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 Simiplifier 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 68

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 \( A \vdash \{ P \} \, pp \, \{ Q \} \) and express that for all states \( S \) that fulfill \( P \), if the execution of the program part \( pp \) terminates, the state that is reached when \( pp \) 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\(^1\):

\[
PP \Rightarrow P \quad A \vdash \{ P \} \, pp \, \{ Q \} \quad Q \Rightarrow QQ \\
\quad A \vdash \{ PP \} \, pp \, \{ QQ \}
\]

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

\(^1\)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.
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 undo()
{
    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 | NullPointerException
— The last line contains the classes that exist in every program by default.

datatype ITypeId = Counter
datatype ATypeId = 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.

datatype Arraytype = BoolAT | IntgAT | ShortAT | ByteAT
    | CClassAT CTypeId | AClassAT ATypeId
    | InterfaceAT ITypeId

datatype Javatype = BoolT | IntgT | ShortT | ByteT | NullT | ArrT Arraytype
    | CClassT CTypeId | AClassT ATypeId
    | InterfaceT ITypeId

We need a function that widens Arraytype to Javatype.

definition at2jt :: Arraytype ⇒ Javatype
where
at2jt at = (case at of
    BoolAT ⇒ BoolT
    | IntgAT ⇒ IntgT
    | ShortAT ⇒ ShortT
    | ByteAT ⇒ ByteT
    | CClassAT CTypeId ⇒ CClassT CTypeId
    | AClassAT ATypeId ⇒ AClassT ATypeId
    | InterfaceAT ITypeId ⇒ InterfaceT ITypeId)

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

primrec isprimitive:: Javatype ⇒ bool
where
isprimitive BoolT = True |
isprimitive IntgT = True |
isprimitive ShortT = True |
isprimitive ByteT = True |
isprimitive NullT = False |
isprimitive (ArrT T) = False |
isprimitive (CClassT c) = False |
isprimitive (AClassT c) = False |
isprimitive (InterfaceT i) = False

primrec isclass:: Javatype ⇒ bool
where
isclass BoolT = False |
isclass IntgT = False |
isclass ShortT = False |
isclass ByteT = False |
isclass NullT = False |
isclass (ArrT T) = False |
isclass (CClassT c) = True |
isclass (AClassT c) = True |
isclass (InterfaceT i) = False

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 NullT
- 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 NullT
- each array type of a primitive type and its supertype Object
- each array type of a “leaf” class or interface and its subtype NullT
- 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
\begin{document}

\section{Widening the Direct Subtype Relation}

\begin{theory}

\begin{imports}
\end{imports}

\begin{begin}

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

\subsection{Auxiliary lemmas}

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

\begin{lemma}

\begin{assumes}
neq-x-y: x \neq y
x-y-rtrancl: (x,y) \in r^*
\end{assumes}

\begin{shows}
(x,y) \in r^+
\end{shows}

\begin{using}
x-y-rtrancl neq-x-y
\end{using}

\begin{proof}
\begin{induct}
\end{induct}

\begin{assume}
x \neq x
\end{assume}

\begin{thus}
(x,x) \in r^+
\end{thus}

by simp

\end{lemma}

\end{begin}

\end{theory}

\end{document}
next
  fix y z
  assume \( x\rightarrow y \): \((x, y) \in r^*\)
  assume \( y\rightarrow z \): \((y, z) \in r\)
  assume \( x \neq y \implies (x, y) \in r^+\)
  assume \( x \neq z \)
  from \( x\rightarrow y \)
  show \((x, z) \in r^+\)
proof (cases)
  assume \( x=y \)
  with \( y\rightarrow z \) 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 rtrancl-into-trancl1)
  from this \( y\rightarrow z \)
  show \((x, z) \in r^+\).
qed

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

lemma acyclic-trancl-rtrancl:
assumes acyclic: acyclic \( r \)
shows \((x, y) \in r^+ = ((x, y) \in r^* \land x\neq y)\)
proof
  assume \( x\rightarrow y \): \((x, y) \in r^*\)
  show \((x, y) \in r^* \land x\neq y\)
proof
  from \( x\rightarrow y \) show \((x, y) \in r^+\).
next
  from \( x\rightarrow y \) acyclic show \( x\neq y \) by (auto simp add: 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 ⇒ Javatype ⇒ bool (- ≺ - [71,71] 70)
where — direct subtype relation
A ≺1 B == (A,B) ∈ direct-subtype

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

abbreviation
widen-strict-syntax :: Javatype ⇒ Javatype ⇒ 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 ≤ b
  shows c. b ≤ c → a ≤ c
  by (insert a-b, rule rtrancl-trans)

The widening relation on Javatype is antisymmetric.

lemma widen-antisym [trans]:
  assumes a-b: a ≤ b
  assumes b-c: b ≤ 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 ord allows us to overwrite the two comparison operators < and ≤. These are the
two comparison operators on Javatype that we want to use subsequently.

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

instantiation Javatype:: ord
begin

definition le-Javatype-def: A ≤ B ≡ A ≥ B

definition less-Javatype-def: A < B ≡ A ≤ B ∧ ¬ B ≤ (A::Javatype)

instance proof
  fix x y z:: Javatype
  { show x ≤ x
    by (simp add: le-Javatype-def )
  next
    assume x ≤ y y ≤ z
    then show x ≤ z
      by (unfold le-Javatype-def) (rule rtrancl-trans)
  next
    assume x ≤ y y ≤ x
    then show x = y
      apply (unfold le-Javatype-def)
      apply (rule widen-antisym)
      apply assumption +
      done
  next
    show (x < y) = (x ≤ y ∧ ¬ y ≤ 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: \( (a::\text{Javatype}) \leq b \Rightarrow a \leq b \)
  by simp

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

ML ⟨
  local val ss = simpset-of \@\{context\} in

  fun widen-tac ctxt =
    resolve-tac ctxt \@\{thms widen-asm\} THEN'
    simp-tac (put-simpset ss ctxt addsimps \@\{thms le-Javatype-def\}) THEN'
    Method.insert-tac ctxt \@\{thms direct-subtype-widened \} 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 \leq CClassT Object
  by (cases C, simp-all)

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

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

lemma Object-root-array: ArrT C \leq 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** \texttt{dtype:: AtId \xrightarrow{} Javatype where}
\[
dtype f = \begin{cases} 
\text{CounterImpl'\text{value}} & \Rightarrow \text{CClassT CounterImpl} \\
\text{UndoCounter'\text{save}} & \Rightarrow \text{CClassT UndoCounter} \\
\text{Dummy'dummy} & \Rightarrow \text{AClassT Dummy} \\
\text{Counter'dummy} & \Rightarrow \text{InterfaceT Counter} 
\end{cases}
\]

**lemma** \texttt{dtype-simps [simp]}:
\[
dtype CounterImpl'\text{value} = \text{CClassT CounterImpl} \\
dtype UndoCounter'\text{save} = \text{CClassT UndoCounter} \\
dtype Dummy'dummy = \text{AClassT Dummy} \\
dtype Counter'dummy = \text{InterfaceT Counter}
\]

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

**definition** \texttt{cDTypeId :: AtId \xrightarrow{} CTypeId where}
\[
cDTypeId f = \begin{cases} 
\text{CounterImpl'\text{value}} & \Rightarrow \text{CounterImpl} \\
\text{UndoCounter'\text{save}} & \Rightarrow \text{UndoCounter} \\
\text{Dummy'dummy} & \Rightarrow \text{undefined} \\
\text{Counter'dummy} & \Rightarrow \text{undefined}
\end{cases}
\]

**definition** \texttt{aDTypeId :: AtId \xrightarrow{} ATypeId where}
\[
aDTypeId f = \begin{cases} 
\text{CounterImpl'\text{value}} & \Rightarrow \text{undefined} \\
\text{UndoCounter'\text{save}} & \Rightarrow \text{undefined} \\
\text{Dummy'dummy} & \Rightarrow \text{Dummy} \\
\text{Counter'dummy} & \Rightarrow \text{undefined}
\end{cases}
\]

**definition** \texttt{iDTypeId :: AtId \xrightarrow{} ITypeId where}
\[
iDTypeId f = \begin{cases} 
\text{CounterImpl'\text{value}} & \Rightarrow \text{undefined} \\
\text{UndoCounter'\text{save}} & \Rightarrow \text{undefined} \\
\text{Dummy'dummy} & \Rightarrow \text{undefined} \\
\text{Counter'dummy} & \Rightarrow \text{Counter}
\end{cases}
\]

**lemma** \texttt{DTypeId-simps [simp]}:
\[
cDTypeId CounterImpl'\text{value} = \text{CounterImpl} \\
cDTypeId UndoCounter'\text{save} = \text{UndoCounter} \\
aDTypeId Dummy'dummy = \text{Dummy} \\
iDTypeId Counter'dummy = \text{Counter}
\]

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

**definition** \texttt{rtype:: AtId \xrightarrow{} Javatype where}
\[
rtype f = \begin{cases} 
\text{CounterImpl'\text{value}} & \Rightarrow \text{IntgT}
\end{cases}
\]
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 \texttt{CounterImpl} cannot have a \texttt{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 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 \texttt{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.

