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.
15 The Universal Specification
1 Introduction

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

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

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

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

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

A large number of the rules and axioms of the Hoare logic is related to the structure of the program part that is currently being examined. Besides these, the logic also contains rules that manipulate the pre- or postcondition of the examined subgoal without affecting the current program part selection. A prominent member of this kind of rules is the rule of consequence:

$$\frac{\text{PP} \Rightarrow P \quad A \triangleright \{ P \} \text{pp} \{ Q \} \quad Q \Rightarrow QQ}{A \triangleright \{ \text{PP} \} \text{pp} \{ QQ \}}$$

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

---

1In 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 {*
  add_path "<PATH_TO_THEORIES>/Isabelle";
  add_path "<PATH_TO_THEORIES>/Isabelle_Store";
  add_path "<PATH_TO_THEORIES>/Isa_COUNTER";
  add_path "<PATH_TO_THEORIES>/Isa_COUNTER_Store";
  *}
```

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

### 3 The Example Program

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

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

class CounterImpl implements Counter {
    protected int value;
    public int incr() {
        int dummy;
        res = this.value;
        res = (int) res + 1;
        this.value = res;
    }

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

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

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

4 TypeIds

theory TypeIds imports Main begin

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

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

datatype CTypeId = CounterImpl | UndoCounter
             | Object | Exception | ClassCastException | 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 \textbf{Arraytype} = \text{BoolAT} | \text{IntgAT} | \text{ShortAT} | \text{ByteAT} \\
| \text{CClassAT CTypeId} | \text{AClassAT ATypeId} \\
| \text{InterfaceAT ITypeId}

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

We need a function that widens \textit{Arraytype} to \textit{Javatype}.

definition \textbf{at2jt} :: Arraytype \Rightarrow Javatype

where

\text{at2jt at} = \text{(case at of \\
BooAT \Rightarrow \text{BoolT} \\
| IntgAT \Rightarrow \text{IntgT} \\
| ShortAT \Rightarrow \text{ShortT} \\
| ByteAT \Rightarrow \text{ByteT} \\
| CClassAT CTypeId \Rightarrow \text{CClassT CTypeId} \\
| AClassAT ATypeId \Rightarrow \text{AClassT ATypeId} \\
| InterfaceAT ITypeId \Rightarrow \text{InterfaceT ITypeId})}

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

primrec \textbf{isprimitive}:: Javatype \Rightarrow bool

where

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

primrec \textbf{isclass}:: Javatype \Rightarrow bool

where

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

end

6 The Direct Subtype Relation of Java Types

theory DirectSubtypes
imports ../Isabelle/JavaType
begin

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

We have the following type hierarchy:

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

We have to specify the direct subtype relation between

- each “leaf” class or interface and its subtype 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
(\text{CClassT UndoCounter}, \text{CClassT CounterImpl}) \in \text{direct-subtype}
(\text{CClassT CounterImpl}, \text{InterfaceT Counter}) \in \text{direct-subtype}
(\text{CClassT NullPointerException}, \text{CClassT Exception}) \in \text{direct-subtype}
(\text{CClassT ClassCastException}, \text{CClassT Exception}) \in \text{direct-subtype}

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

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

(\text{ArrT (CClassAT Object)}, \text{CClassT Object}) \in \text{direct-subtype}

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

by (simp-all add: direct-subtype-def)

7 Widening the Direct Subtype Relation

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

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

7.1 Auxiliary lemmas

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

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

lemma acyclic-imp-antisym-rtrancl: acyclic r ⇒ antisym (r∗)
proof (clarsimp simp only: acyclic-def antisym-def)
fix x y
assume acyclic: ∀ x. (x, x) ∉ r+
assume x-y: (x, y) ∈ r∗
assume y-x: (y, x) ∈ r∗
show x=y
proof (cases x=y)
case True thus ?thesis .
next
case False
from False x-y have (x, y) ∈ r+
by (rule distinct-rtrancl-into-trancl)
also
from False y-x have (y, x) ∈ r+
by (fastforce intro: distinct-rtrancl-into-trancl)
finally have (x,x) ∈ r+.
with acyclic show ?thesis by simp
qed

lemma acyclic-trancl-rtrancl:
assumes acyclic: acyclic r
shows (x,y) ∈ r+ = ((x,y) ∈ r∗ ∧ x≠y)
proof
assume x-y-trancl: (x,y) ∈ r+
show (x,y) ∈ r∗ ∧ x≠y
proof
from x-y-trancl show (x,y) ∈ r+..
next
from x-y-trancl acyclic show x≠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 (- ≺ 1 - [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 \( \bigwedge c. b ≤ c \Rightarrow a ≤ c \)
  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:: order
