k ≠ nullV by fact
  hence treach (typeof (ref k)) (typeof (s@@k))
    by simp
  ultimately show treach (typeof (ref k)) (ltype l)
    by (iprover intro: treach.Trans)
qed

lemma not-treach-ref-impl-not-reach:
  assumes not-treach: ¬ treach (typeof x) (typeof (ref l))
  shows ¬ s |- l reachable-from x
proof
  assume reach-l: s |- l reachable-from x
  from this not-treach
  show False
  proof (induct)
    case (Immediate l)
    have ¬ treach (typeof (ref l)) (typeof (ref l)) by fact
    thus False by (iprover intro: treach.intros order-refl)
  next
    case (Indirect l k)
    have hyp: ¬ treach (typeof (s@@k)) (typeof (ref l)) ⇒ False by fact
    have not-Null: ref k ≠ nullV by fact
    have not-k-l: ¬ treach (typeof (ref k)) (typeof (ref l)) by fact
    show False
    proof (cases treach (typeof (s@@k)) (typeof (ref l)))
      case False thus False by (rule hyp)
    next
      case True
      from not-Null have treach (typeof (ref k)) (typeof (s@@k))
        by (rule treach-ref-l)
      also note True
      finally have treach (typeof (ref k)) (typeof (ref l))
        . with not-k-l show False ..
    qed
  qed

Lemma 4.6 in [PH97, p. 107].
lemma treach1:
    assumes x-t: typeof x ≤ T
    assumes not-treach: ¬ treach T (typeof (ref l))
    shows ¬ s ⊨ l reachable-from x
proof –
    have ¬ treach (typeof x) (typeof (ref l))
    proof
      from x-t have treach T (typeof x) by (rule treach.intros)
      also assume treach (typeof x) (typeof (ref l))
      finally have treach T (typeof (ref l)) .
      with not-treach show False ..
    qed
thus ?thesis
  by (rule not-treach-ref-impl-not-reach)
qed

end

14 The Formalization of JML Operators

theory JML imports ../Isabelle-Store/StoreProperties begin

JML operators that are to be used in Hoare formulae can be formalized here.

definition instanceof :: Value ⇒ Javatype ⇒ bool (- @instanceof -)
where
instanceof v t = (typeof v ≤ t)

end

15 The Universal Specification

theory UnivSpec imports ../Isabelle/JML begin

This theory contains the Isabelle formalization of the program-dependent specification. This
theory has to be provided by the user. In later versions of Jive, one may be able to generate it
from JML model classes.

definition aCounter :: Value ⇒ Store ⇒ JavaInt where
aCounter x s =
  (if x ≠ nullV & (alive x s) & typeof x = CClassT CounterImpl then
   aI ( s@@x.CounterImpl'value )
  else undefined)

end

References