datatype \texttt{CAttId} = CounterImpl'CounterImpl'value | UndoCounter'CounterImpl'value
| UndoCounter'UndoCounter'save
| CounterImpl'Counter'dummy | UndoCounter'Counter'dummy

Function \texttt{catt} builds a \texttt{CAttId} from a class name and a field name. In case of the illegal combinations we just return \texttt{undefined}. We can also filter out static fields in \texttt{catt}.

definition catt :: CTypeId ⇒ AttId ⇒ CAttId where
catt C f =
 (case C of
  CounterImpl ⇒ (case f of
    CounterImpl'value ⇒ CounterImpl'CounterImpl'value
    | UndoCounter'save ⇒ undefined
    | Dummy'dummy ⇒ undefined
    | Counter'dummy ⇒ CounterImpl'Counter'dummy)
  | UndoCounter ⇒ (case f of
    CounterImpl'value ⇒ UndoCounter'CounterImpl'value
    | UndoCounter'save ⇒ UndoCounter'UndoCounter'save
    | Dummy'dummy ⇒ undefined
    | Counter'dummy ⇒ UndoCounter'Counter'dummy)
  | Object ⇒ undefined
  | Exception ⇒ undefined
  | ClassCastException ⇒ undefined
  | NullPointerException ⇒ undefined
)
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
ccls 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

9  Program-Independent Lemmas on Attributes

theory AttributesIndep
imports ..../Isa-Counter-Undo/Attributes
begin
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]:
\[ CClassT c \leq dtype f \Rightarrow cls (catt c f) = c \]
apply (case-tac c)
apply (case-tac ![f])
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

**lemma** att-catt [simp]:
\[ CClassT c \leq dtype f \Rightarrow att (catt c f) = f \]
apply (case-tac c)
apply (case-tac ![f])
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

**The following lemmas are just a demonstration of simplification.**

**lemma** rtype-att-catt:
\[ CClassT c \leq dtype f \Rightarrow rtype (att (catt c f)) = rtype f \]
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 *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 *nat* here, but in general it is sufficient to use an infinite type with a successor function and a comparison predicate.
**type-synonym** \( ObjectId = \text{nat} \)

The definition of the datatype \( \text{Value} \). 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.

**datatype** \( \text{Value} = \text{boolV JavaBoolean} \)
\[ \text{| intgV JavaInt} \]
\[ \text{| shortV JavaShort} \]
\[ \text{| byteV JavaByte} \]
\[ \text{| objV C \text{TypeID} ObjectId — typed object reference} \]
\[ \text{| arrV A \text{ArrayType} ObjectId — typed array reference} \]
\[ \text{| nullV} \]

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** \( \text{isBoolV :: Value} \Rightarrow \text{bool} \text{ where} \)
\[ \text{isBoolV v = (case v of} \]
\[ \text{\hspace{1em} boolV b \Rightarrow True} \]
\[ \text{\hspace{1em} intgV i \Rightarrow False} \]
\[ \text{\hspace{1em} shortV s \Rightarrow False} \]
\[ \text{\hspace{1em} byteV by \Rightarrow False} \]
\[ \text{\hspace{1em} objV C a \Rightarrow False} \]
\[ \text{\hspace{1em} arrV T a \Rightarrow False} \]
\[ \text{\hspace{1em} nullV \Rightarrow False} \]

**lemma** \( \text{isBoolV-simps [simp]:} \)
\[ \text{isBoolV (boolV b) \hspace{1em} = True} \]
\[ \text{isBoolV (intgV i) \hspace{1em} = False} \]
\[ \text{isBoolV (shortV s) \hspace{1em} = False} \]
\[ \text{isBoolV (byteV by) \hspace{1em} = False} \]
\[ \text{isBoolV (objV C a) \hspace{1em} = False} \]
\[ \text{isBoolV (arrV T a) \hspace{1em} = False} \]
\[ \text{isBoolV (nullV) \hspace{1em} = False} \]
\[ \text{by (simp-all add: isBoolV-def)} \]

**definition** \( \text{isIntgV :: Value} \Rightarrow \text{bool} \text{ where} \)
\[ \text{isIntgV v = (case v of} \]
\[ \text{\hspace{1em} boolV b \Rightarrow False} \]
\[ \text{\hspace{1em} intgV i \Rightarrow True} \]
\[ \text{\hspace{1em} shortV s \Rightarrow False} \]
\[ \text{\hspace{1em} byteV by \Rightarrow False} \]
\[ \text{\hspace{1em} objV C a \Rightarrow False} \]
\[ \text{\hspace{1em} arrV T a \Rightarrow False} \]
\[ \text{\hspace{1em} nullV \Rightarrow False} \]

**lemma** \( \text{isIntgV-simps [simp]:} \)
\[ \text{isIntgV (boolV b) \hspace{1em} = False} \]
\[ \text{isIntgV (intgV i) \hspace{1em} = True} \]
\[ \text{isIntgV (shortV s) \hspace{1em} = False} \]
isIntgV (byteV by)  = False
isIntgV (obj V 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]:

\[
\begin{align*}
isRefV \ (boolV \ b) &= False \\
isRefV \ (intgV \ i) &= False \\
isRefV \ (shortV \ s) &= False \\
isRefV \ (byteV \ by) &= False \\
isRefV \ (objV \ C \ a) &= True \\
isRefV \ (arrV \ T \ a) &= True \\
isRefV \ \nullV &= True
\end{align*}
\]

by (simp-all add: `isRefV-def`)

definition `isObjV :: Value ⇒ bool` where

\[
\begin{align*}
isObjV \ v &= (case \ v \ of \\
\quad boolV \ b \ ⇒ False \\
\quad intgV \ i \ ⇒ False \\
\quad shortV \ s \ ⇒ False \\
\quad byteV \ by \ ⇒ False \\
\quad objV \ C \ a \ ⇒ True \\
\quad arrV \ T \ a \ ⇒ False \\
\quad \nullV \ ⇒ False)
\end{align*}
\]

definition `isArrV :: Value ⇒ bool` where

\[
\begin{align*}
isArrV \ v &= (case \ v \ of \\
\quad boolV \ b \ ⇒ False \\
\quad intgV \ i \ ⇒ False \\
\quad shortV \ s \ ⇒ False \\
\quad byteV \ by \ ⇒ False \\
\quad objV \ C \ a \ ⇒ False \\
\quad arrV \ T \ a \ ⇒ True \\
\quad \nullV \ ⇒ False)
\end{align*}
\]

definition `isArrV::Value⇒bool` where

\[
\begin{align*}
isArrV \ v &= (case \ v \ of \\
\quad boolV \ b \ ⇒ False \\
\quad intgV \ i \ ⇒ False \\
\quad shortV \ s \ ⇒ False \\
\quad byteV \ by \ ⇒ False \\
\quad objV \ C \ a \ ⇒ False \\
\quad arrV \ T \ a \ ⇒ True \\
\quad \nullV \ ⇒ False)
\end{align*}
\]
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 :: Value ⇒ JavaShort where
aSh v = (case v of
  boolV b ⇒ undefined
  | intgV i ⇒ undefined
  | shortV s ⇒ s
  | byteV by ⇒ undefined
  | objV C a ⇒ undefined
  | arrV T a ⇒ undefined
  | nullV ⇒ undefined)

**lemma** aSh-simps [simp]:
aSh (shortV s) = s
  by (simp add: aSh-def)

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

**lemma** aBy-simps [simp]:
aBy (byteV by) = by
  by (simp add: aBy-def)

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

**lemma** tid-simps [simp]:
tid (objV C a) = C
  by (simp add: tid-def)

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

**lemma** oid-simps [simp]:
10.3 Determining the Type of a Value

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

**Definition** \( \text{typeof} :: \text{Value} \Rightarrow \text{Javatype} \) where

\[
\text{typeof } v = \begin{cases} 
\text{boolV } b & \Rightarrow \text{BoolT} \\
\text{intgV } i & \Rightarrow \text{IntgT} \\
\text{shortV } s & \Rightarrow \text{ShortT} \\
\text{byteV } by & \Rightarrow \text{ByteT} \\
\text{objV } C a & \Rightarrow \text{CClassT } C \\
\text{arrV } T a & \Rightarrow \text{ArrT } T \\
\text{nullV } & \Rightarrow \text{NullT} 
\end{cases}
\]

**Abbreviation** \( \text{tau-syntax} :: \text{Value} \Rightarrow \text{Javatype} (\tau -) \) where \( \tau v == \text{typeof } v \)

**Lemma** \( \text{typeof-simps} [\text{simp}]: \)

\( (\tau (\text{boolV } b)) = \text{BoolT} \)

\( (\tau (\text{intgV } i)) = \text{IntgT} \)
\[(\tau (\text{shortV sh})) = \text{ShortT}\]
\[(\tau (\text{byteV by})) = \text{ByteT}\]
\[(\tau (\text{objV c a})) = \text{CClassT} c\]
\[(\tau (\text{arrV t a})) = \text{ArrT} t\]
\[(\tau (\text{nullV})) = \text{NullT}\]
by (simp-all add: typeof-def)