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::Javatype) ≤ b ⇒ a ≤ 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 =
  rtac {thm widen-asm} THEN'
  simp-tac (put-simpset ss ctxt addsimps {thms le-Javatype-def}) THEN'
  Method.insert-tac {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 ≤ CClassT Object
  by (cases C, simp-all)

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

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

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

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

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

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

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

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

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

8 Attributes

theory Attributes
imports ../Isabelle/Subtype
begin

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

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

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

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

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

**definition** dtype:: AttId ⇒ Javatype where
dtype f = (case f of
  CounterImpl’value ⇒ CClassT CounterImpl
  UndoCounter’save ⇒ CClassT UndoCounter
  Dummy’dummy ⇒ AClassT Dummy
  Counter’dummy ⇒ InterfaceT Counter)

**lemma** dtype-simps [simp]:
dtype CounterImpl’value = CClassT CounterImpl
dtype UndoCounter’save = CClassT UndoCounter
dtype Dummy’dummy = AClassT Dummy
dtype Counter’dummy = InterfaceT Counter
by (simp-all add: dtype-def dtype-def dtype-def)

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

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

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

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

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

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

**definition** rtype:: AttId ⇒ Javatype where
rtype f = (case f of
  CounterImpl’value ⇒ IntgT
With the datatype \textit{CAttId} we describe the possible locations in memory for instance fields. We rule out the impossible combinations of class names and field names. For example, a \textit{CounterImpl} cannot have a \textit{save} field. A store model which provides locations for all possible combinations of the Cartesian product of class name and field name works out fine as well, because we cannot express modification of such “wrong” locations in a Java program. So we can only prove useful properties about reasonable combinations. The only drawback in such a model is that we cannot prove a property like \textit{not-treach-ref-impl-not-reach} in theory \textit{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 \textit{CAttId}.

While \textit{AttId} describes which fields are declared in which classes and interfaces, \textit{CAttId} describes which objects of which classes may contain which fields at run-time. Thus, \textit{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.

**definition** \textit{catt}:: \textit{CTYPEId} \Rightarrow \textit{AttId} \Rightarrow \textit{CAttId} \textbf{where}

\begin{align*}
\text{catt \textit{C} \textit{f} =} & \\
\text{\hspace{0.5em} (case \textit{C} of} & \\
\text{\hspace{1em} \textit{CounterImpl} \Rightarrow (case \textit{f} of} & \\
\text{\hspace{2em} \textit{CounterImpl}'value \Rightarrow \textit{CounterImpl}'value} & \\
\text{\hspace{2em} \textit{Dummy}'dummy \Rightarrow \text{undefined}} & \\
\text{\hspace{2em} \textit{Counter}'dummy \Rightarrow \textit{Counter}'dummy}) & \\
\text{\hspace{1em} \textit{UndoCounter} \Rightarrow (case \textit{f} of} & \\
\text{\hspace{2em} \textit{CounterImpl}'value \Rightarrow \textit{UndoCounter}'CounterImpl'value} & \\
\text{\hspace{2em} \textit{UndoCounter}'save \Rightarrow \text{undefined}} & \\
\text{\hspace{2em} \textit{Dummy}'dummy \Rightarrow \text{undefined}} & \\
\text{\hspace{2em} \textit{Counter}'dummy \Rightarrow \textit{UndoCounter}'Counter'dummy}) & \\
\text{\hspace{1em} \textit{Object} \Rightarrow \text{undefined}} & \\
\text{\hspace{1em} \textit{Exception} \Rightarrow \text{undefined}} & \\
\text{\hspace{1em} \textit{ClassCastException} \Rightarrow \text{undefined}} & \\
\text{\hspace{1em} \textit{NullPointerException} \Rightarrow \text{undefined}} & \\
\text{\hspace{1em} \text{undefined})} & \\
\end{align*}
lemmas 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
  )

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

Selection of the field name.

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

lemmas 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-Store/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]:
\[ \text{CClassT } c \leq \text{dtype } f \implies \text{cls (catt } c f ) = c \]
apply (case-tac c)
apply (case-tac [!] f)
apply simp-all
— solves all goals where CClassT c ≤ dtype f
apply (fastforce elim: subtype-wrong-elims simp add: subtype-defs)+
— solves all the rest where ¬ CClassT c ≤ dtype f can be derived

**done**

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

**done**

The following lemmas are just a demonstration of simplification.

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

**lemma** widen-cls-dtype-att [simp,intro]:
\[ (\text{CClassT } (\text{cls } cf ) \leq \text{dtype } (\text{att } cf )) \]
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** \( \text{ObjectId} = \text{nat} \)