### 10.4 Default Initialization Values for Types

The function \(\text{init}\) 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** \(\text{init} :: \text{Javatype} \Rightarrow \text{Value}\) **where**
\[
\text{init } T = (\text{case } T \text{ of }
    \begin{array}{ll}
        \text{BoolT} & \Rightarrow \text{boolV False} \\
        \text{IntgT} & \Rightarrow \text{intgV 0} \\
        \text{ShortT} & \Rightarrow \text{shortV 0} \\
        \text{ByteT} & \Rightarrow \text{byteV 0} \\
        \text{NullT} & \Rightarrow \text{nullV} \\
        \text{ArrT } T & \Rightarrow \text{nullV} \\
        \text{CClassT } c & \Rightarrow \text{nullV} \\
        \text{AClassT } a & \Rightarrow \text{nullV} \\
        \text{InterfaceT } i & \Rightarrow \text{nullV}
    \end{array}
)
\]

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

**lemma** \(\text{typeof-init-widen} [\text{simp, intro}]:\) typeof \((\text{init } T) \leq T\)

**proof (cases } T)\)
\[
\begin{align*}
    &\text{assume } c : T = \text{BoolT} \\
    &\text{show } (\tau (\text{init } T)) \leq T \\
    &\quad \text{using } c \text{ by simp}
\end{align*}
\]
next
\[
\begin{align*}
    &\text{assume } c : T = \text{IntgT} \\
    &\text{show } (\tau (\text{init } T)) \leq T \\
    &\quad \text{using } c \text{ by simp}
\end{align*}
\]
next
\[
\begin{align*}
    &\text{assume } c : T = \text{ShortT} \\
    &\text{show } (\tau (\text{init } T)) \leq T \\
    &\quad \text{using } c \text{ by simp}
\end{align*}
\]
next
\[
\begin{align*}
    &\text{assume } c : T = \text{ByteT} \\
    &\text{show } (\tau (\text{init } T)) \leq T \\
    &\quad \text{using } c \text{ by simp}
\end{align*}
\]
next
assume $c: T = NullT$
show $(\tau (\text{init } T)) \leq T$
  using $c$ by simp
next
fix $x$
assume $c: T = CClassT x$
show $(\tau (\text{init } T)) \leq T$
  using $c$ by (cases $x$, simp-all)
next
fix $x$
assume $c: T = AClassT x$
show $(\tau (\text{init } T)) \leq T$
  using $c$ by (cases $x$, simp-all)
next
fix $x$
assume $c: T = InterfaceT x$
show $(\tau (\text{init } T)) \leq T$
  using $c$ by (cases $x$, simp-all)
next
fix $x$
assume $c: T = ArrT x$
show $(\tau (\text{init } T)) \leq T$
  using $c$
proof (cases $x$)
  fix $y$
  assume $c2: x = CClassAT y$
  show $(\tau (\text{init } T)) \leq T$
    using $c$ $c2$ by (cases $y$, simp-all)
next
  fix $y$
  assume $c2: x = AClassAT y$
  show $(\tau (\text{init } T)) \leq T$
    using $c$ $c2$ by (cases $y$, simp-all)
next
  fix $y$
  assume $c2: x = InterfaceAT y$
  show $(\tau (\text{init } T)) \leq T$
    using $c$ $c2$ by (cases $y$, simp-all)
next
  assume $c2: x = BoolAT$
  show $(\tau (\text{init } T)) \leq T$
    using $c$ $c2$ by simp
next
  assume $c2: x = IntgAT$
  show $(\tau (\text{init } T)) \leq T$
    using $c$ $c2$ by simp
next
  assume $c2: x = ShortAT$
  show $(\tau (\text{init } T)) \leq T$
    using $c$ $c2$ by simp
next
  assume $c2: x = ByteAT$
  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.

```plaintext
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.

```plaintext
definition ltype:: Location ⇒ Javatype where
ltype l = (case l of
  objLoc cf a ⇒ rtype (att cf))
| staticLoc f ⇒ rtype f
| arrLenLoc T a ⇒ IntgT
| arrLoc T a i ⇒ at2jt T)
```

```plaintext
lemma ltype-simps [simp]:
ltype (objLoc cf a) = rtype (att cf)
ltype (staticLoc f) = rtype f
ltype (arrLenLoc T a) = IntgT
ltype (arrLoc T a i) = at2jt T
by (simp-all add: ltype-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.

```plaintext
definition isArrLenLoc:: Location ⇒ bool where
isArrLenLoc l = (case l of
  objLoc cf a ⇒ False
| staticLoc f ⇒ False
| arrLenLoc T a ⇒ True
| arrLoc T a i ⇒ False)
```

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

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

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

\textbf{definition} \textit{ref:: Location ⇒ Value where}
\begin{align*}
\text{ref} \ l &= \begin{cases} 
\text{objLoc} \ c f \ a \Rightarrow \text{objV} \ (\text{cls} \ cf) \ a \\
\text{staticLoc} \ f \Rightarrow \text{nullV} \\
\text{arrLenLoc} \ T \ a \Rightarrow \text{arrV} \ T \ a \\
\text{arrLoc} \ T \ a \ i \Rightarrow \text{arrV} \ T \ a
\end{cases}
\end{align*}

\textbf{lemma} \textit{ref-simps [simp]::}
\begin{align*}
\text{ref} \ (\text{objLoc} \ cf \ a) &= \text{objV} \ (\text{cls} \ cf) \ a \\
\text{ref} \ (\text{staticLoc} \ f) &= \text{nullV} \\
\text{ref} \ (\text{arrLenLoc} \ T \ a) &= \text{arrV} \ T \ a \\
\text{ref} \ (\text{arrLoc} \ T \ a \ i) &= \text{arrV} \ T \ a
\end{align*}
\begin{align*}
\text{by (simp-all add: ref-def)}
\end{align*}

The function \textit{loc} denotes the subscription of an object reference with an attribute.

\textbf{primrec} \textit{loc:: Value ⇒ AttId ⇒ Location (-..- [80,80] 80)}
\begin{align*}
\text{where} \ loc \ (\text{objV} \ c \ a \ f) &= \text{objLoc} \ (\text{catt} \ c \ f) \ a
\end{align*}

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 \textit{arbitrary}.

The length of an array can be selected with the function \textit{arr-len}.

\textbf{primrec} \textit{arr-len:: Value ⇒ Location (-..- [80,80] 80)}
\begin{align*}
\text{where} \ arr-len \ (\text{arrV} \ T \ a) &= \text{arrLenLoc} \ T \ a
\end{align*}

Arrays can be indexed by the function \textit{arr-loc}.

\textbf{primrec} \textit{arr-loc:: Value ⇒ nat ⇒ Location (-..- [80,80] 80)}
\begin{align*}
\text{where} \ arr-loc \ (\text{arrV} \ T \ a) \ i &= \text{arrLoc} \ T \ a \ i
\end{align*}

The functions \textit{loc}, \textit{arr-len} and \textit{arr-loc} define the interface between the basic store model (based on locations) and the programming language Java. Instance field access \textit{obj.x} is modelled as \textit{obj.x} or \textit{loc obj x} (without the syntactic sugar), array length \textit{a.length} with \textit{arr-len a}, array indexing \textit{a[i]} with \textit{a.[i]} or \textit{arr-loc a i}. The accessing of a static field \textit{C.f} can be expressed by the location itself \textit{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 \textit{static
which indicates whether a field is static or not and based on that create an `objLoc` location or a `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.

```
lemma ref-loc [simp]: [isObjV r; typeof r ≤ dtype f] ⇒ ref (r..f) = r
  apply (case-tac r)
  apply (case-tac ![f])
  apply (simp-all)
  done

lemma obj-arr-loc [simp]: isArrV r ⇒ ref (r.[i]) = r
  by (cases r) simp-all

lemma obj-arr-len [simp]: isArrV r ⇒ ref (arr-len r) = r
  by (cases r) simp-all
```

12 Store

theory Store
imports Location
begin

12.1 New

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

```
datatype New = new-instance CTypeId — New object, can only be of a concrete class type
   | new-array Arraytype nat — New array with given size
```

The discriminator `isNewArr` can be used to distinguish both kinds of newly created elements.

```
definition isNewArr :: New ⇒ bool where
  isNewArr t = (case t of
    new-instance C ⇒ False
    | new-array T l ⇒ True)

lemma isNewArr-simps [simp]:
  isNewArr (new-instance C) = False
  isNewArr (new-array T l) = True
  by (simp-all add: isNewArr-def)
```

The function `typeofNew` yields the type of the newly created created element.

```
definition typeofNew :: New ⇒ Javatype where
  typeofNew n = (case n of
    new-instance C ⇒ CClassT C
    | new-array T l ⇒ ArrT T)

lemma typeofNew-simps:
  typeofNew (new-instance C) = CClassT C
  typeofNew (new-array T l) = 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`).

```plaintext
class StoreImp {
    fun newOID :: CTypeId => ObjectId
    fun newAID :: Arraytype => ObjectId
    fun vals :: Location => Value
}
```

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

```plaintext
definition aliveImpl :: Value => StoreImp => 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)
```

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).

```plaintext
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)}
```

One might also model the Store as axiomatic type class and prove that the type `StoreImp` 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: \texttt{access} to read the value that is stored at a location; \texttt{alive} to test whether a value is alive in the store; \texttt{alloc} to allocate a new element in the store; \texttt{new} to read the value of a newly allocated element; \texttt{update} to change the value that is stored at a location.

\begin{verbatim}
consts access:: Store ⇒ Location ⇒ Value
alive:: Value ⇒ Store ⇒ bool
alloc:: Store ⇒ New ⇒ Store
new:: Store ⇒ New ⇒ Value
update:: Store ⇒ Location ⇒ Value ⇒ Store
\end{verbatim}

nonterminal smodifybinds and smodifybind

\textbf{syntax}

\begin{verbatim}
-smmodifbind :: ['a,'a] ⇒ smodifybind ((?::=/ -))
:: smodifybind ⇒ smodifybinds (\-
:: CTypeId ⇒ smodifybind (\-
-smmodifbinds:: [smodifybind, smodifybinds] ⇒ smodifybinds (\-/- -)
-sModify :: ['a, smodifybinds] ⇒ 'a (\-\langle\rangle [900.0] 900)
\end{verbatim}

\textbf{translations}

\begin{verbatim}
-sModify s (-smodifybinds b bs) == -sModify (-sModify s b) bs
s(x:=y) == CONST update s x y
s(c) == CONST alloc s c
\end{verbatim}

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

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

\textbf{overloading} \texttt{alive} \equiv \texttt{alive}

\begin{verbatim}
definition alive where alive x s \equiv aliveImpl x (Rep-Store s)
end
\end{verbatim}

\textbf{lemma} \texttt{alive-trivial-simps} [simp,intro]:

\begin{verbatim}
alive (boolV b) s
alive (intgV i) s
alive (shortV sh) s
alive (byteV by) s
alive nullV s
by (simp-all add: alive-def aliveImpl-def)
\end{verbatim}

\textbf{overloading}

\begin{verbatim}
access \equiv access
update \equiv update
alloc \equiv alloc
new \equiv new
\end{verbatim}

\begin{verbatim}
definition access where access s l \equiv vals (Rep-Store s) l
\end{verbatim}
12.4 Derived Properties of the Store

**Definition update**

**Where** update \(s \ l \ v \equiv\)

\[
\text{if alive (ref } l \text{)} \ s \land \text{alive } v \ s \land \text{typeof } v \leq \text{ltype } l \\
\text{then Abs-Store ((Rep-Store } s\{\text{vals:=}(\text{vals (Rep-Store } s))(l:=v)\}) \text{)} \text{else } s
\]

**Definition alloc**

**Where** alloc \(s \ t \equiv\)

\[
\text{(case } t \text{ of} \\
\text{new-instance } C \\
\Rightarrow \text{Abs-Store} \\
\((\text{Rep-Store } s\{\text{newOID := } \lambda D. \text{if } C=D \text{ then } \text{Suc (newOID (Rep-Store } s) C) \text{ else } \text{newOID (Rep-Store } s) D\})\) \\
\mid \text{new-array } T l \\
\Rightarrow \text{Abs-Store} \\
\((\text{Rep-Store } s\{\text{newAID := } \lambda S. \text{if } T=S \text{ then } \text{Suc (newAID (Rep-Store } s) T) \text{ else } \text{newAID (Rep-Store } s) S,} \text{vals := } (\text{vals (Rep-Store } s))\) \\
\text{arrLenLoc } T (\text{newAID (Rep-Store } s) T) \text{ := intV (int } l))\))
\]

**Definition new**

**Where** new \(s \ t \equiv\)

\[
\text{(case } t \text{ of} \\
\text{new-instance } C \Rightarrow \text{objV } C \text{ (newOID (Rep-Store } s) C) \\
\mid \text{new-array } T l \Rightarrow \text{arrV } T \text{ (newAID (Rep-Store } s) T)}
\]

end

The predicate \(\text{wts}\) tests whether the store is well-typed.

**Definition** \(\text{wts} :: \text{Store} \Rightarrow \text{bool}\)

\(\text{wts } OS = (\forall (l:\text{Location}) . \text{typeof } (OS@\ l)) \leq (\text{ltype } l)\)

### 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]**

\[\text{alive } (\text{init } T) \ s \text{ by (cases } T) \text{ (simp-all add: alive-def aliveImpl-def)}\]

**Lemma alive-loc [simp]**

\[\text{isObjV } x; \text{typeof } x \leq \text{dtype } f \Rightarrow \text{alive } (\text{ref } (x..f)) \ s = \text{alive } x \ s \text{ by (cases } x) \text{ (simp-all)}\]

**Lemma alive-arr-loc [simp]**

\[\text{isArrV } x \Rightarrow \text{alive } (\text{ref } (x.[i])) \ s = \text{alive } x \ s \text{ by (cases } x) \text{ (simp-all)}\]

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

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

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

\textbf{lemma ref-loc-new} \ [\text{simp}]: \\
CClassT \ C \leq \ dtype \ f \\
\implies \text{ref} \ (\text{(new} \ s \ (\text{new-instance} \ C))_.f) = \text{new} \ s \ (\text{new-instance} \ C) \\
by \ (\text{simp add: new-def})

\textbf{lemma access-type-safe} \ [\text{simp, intro}]: \\
typeof \ (s@@l) \leq \ ltype \ l \\
proof \\
\have \ Rep-Store \ s \in \ Store \\
\by \ (\text{rule Rep-Store}) \\
\thus \ ?thesis \\
\by \ (\text{auto simp add: access-def Store-def}) \\
qed

\textbf{The store is well-typed by construction.}

\textbf{lemma always-welltyped-store}: \\
wts \ OS \\
by \ (\text{simp add: wts-def access-type-safe})

\textbf{Store8}

\textbf{lemma alive-access} \ [\text{simp, intro}]: \\
alive \ (s@@l) \ s \\
proof \\
\have \ Rep-Store \ s \in \ Store \\
\by \ (\text{rule Rep-Store}) \\
\thus \ ?thesis \\
\by \ (\text{auto simp add: access-def Store-def alive-def aliveImpl-def}) \\
qed

\textbf{Store3}

\textbf{lemma access-unalive} \ [\text{simp}]: \\
\textbf{assumes} unalive: \neg \ alive \ (\text{ref} \ l) \ s \\
\textbf{shows} \ s@@l = \text{init} \ (\text{ltype} \ l) \\
proof \\
\have \ Rep-Store \ s \in \ Store \\
\by \ (\text{rule Rep-Store}) \\
\with \ unalive \ \textbf{show} \ ?thesis \\
\by \ (\text{simp add: access-def Store-def alive-def aliveImpl-def}) \\
qed

\textbf{lemma update-induct}: \\
\textbf{assumes} skip: \ P \ s \\
\textbf{assumes} update: \ [alive \ (\text{ref} \ l) \ s; alive \ v \ s; typeof \ v \leq \ ltype \ l] \implies \\
P \ (Abs-Store \ ((\text{Rep-Store} \ s)[vals:=(vals \ (\text{Rep-Store} \ s))[l:=v]]) \\
\textbf{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 simp
have ∀ l. aliveImpl (vals ?s-upd l) ?s-upd
proof
  fix k
  show aliveImpl (vals ?s-upd k) ?s-upd
  proof (cases k=l)
    case True
    with alive-y show ?thesis
    by (simp add: alloc-eq alive-def aliveImpl-def split: Value.splits)
  next
    case False
    from s have ∀ l. aliveImpl (vals (Rep-Store s) l) (Rep-Store s)
      by (simp add: Store-def)
    with False show ?thesis
    by (simp add: aliveImpl-def split: Value.splits)
  qed
qed
moreover
have ∀ l. ¬ aliveImpl (ref l) ?s-upd → vals ?s-upd l = init (ltype l)
proof (intro allI impI)
  fix k
  assume unalive: ¬ aliveImpl (ref k) ?s-upd
  show vals ?s-upd k = init (ltype k)
  proof –
    from unalive alive-l
    have k≠l
      by (auto simp add: alive-def aliveImpl-def split: Value.splits)
    hence vals ?s-upd k = vals (Rep-Store s) k
      by simp
    moreover from unalive
    have ¬ aliveImpl (ref k) (Rep-Store s)
      by (simp add: aliveImpl-def split: Value.splits)
    ultimately show ?thesis
    using s by (simp add: Store-def)
  qed
qed
moreover
have ∀ l. typeof (vals ?s-upd l) ≤ ltype l
proof
  fix k show typeof (vals ?s-upd k) ≤ ltype k
  proof (cases k=l)
case True
  with type-conform show \?thesis
  by simp
next
  case False
  hence vals \?s-upd k = vals (Rep-Store s) k
  by simp
  with s show \?thesis
  by (simp add: Store-def)
qed
qed
ultimately show \?thesis
by (simp add: Store-def)
qed

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 \leq 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])
thus 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 \neq 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 \leq 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 \leq ltype l
shows $s(l:=x)@\alpha l = x$

proof

- from alive-l alive-x widen-x-l

  have Rep-Store
    ($\text{Abs-Store} (\text{Rep-Store} s[\text{vals} := (\text{vals} (\text{Rep-Store} s))(l := x)]))$
    = $\text{Rep-Store} s[\text{vals} := (\text{vals} (\text{Rep-Store} s))(l := x)]$