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** \( \text{Value} = \text{boolV} \ \text{JavaBoolean} \\
| \text{intgV} \ \text{JavaInt} \\
| \text{shortV} \ \text{JavaShort} \\
| \text{byteV} \ \text{JavaByte} \\
| \text{objV} \ C\text{TypeId} \ \text{ObjectId} \ — \text{typed object reference} \\
| \text{arrV} \ \text{Arraytype} \ \text{ObjectId} \ — \text{typed array reference} \\
| \text{nullV} \)

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

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

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

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

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

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

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

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

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

definition isRefV :: Value ⇒ bool where
  isRefV v = (case v of
     boolV b ⇒ False
     | intgV i ⇒ False
     | shortV s ⇒ False
lemma isRefV-simps [simp]:
isRefV (boolV b) = False
isRefV (intgV i) = False
isRefV (shortV s) = False
isRefV (byteV by) = False
isRefV (objV C a) = True
isRefV (arrV T a) = True
isRefV (nullV) = True
by (simp-all add: isRefV-def)

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

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

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

lemma isArrV-simps [simp]:
isArrV (boolV b) = False
isArrV (intgV i) = False
isArrV (shortV s) = False
isArrV (byteV by) = False
isArrV (objV c a) = False
isArrV (arrV T a) = True
10.2 Selector Functions

**isArrV nullV** = False
by (simp-all add: isArrV-def)

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

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

**10.2 Selector Functions**

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

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

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

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

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

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

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

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

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

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

lemma \( oid\text{-simps } [\text{simp}]: \)
\begin{verbatim}
10.3 Determining the Type of a Value

oid (objV C a) = a
by (simp add: oid-def)

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

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

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

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

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

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

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

lemma typeof-simps [simp]:
(\tau (boolV b)) = BoolT
(\tau (intgV i)) = IntgT
\end{verbatim}
$(\tau (\text{shortV } sh)) = \text{ShortT}$

$(\tau (\text{byteV } by)) = \text{ByteT}$

$(\tau (\text{objV } c a)) = \text{CClassT } c$

$(\tau (\text{arrV } t a)) = \text{ArrT } t$

$(\tau (\text{nullV})) = \text{NullT}$

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** $\text{init} :: \text{Javatype} \Rightarrow \text{Value where}$

$\text{init } T = (\text{case } T \text{ of}$

  $\text{BoolT} \Rightarrow \text{boolV False}$

  $| \text{IntgT} \Rightarrow \text{intgV 0}$

  $| \text{ShortT} \Rightarrow \text{shortV 0}$

  $| \text{ByteT} \Rightarrow \text{byteV 0}$

  $| \text{NullT} \Rightarrow \text{nullV}$

  $| \text{ArrT } T \Rightarrow \text{nullV}$

  $| \text{CClassT } C \Rightarrow \text{nullV}$

  $| \text{AClassT } C \Rightarrow \text{nullV}$

  $| \text{InterfaceT } I \Rightarrow \text{nullV})$

**lemma** $\text{init-simps} [\text{simp}]:$

$\text{init } \text{BoolT} = \text{boolV False}$

$\text{init } \text{IntgT} = \text{intgV 0}$

$\text{init } \text{ShortT} = \text{shortV 0}$

$\text{init } \text{ByteT} = \text{byteV 0}$

$\text{init } \text{NullT} = \text{nullV}$

$\text{init } (\text{ArrT } T) = \text{nullV}$

$\text{init } (\text{CClassT } C) = \text{nullV}$

$\text{init } (\text{AClassT } C) = \text{nullV}$

$\text{init } (\text{InterfaceT } I) = \text{nullV}$

by (simp-all add: init-def)

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

**proof** (cases $T$)

assume $c: T = \text{BoolT}$

show $(\tau (\text{init } T)) \leq T$

using $c$ by simp

next

assume $c: T = \text{IntgT}$

show $(\tau (\text{init } T)) \leq T$

using $c$ by simp

next

assume $c: T = \text{ShortT}$

show $(\tau (\text{init } T)) \leq T$

using $c$ by simp

next

assume $c: T = \text{ByteT}$

show $(\tau (\text{init } T)) \leq T$

using $c$ by simp

next
assume $c$: $T = \text{NullT}$  
show $(\tau (\text{init } T)) \leq T$  
using $c$ by simp  

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

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

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

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

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

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

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

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

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

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