    by (rule vals-update-in-Store [THEN Abs-Store-inverse])

  hence $\text{Abs-Store} (\text{Rep-Store} s[\text{vals} := (\text{vals} (\text{Rep-Store} s))(l := x)])(@\alpha l = x)$
  by (simp add: access-def)

with alive-l alive-x widen-x-l

show thesis
by (simp add: update-def)

qed

Store4

lemma update-unalive-val [simp,intro]: $\neg$ alive $x$ s $\implies$ s(l:=x) = s
by (simp add: update-def)

lemma update-unalive-loc [simp,intro]: $\neg$ alive (ref l) s $\implies$ s(l:=x) = s
by (simp add: update-def)

lemma update-type-mismatch [simp]: $\neg$ typeof $x$ s $\leq$ ltype l $\implies$ s(l:=x) = s
by (simp add: update-def)

Store9

lemma alive-primitive [simp,intro]: isprimitive (typeof $x$) $\implies$ alive $x$ s
by (cases $x$) (simp-all)

Store10

lemma new-unalive-old-Store [simp]: $\neg$ alive (new s t) s
by (simp-all add: aliveImpl-def new-def)

lemma alloc-new-instance-in-Store:
(Rep-Store s[[newOID := $\lambda$D. if C = D then Suc (newOID (Rep-Store s) C) else newOID (Rep-Store s) D]]) $\in$ Store
(is ?s-alloc $\in$ Store)

proof
- have s: Rep-Store s $\in$ Store
  by (rule Rep-Store)

  hence $\forall$ l. aliveImpl (vals (Rep-Store s l) (Rep-Store s)
  by (simp add: Store-def)

then

- have $\forall$ l. aliveImpl (vals ?s-alloc l) ?s-alloc
  by (auto intro: less-SucI simp add: aliveImpl-def split: Value.splits)

moreover

- have $\forall$ l. $\neg$ aliveImpl (ref l) ?s-alloc $\implies$ vals ?s-alloc l = init (ltype l)

proof (intro allI impI)

  fix l
  assume $\neg$ aliveImpl (ref l) ?s-alloc

  hence $\neg$ aliveImpl (ref l) (Rep-Store s)
  by (simp add: aliveImpl-def split: Value.splits if-split-asm)

with s have vals (Rep-Store s) l = init (ltype l)
by (simp add: Store-def)
thus \text{vals } ?s\text{-alloc } l = \text{init } (\text{ltype } l)
by simp
qed

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

\textbf{lemma alloc-new-array-in-Store:}
(Rep-Store \text{s } \|newAID := 
\lambda S. \text{if } T = S
\text{then } \text{Suc } (\text{newAID } (\text{Rep-Store } s) T)
\text{else } \text{newAID } (\text{Rep-Store } s) S,
\text{vals } := (\text{vals } (\text{Rep-Store } s))
(\text{arrLenLoc } T
(\text{newAID } (\text{Rep-Store } s) T) :=
\text{intgV } (\text{int } n))))) \in \text{Store}
(is ?s\text{-alloc } \in \text{Store})
proof –
have \text{s } : \text{Rep-Store } s \in \text{Store}
by (rule Rep-Store)
have \forall l. \text{aliveImpl } (\text{vals } ?s\text{-alloc } l) ?s\text{-alloc}
proof
fix l show \text{aliveImpl } (\text{vals } ?s\text{-alloc } l) ?s\text{-alloc}
proof (cases l = \text{arrLenLoc } T (\text{newAID } (\text{Rep-Store } s) T))
\text{case True}
thus ?thesis
by (simp add: aliveImpl-def split: Value.splits)
next
\text{case False}
from \text{s have } \forall l. \text{aliveImpl } (\text{vals } (\text{Rep-Store } s) l) (\text{Rep-Store } s)
by (simp add: Store-def)
with \text{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 } (\text{ref } l) ?s\text{-alloc } \rightarrow \text{vals } ?s\text{-alloc } l = \text{init } (\text{ltype } l)
proof (intro allI implI)
fix l
assume unalive: \neg \text{aliveImpl } (\text{ref } l) ?s\text{-alloc}
show \text{vals } ?s\text{-alloc } l = \text{init } (\text{ltype } l)
proof (cases l = \text{arrLenLoc } T (\text{newAID } (\text{Rep-Store } s) T))
\text{case True}
with \text{unalive show } ?\text{thesis by } (\text{simp add: aliveImpl-def})
next
\text{case False}
from \text{unalive}
have \neg \text{aliveImpl } (\text{ref } l) (\text{Rep-Store } s)
by (simp add: aliveImpl-def split: Value.splits if-split-asm)
with $s$ have $\text{vals} \ (\text{Rep-Store} \ s) \ l = \text{init} \ (\text{lt} \ l)$
by (simp add: Store-def)
with False show $\text{thesis}$
by simp
qed

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

lemma new-alive-alloc [simp,intro]: $\text{alive} \ (\text{new} \ s \ t) \ (s(t))$
proof (cases $t$)
case new-instance thus $\text{thesis}$
  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 $\text{thesis}$
  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:
$(\forall x. \ \text{typeof} \ x = \text{CClassT} \ \text{typeId} \rightarrow P \ x) = (\forall a. P \ (\text{objV} \ \text{typeId} \ a))$
(is $(\forall x. \ ?A \ x) = \ ?B$)
proof
assume $\forall x. \ ?A \ x$ thus $\ ?B$
  by simp
next
assume $B$: $\ ?B$ show $\forall x. \ ?A \ x$
proof
fix $x$ from $B$ show $\ ?A \ x$
  by (cases $x$) auto
qed
qed

lemma value-array-inhabitants:
$(\forall x. \ \text{typeof} \ x = \text{ArrT} \ \text{typeId} \rightarrow P \ x) = (\forall a. P \ (\text{arrV} \ \text{typeId} \ a))$
(is $(\forall x. \ ?A \ x) = \ ?B$)
proof
assume $\forall x. \ ?A \ x$ thus $\ ?B$
  by simp
next
assume $B$: $\ ?B$ show $\forall x. \ ?A \ x$
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**:
\[
(\forall a. (a < Suc n) = (a < Suc m)) = (\forall a. (a < n) = (a < m))
\]

**Proof**

1. Assume \(\forall a. ?A a\) thus \(\forall a. ?B a\)
   - By `fastforce`

2. Assume \(B: \forall a. ?B a\)
   - Show \(\forall a. ?A a\)
     - Proof
       - Fix \(a\) from \(B\)
       - Show \(\forall a. ?A a\)
         - By `(cases a)` `simp-all`

**Qed**

**Qed**

**Lemma all-le-eq-imp-eq**:
\[
(\forall c :: nat. (\forall a. (a < d) = (a < c)) \rightarrow (d = c))
\]

**Proof**

1. Case \(0\) thus \(?case\) by `fastforce`

2. Case \((Suc n c)\)
   - Thus \(?case\)
     - By `(cases c)` `auto simp add: le-Suc-eq`

**Qed**

**Lemma all-le-eq**:
\[
(\forall a :: nat. (a < d) = (a < c)) = (d = c)
\]

Using `all-le-eq-imp-eq` by `auto`

**Store11**

**Lemma typeof-new**: `typeof new s t = typeofNew t`
- By `(cases t)` `(simp-all add: new-def typeofNew-def)`

**Store12**

**Lemma new-eq**:
\[
(new s1 t = new s2 t) =
(\forall x. typeof x = typeofNew t \rightarrow alive x s1 = alive x s2)
\]

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{\text{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

Store7

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: if-split-asm
    simp add: alive-def new-def alloc-def aliveImpl-def
    alloc-new-instance-in-Store [THEN Abs-Store-inverse])
next
  case (new-array T l)
  with alive-alloc show ?thesis
  by (cases x) (auto split: if-split-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 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(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@@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: alloc-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 $s(t) @@ l = \text{init (ltype l)}$
  by \ (\text{simp add: access-new})
moreover
from ref-new have $s @@ l = \text{init (ltype l)}$
  by simp
ultimately
show $?thesis$ by simp
qed

next
  case False
  hence $s(t) @@ l = \text{init (ltype l)}$
  by simp
moreover
  from False have $\neg \text{alive (ref l) s}$
  by \ (auto simp add: alive-alloc-propagation)
  hence $s @@ l = \text{init (ltype l)}$
  by simp
ultimately show $?thesis$ by simp
qed

Store13

lemma Store-eqI:
  assumes eq-alive: $\forall x. \text{alive x s1} = \text{alive x s2}$
  assumes eq-access: $\forall l. s1 @@ l = s2 @@ l$
  shows $s1 = s2$
proof (cases $s1 = s2$)
  case False thus $?thesis$.
next
  case True note neq-s1-s2 = this
  show $?thesis$
    proof (cases newOID (Rep-Store s1) = newOID (Rep-Store s2))
      case False
      have $\exists C. \text{newOID (Rep-Store s1)} C \neq \text{newOID (Rep-Store s2)} C$
        proof (rule ccontr)
          assume $\neg (\exists C. \text{newOID (Rep-Store s1)} C \neq \text{newOID (Rep-Store s2)} C)$
          then have $\text{newOID (Rep-Store s1)} = \text{newOID (Rep-Store s2)}$
            by (blast intro: ext)
          with False show False ..
        qed
      with eq-alive obtain C
        where $\text{newOID (Rep-Store s1)} C \neq \text{newOID (Rep-Store s2)} C$
          $\forall a. \text{alive (objV C a) s1} = \text{alive (objV C a) s2}$
          by auto
        then show $?thesis$
          by (simp add: all-le-eq alive-def aliveImpl-def)
    next
      case True note eq-newOID = this
      show $?thesis$
        proof (cases newAID (Rep-Store s1) = newAID (Rep-Store s2))
          case False
          have $\exists T. \text{newAID (Rep-Store s1)} T \neq \text{newAID (Rep-Store s2)} T$
            proof (rule ccontr)
              assume $\neg (\exists T. \text{newAID (Rep-Store s1)} T \neq \text{newAID (Rep-Store s2)} T)$
              then have $\text{newAID (Rep-Store s1)} = \text{newAID (Rep-Store s2)}$
                by (simp add: all-le-eq
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: ref l ≠ new s t
- **assumes** neq-x-new: x ≠ new s t
- **shows** s(t)(l:=x) = s(l:=x)(t)
- **proof** (rule Store-eqI [rule-format!])
  - **fix** y
  - **show** alive y (s(t)(l:=x)) = alive y (s(l:=x)(t))
  - **proof**
    - **have** alive y (s(t)(l:=x)) = alive y (s(t))
      - **by** (rule alive-update-invariant)
    - **also have** ... = (alive y s ∨ (y = new s t))
      - **by** (rule alive-alloc-exhaust)
also have \( \ldots = (\text{alive } y (s(|:x|)) \lor y = \text{new } s \ t) \)
by (simp only: alive-update-invariant)
also have \( \ldots = (\text{alive } y (s(|:x|)) \lor y = \text{new } (s(|:x|)) \ t) \)
proof –
have new s t = new (s(|:x|)) \ t
by simp
thus \( \text{thesis} \) by simp
qed
also have \( \ldots = \text{alive } y (s(|:x|))(t) \)
by (simp add: alive-alloc-exhaust)
finally show \( \text{thesis} \).
qed
next
fix \( k \)
show \( s(t)(l := x)@@k = s(l := x)(t)@@k \)
proof (cases \( l=k \))
case False note neq-l-k = this
show \( \text{thesis} \)
proof (cases isNewArr \( t \) \( \rightarrow k \neq \text{arr-len (new } s \ t) \))
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 \( \text{thesis} \).
next
case False
then obtain \( T \ n \) where
\( t; t=\text{new-array } T \ n \) and \( k; k=\text{arr-len (new } s \text{ (new-array } T \ n) \) by (cases \( t \) auto)
from \( k \) have \( k'; k=\text{arr-len (new } (s(|:x|)) \text{ (new-array } T \ n) \)
by simp
from neq-l-k
have \( s(t)(l := x)@@k = s(t')@@k \) by simp
also from \( t \ k \)
have \( \ldots = \text{intgV (int } n) \)
by simp
also from \( t \ k' \)
have \( \ldots = s(l := x)(t)@@k \)
by (simp del: new-update)
finally show \( \text{thesis} \).
qed
next
case True note eq-l-k = this
have lemma-3-1:
ref \( l \neq \text{new } s \ t \rightarrow \text{alive (ref } l) (s(t)) = \text{alive (ref } l) \ s \)
by (simp add: alive-alloc-exhaust)
have lemma-3-2:
x \neq \text{new } s \ t \rightarrow \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 ref l \neq new (s(l:=x)) t 
    by simp
  hence isNewArr t \rightarrow l \neq arr-len (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 (ref l) s)
      case True
      note alive-l = this
      show thesis 
        proof (cases typeof x \leq ltype l)
          case True
          with alive-l alive-x 
          have s(l:=x)@@l = x 
            by (rule update-access-same)
          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-access-same)
            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 \neg 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 \ldots = s(l:=x)@@l 
        by simp
      also have \ldots = 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 \neg 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 neg-l-new
  have isNewArr \( t \longrightarrow l \neq \text{arr-len}\ (\text{new } s\ t) \)
    by (cases \( t \)) auto
  thus \( \text{?thesis} \)
    by (simp)
qed
also from not-alive-x have ... = s(l := x) @@ l
  by (simp)
also have ... = s(l := x)(t) @@ l
  by (simp add: lemma-3-3)
finally show \( \text{?thesis} \) by (simp add: eq-l-k)
qed
qed

end

13 Store Properties

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 \( \text{reachS} \) 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.

inductive \( \text{reach} :: \text{Store} \Rightarrow \text{Location} \Rightarrow \text{Value} \Rightarrow \text{bool} \)
  where
  Immediate: \( \text{ref } l \neq \text{nullV} \Longrightarrow s \vdash l \text{ reachable-from } (\text{ref } l) \)
  | Indirect: \([s \vdash l \text{ reachable-from } (s @@ k); \text{ref } k \neq \text{nullV}]
  \Longrightarrow s \vdash l \text{ reachable-from } (\text{ref } k) \)

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.
lemma unreachable-Null:
assumes reach: \( s \vdash l \text{ reachable-from } x \)
shows \( x \neq \text{nullV} \)
using reach by (induct) auto

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

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

lemma reachObjLoc [simp,intro]:
\( C = \text{cls cf} \implies s \vdash \text{objLoc cf a reachable-from objV C a} \)
by (rule reach.Indirect [OF reach-loc, simplified])

lemma reachArrLoc [simp,intro]: \( s \vdash \text{arrLoc T a i reachable-from arrV T a} \)
by (rule reach.Indirect [of arrLoc T a i, simplified])

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

lemma unreachable-static [simp]: \( \neg s \vdash \text{staticLoc f reachable-from x} \)
proof -
{ 
  fix \( y \) assume \( s \vdash y \text{ reachable-from x } y=\text{staticLoc f} \)
  then have False
  by (induct auto)
}
thus \( \neg \\text{thesis} \)
by auto
qed

lemma unreachable-staticE [elim]: \( s \vdash \text{staticLoc f reachable-from x} \implies P \)
by (simp add: unreachable-static)

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

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

Lemma 3.2 (i)

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

Lemma 3.2 (ii)

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

Lemma 3.2 (iv)

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

lemma \text{reachable-isRef}:
assumes \( \text{reach}: s \vdash \text{reachable-from} \ x \)
shows \( \text{isRefV} \ x \)
using \( \text{reach} \)
proof (induct)
case (Immediate \( l \))
show \( \text{isRefV} \ (\text{ref} \ l) \)
  by \( \text{(cases} \ l \text{)} \) simp-all
next
case (Indirect \( l \) \( k \))
show \( \text{isRefV} \ (\text{ref} \ k) \)
  by \( \text{(cases} \ k \text{)} \) simp-all
qed

lemma \text{val-ArrLen-IntgT}: \text{isArrLenLoc} \ l \Rightarrow \text{typeof} \ (s \ @@ l) = \text{IntgT}
proof
  assume \( \text{isArrLen}: \text{isArrLenLoc} \ l \)
  have \( T: \text{typeof} \ (s \ @@ l) \leq \text{ltype} \ l \)
    by \( \text{(simp)} \)
  also from \( \text{isArrLen} \) have \( I: \text{ltype} \ l = \text{IntgT} \)
    by \( \text{(cases} \ l \text{)} \) simp-all
  finally show \( \text{?thesis} \)
    by \( \text{(auto elim: rtranclE simp add: le-Javatype-def subtype-defs)} \)
qed

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

Lemma 3.2 (v)

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

Lemma 3.2 (vi)

lemma reach6: \( \text{isprimitive} (\text{typeof} x) \implies \neg s \vdash l \text{ reachable-from} x \)
proof
  assume prim: \( \text{isprimitive} (\text{typeof} x) \)
assume $s \vdash l$ reachable-from $x$

hence isRefV $x$

by (rule reachable-isRef)

with prim show False

by (cases $x$) simp-all

qed

Lemma 3.2 (iii)

lemma reach3:

assumes $k-y$: $\neg s \vdash k$ reachable-from $y$

assumes $k-x$: $\neg s \vdash k$ reachable-from $x$

shows $\neg s \langle l := y \rangle \vdash k$ reachable-from $x$

proof

assume $s \langle l := y \rangle \vdash k$ reachable-from $x$

from this $k-y$ $k-x$

show False

proof (induct)
  case (Immediate $l$)
  have $\neg s \vdash l$ reachable-from ref $l$ and ref $l \neq$ nullV by fact+

  thus False

  by (iprover intro: reach.intros)

next
  case (Indirect $m$ $k$)
  have $k$-not-Null: ref $k \neq$ nullV by fact

  have not-m-y: $\neg s \vdash m$ reachable-from $y$ by fact

  have not-m-k: $\neg s \vdash m$ reachable-from ref $k$ by fact

  have hyp: $\neg \vdash s \vdash m$ reachable-from $y$; $\neg s \vdash m$ reachable-from $(s \langle l := y \rangle@@k)$

  $\Rightarrow$ False by fact

  have m-upd-k: $s \langle l := y \rangle \vdash m$ reachable-from $(s \langle l := y \rangle@@k)$ by fact

  show False

  proof (cases $l$=$k$)

  case False

  then have $s \langle l := y \rangle@@k = s@@k$ by simp

  moreover

  from not-m-k $k$-not-Null have $\neg s \vdash m$ reachable-from $(s@@k)$

  by (iprover intro: reach.intros)

  ultimately show False

  using not-m-y hyp by simp

next
  case True note eq-l-k = this

  show $?thesis$

  proof (cases alive (ref $l$) $s \land$ alive $y$ $s \land$ typeof $y \leq$ ltype $l$)

  case True

  with eq-l-k have $s \langle l := y \rangle@@k = y$

  by simp

  with not-m-y hyp show False by simp

next
  case False

  hence $s \langle l := y \rangle = s$

  by auto

  moreover

  from not-m-k $k$-not-Null have $\neg s \vdash m$ reachable-from $(s@@k)$

  by (iprover intro: reach.intros)

  ultimately show False
Lemma 3.2 (vii).

**Lemma unreachable-from-init** [simp, intro]: \( \neg s \vdash l \text{ reachable-from } \langle \text{init } T \rangle \)

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

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

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

**Proof induction**
- **case** \( \text{Immediate } l \)
  - **show** \( \text{ ref } l = \text{ new } s \, t \) by fact
- **next**
  - **case** \( \text{Indirect } l \, k \)
  - **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** False by simp
  - **thus** ?case ..

**Lemma alive-reach-alive:**
- **assumes** alive-x: \( \text{ alive } x \, s \)
- **assumes** reach-l: \( s \vdash l \text{ reachable-from } x \)
shows alive (ref l) s
using reach-l alive-x
proof (induct)
case (Immediate l)
show ?case by fact
next
case (Indirect l k)
have hyp: alive (s@@k) s \implies alive (ref l) s by fact
moreover have alive (s@@k) s by simp
ultimately
show alive (ref l) s
by iprover
qed

Lemma 3.2 (ix)

lemma reach9:
assumes reach-impl-access-eq: \\forall l. s1\vdash l reachable-from x \implies (s1@@l = s2@@l)
shows s1\vdash l reachable-from x = s2\vdash l reachable-from x
proof
assume s1\vdash l reachable-from x
from this reach-impl-access-eq
show s2\vdash l reachable-from x
proof (induct)
case (Immediate l)
show s2\vdash l reachable-from ref l
by (rule reach.intros) (rule Immediate.hyps)
next
case (Indirect l k)
have hyp: \\forall l. s1\vdash l reachable-from (s1@@k) \implies s1@@l = s2@@l
\implies s2\vdash l reachable-from (s1@@k) by fact
have k-not-Null: ref k \neq nullV by fact
have reach-impl-access-eq:
\\forall l. s1\vdash l reachable-from ref k \implies s1@@l = s2@@l by fact
have s1\vdash l reachable-from (s1@@k) by fact
with k-not-Null
have s1@@k = s2@@k
by (iprover intro: reach-impl-access-eq [rule-formats] reach.intros)
moreover from reach-impl-access-eq k-not-Null
have \\forall l. s1\vdash l reachable-from (s1@@k) \implies s1@@l = s2@@l
by (iprover intro: reach.intros)
then have s2\vdash l reachable-from (s1@@k)
by (rule hyp)
ultimately have s2\vdash l reachable-from (s2@@k)
by simp
thus s2\vdash l reachable-from ref k
by (rule reach.intros) (rule Indirect.hyps)
qed
next
assume s2\vdash l reachable-from x
from this reach-impl-access-eq
show s1\vdash l reachable-from x
proof (induct)
case (Immediate l)
show s1\vdash l reachable-from ref l
13.2 Reachability of a Reference from a Reference

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

\textbf{definition} \texttt{rreach} :: \textit{Store} ⇒ \textit{Value} ⇒ \textit{Value} ⇒ \textit{bool} where \texttt{s \triangleright Ref y reachable-from x = (\exists l. s \triangleright l reachable-from x \land y = \text{ref} l)}

13.3 Disjointness of Reachable Locations

The predicate \texttt{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.

\textbf{definition} \texttt{disj} :: \textit{Value} ⇒ \textit{Value} ⇒ \textit{Store} ⇒ \textit{bool} where \texttt{disj x y s = (\forall l. \neg s \triangleright l reachable-from x \lor \neg s \triangleright l reachable-from y)}

\textbf{lemma} \texttt{disjI1} : [\land l. s \triangleright l reachable-from x \implies \neg s \triangleright l reachable-from y] \implies \texttt{disj x y s} by (simp add: disj-def)

\textbf{lemma} \texttt{disjI2} : [\land l. s \triangleright l reachable-from y \implies \neg s \triangleright l reachable-from x] \implies \texttt{disj x y s} by (auto simp add: disj-def)

\textbf{lemma} \texttt{disj-cases} [consumes 1]:
\textbf{assumes} \texttt{disj x y s}
assumes $\bigwedge l. \neg s \vdash l \text{ reachable-from } x \implies P$
assumes $\bigwedge l. \neg s \vdash l \text{ reachable-from } y \implies P$
shows $P$
using assms by (auto simp add: disj-def)

Lemma 3.3 (i) in [PH97]

lemma disj1: $[[\text{disj } x y s; \neg s \vdash l \text{ reachable-from } x; \neg s \vdash l \text{ reachable-from } y]] 
\implies \text{ disj } x y (s(l:=z))$
by (auto simp add: disj-def)

Lemma 3.3 (ii)

lemma disj2:
assumes disj-x-y: $\text{ disj } x y s$
assumes disj-x-z: $\text{ disj } x z s$
assumes unreach-l-x: $\neg s \vdash l \text{ reachable-from } x$
shows $\text{ disj } x y (s(l:=z))$
proof (rule disjI1)
fix $k$
assume reach-k-x: $s(l := z)\vdash k \text{ reachable-from } x$
show $\neg s(l := z)\vdash k \text{ reachable-from } y$
proof
  assume reach-l-new: $s(l := z)\vdash l \text{ reachable-from } new s t$
  have unalive-new: $\neg \text{ alive } (new s t) s$
    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
qed

Lemma 3.3 (iii)

lemma disj3: assumes alive-x-s: $\text{ alive } x s$
shows $\text{ disj } x (new s t) (s(t))$
proof (rule disjI1, simp only: reach-alloc)
fix $l$
assume reach-l-x: $s \vdash l \text{ reachable-from } x$
show $\neg s \vdash l \text{ reachable-from } new s t$
proof
  assume reach-l-new: $s \vdash l \text{ reachable-from } new s t$
have unalive-new: $\neg \text{ alive } (new s t) s$
  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
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** \( xeq \):: \( \text{Value} \Rightarrow \text{Store} \Rightarrow \text{Store} \Rightarrow \text{bool} \) where

\[
xeq x s t = (\text{alive } x s = \text{alive } x t \land \forall l. s \vdash l \text{ reachable-from } x \rightarrow s\@l = t\@l)
\]

**Abbreviation** \( xeq\text{-syntax} :: \text{Store} \Rightarrow \text{Value} \Rightarrow \text{Store} \Rightarrow \text{bool} \)

\(-/ (\equiv[])// [-900,0,900] 900\)

where \( s \equiv[x] t == xeq x s t \)

**Lemma** \( xeq1 :: [\text{alive } x s = \text{alive } x t; \]

\( \land l. s \vdash l \text{ reachable-from } x \Rightarrow s\@l = t\@l \)

\( \Rightarrow s \equiv[x] t \)

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 \( \text{alive } x s = \text{alive } x t \) by \( (simp \ add: xeq-def) \)

moreover

from \( s-t \) have \( \forall l. s \vdash l \text{ reachable-from } x \rightarrow s\@l = t\@l \)

by \( (simp \ add: xeq-def) \)

with \( \text{reach9} \) [OF this]

have \( \forall l. t \vdash l \text{ reachable-from } x \rightarrow t\@l = s\@l \)

by \( simp \)

ultimately show \( \text{thesis} \)

by \( (simp \ add: xeq-def) \)

qed
lemma \( \text{xeq1-sym}: s \equiv [x] t = t \equiv [x] s \)
by (auto intro: xeq1-sym)

Lemma 3.4 (i)

lemma \( \text{xeq1-trans} \): \[ \text{trans} : \]
assumes \( s \equiv [x] t \)
assumes \( t \equiv [x] r \)
shows \( s \equiv [x] r \)

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

Lemma 3.4 (ii)

lemma \( \text{xeq2} \):
assumes \( \text{xeq}: \forall x. s \equiv [x] t \)
assumes \( \text{static-eq}: \forall f. s @ @ (\text{staticLoc } f) = t @ @ (\text{staticLoc } f) \)
shows \( s = t \)

proof (rule Store-eqI)
from \( \text{xeq} \)
show \( \forall x. \text{alive } x \ s = \text{alive } x \ t \)
by (simp add: xeq-def)
next
show \( \forall l. s @ @ l = t @ @ l \)
proof
fix \( l \)
show \( s @ @ l = t @ @ l \)
proof (cases \( l \))
case \( \text{objLoc } cf \ a \)
have \( l = \text{objLoc } cf \ a \) by fact

hence $s \vdash l$ reachable-from $(objV (cls cf) a)$
  by simp
with $xeq$ show $\text{thesis}$
  by (simp add: $xeq$-def)

next
  case (staticLoc $f$)
  have $l = \text{staticLoc } f$ by fact
  with $static$-eq show $\text{thesis}$
  by (simp add: $xeq$-def)

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

qed

Lemma 3.4 (iii)
lemma $xeq$-3:
  assumes $unreach$-$l$: $\neg s \vdash l$ reachable-from $x$
  shows $s \equiv [x] s'(l := y)$
proof (rule $xeq$-I)
  show $\text{alive } x\ s = \text{alive } x\ (s(l := y))$
    by simp

next
  fix $k$
  assume $reach$-$k$: $s \vdash k$ reachable-from $x$
  with $unreach$-$l$ have $l \neq k$ by auto
  then show $s @@ k = s(l := y) @@ k$
    by simp

qed

Lemma 3.4 (iv)
lemma $xeq$-4:
  assumes $not$-$new$: $x \neq \text{new } s\ t$
  shows $s \equiv [x] s(t)$
proof (rule $xeq$-I)
  from $not$-$new$
  show $\text{alive } x\ s = \text{alive } x\ (s(t))$
    by (simp add: $alive$-alloc-exhaust)

next
  fix $l$
  assume $reach$-$l$: $s \vdash l$ reachable-from $x$
  show $s @@ l = s(t) @@ l$
proof (cases isNewArr $t$ $\rightarrow l \neq \text{arr-len } (\text{new } s\ t)$)

qed
case True
with reach-l show ?thesis
by simp
next
case False
then obtain T n where t: t = new-array T n and
l: l = arr-len (new s t)
by (cases t) auto
hence ref l = new s t
by simp
from this reach-l
have x = new s t
by (rule loc-new-reach)
with not-new show ?thesis ..
qed
qed

Lemma 3.4 (v)

lemma xeq5: s ≡ [x] t ⇒ s≡ [x] t reachable-from x = t≡ [x] t reachable-from x
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 teq [PH97, p. 55].

definition teq:: Javatype ⇒ Store ⇒ Store ⇒ bool where
    teq t s1 s2 = (∀ x. typeof x ≤ t −→ s1≡ [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 lessalive [PH97, p. 55]. 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.

definition lessalive:: Store ⇒ Store ⇒ bool (- ≪ [- 70,71] 70)
    where lessalive s t = ((∀ x. alive x s −→ s≡ [x] t) ∧ (∀ f. s@[staticLoc f] = t@[staticLoc f]))

We define an introduction rule for the new operator.

lemma lessaliveI:
    [∀ x. alive x s −→ s≡ [x] t; ∀ f. s@[staticLoc f] = t@[staticLoc f]]
⇒ s ≪ t
by (simp add: lessalive-def)

It can be shown that lessalive is reflexive, transitive and antisymmetric.

lemma lessalive-refl: s ≪ s
by (simp add: lessalive-def xeq1-refl)

lemma lessalive-trans [trans]:
    assumes s-t: s ≪ t
assumes $t \prec w$: $t \ll w$
shows $s \ll w$
proof (rule lessaliveI)
  fix $x$
  assume alive-x-s: $\text{alive } x \ s$
  with $s-t$ have $s \equiv[x] \ t$
    by (simp add: lessalive-def)
  also
  have $t \equiv[x] \ w$
  proof
    from alive-x-s $s-t$ have $\text{alive } x \ t$ 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 $s@@\text{staticLoc } f = w@@\text{staticLoc } f$
    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 $\text{alive } x \ s = \text{alive } x \ t$ by simp
                    next
                      fix $l$
                      assume reach-s-x: $s \vdash l \text{ reachable-from } x$
                      with unalive-x-s have $x: \ x = \text{ref } l$
                        by (rule ref-reach-unalive)
                      with unalive-x-s have $s@@l = \text{init } (\text{ltype } l)$
                        by simp
                      also from reach-s-x have $t \vdash l \text{ reachable-from } x$
                        by (auto intro: reach.Immediate unreachable-Null)
                      with False $x$ have $t@@l = \text{init } (\text{ltype } l)$
                        by simp
                    qed
                qed
            qed
          qed
        qed
      qed
    qed
  qed

finally show \( s \preceq t \) by simp 
qed
qed
qed
qed

next
from s-t show \( \forall f. \ s \preceq f \) by (simp add: lessalive-def)
qed

This gives us a partial ordering on the store. Thus, the type \( \text{Store} \) can be added to the appropriate type class \( \text{ord} \) which lets us define the \( < \) and \( \leq \) symbols, and to the type class \( \text{order} \) which axiomatizes partial orderings.

instantiation \( \text{Store} :: \text{order} \)
begin

definition \( \text{le-Store-def} \): \( s \leq t \) if \( s \preceq t \)

definition \( \text{less-Store-def} \): \( (s::\text{Store}) < t \) if \( s \leq t \wedge \neg t \leq s \)

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

Lemma 3.5 (i)

instance proof
fix \( s \ t \ w :: \text{Store} \) 
{
  show \( s \leq s \) by (simp add: le-Store-def lessalive-refl)
next
  assume \( s \leq t \ t \leq w \) 
  then show \( s \leq w \) by (unfold le-Store-def) (rule lessalive-trans)
next
  assume \( s \leq t \ t \leq s \) 
  then show \( s = t \) by (unfold le-Store-def) (rule lessalive-antisym)
next
  show \( (s < t) = (s \leq t \wedge \neg t \leq s) \) 
  by (simp add: less-Store-def)
qed

end

Lemma 3.5 (ii)

lemma lessalive2: \( [s \preceq t; \ alive x s] \) if \( alive x t \)
by (simp add: lessalive-def xeq-def)

Lemma 3.5 (iii)

lemma lessalive3:
assumes \( s \preceq t \)
assumes alive: \( \text{alive } x s \lor \neg \text{alive } x t \)
shows \( s \equiv [x] t \)

proof (cases alive \( x s \))

next
case \( \text{True} \)
with \( s \preceq t \) show \(?thesis\)
  by (simp add: lessalive-def)

next
case \( \text{False} \)

note unalive-x-s = this

with \( \text{alive} \) have unalive-x-t: \( \neg \text{alive } x t \)
  by simp

show \(?thesis\)
proof (rule xeqI)

  from \( \neg \text{alive} \) show \( \text{alive } x s = \text{alive } x t \)
  by simp

next
fixed \( l \)
assume reach-s-x: \( s \vdash l \text{ reachable-from } x \)

with \( \neg \text{alive} \) have \( x = \text{ref } l \)
  by (rule ref-reach-unalive)

with \( \neg \text{alive} \) have \( s \equiv l = \text{init } (\text{ltype } l) \)
  by simp

also from \( \text{reachable-from } x \) have \( t \vdash l \text{ unreachable-Null} \)
  by (auto intro: reach.Unreachable-Null)

with \( \neg \text{alive} \) have \( t \equiv l = \text{init } (\text{ltype } l) \)
  by simp

finally show \( s \equiv t \equiv l \equiv l \)
  by simp

qed

proof

lemmas [simp,intro]:

assumes \( s \preceq t \)
assumes alive-l: \( \neg \text{alive } (\text{ref } l) t \)
shows \( s \preceq t(l:=x) \)

proof
  from \( \neg \text{alive} \) have \( t(l:=x) = t \)
  by simp

next
  from \( \neg \text{alive} \) show \( x \neq \text{new } s t \)
  by auto

thus \(?thesis\)
  by (rule xeq4)

qed

Lemma 3.5 (v)

lemma lessalive-alloc [simp,intro]: \( s \preceq s'(t) \)
by (simp add: lessalive-def Xeq4')

13.7 Reachability of Types from Types

The predicate \( \text{treach} \) 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 \([PH97, \text{p. 106}]\) 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 \( \text{not-treach-ref-impl-not-reach} \) below.

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

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

\[
\begin{align*}
\text{lemma } \text{treach-ref-l'} \ [\text{simp},\text{intro}]: & \quad \text{assumes } \text{not-Null: } \text{ref } l \neq \text{nullV} \\
& \quad \text{shows } \text{treach } (\text{typeof } (\text{ref } l)) (\text{typeof } (s@@l))
\end{align*}
\]
proof
  from not-Null have treach (typeof (ref l)) (ltype l) by (rule treach-ref-l)
  also have typeof (s@@l) ≤ ltype l
    by simp
  hence treach (ltype l) (typeof (s@@l))
    by (rule treach.intros)
  finally show ?thesis .
qed

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
qed

Lemma 4.6 in [PH97, p. 107].

lemma treach1:
assumes x-t: typeof x ≤ T
assumes not-treach: ¬ treach T (typeof (ref l))
sows ¬ 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
  a1 ( s@@(x..CounterImpl\'value) )
else undefined)
end
References