```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 \texttt{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)
```
definition isStaticLoc:: Location ⇒ bool where
isStaticLoc l = (case l of
         objLoc cf a ⇒ False
       | staticLoc f  ⇒ True
       | arrLenLoc T a ⇒ False
       | arrLoc T a i ⇒ False)
lemma isStaticLoc-simps $simp$:
isStaticLoc (objLoc cf a) = False
isStaticLoc (staticLoc f)  = True
isStaticLoc (arrLenLoc T a) = False
isStaticLoc (arrLoc T a i) = False
  by (simp-all add: isStaticLoc-def)

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

definition ref:: Location ⇒ Value where
ref l = (case l of
         objLoc cf a ⇒ objV (cls cf) a
       | staticLoc f  ⇒ nullV
       | arrLenLoc T a ⇒ arrV T a
       | arrLoc T a i ⇒ arrV T a)

lemma ref-simps $simp$:
ref (objLoc cf a) = objV (cls cf) a
ref (staticLoc f)  = nullV
ref (arrLenLoc T a) = arrV T a
ref (arrLoc T a i) = arrV T a
  by (simp-all add: ref-def)

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

primrec loc:: Value ⇒ AttId ⇒ Location (..- [80,80] 80)
  where loc (objV c a) f = objLoc (catt c f) a

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

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

primrec arr-len:: Value ⇒ Location
  where arr-len (arrV T a) = arrLenLoc T a

Arrays can be indexed by the function arr-loc.

primrec arr-loc:: Value ⇒ nat ⇒ Location (..- [80,80] 80)
  where arr-loc (arrV T a) i = arrLoc T a i

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

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

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

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

\section{12 Store}

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

\subsection{12.1 New}

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

\textbf{datatype} New = new-instance CTypeId \hspace{1em} — New object, can only be of a concrete class type
| new-array Arraytype nat \hspace{1em} — New array with given size

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

\textbf{definition} isNewArr :: New \Rightarrow bool \hspace{1em} \text{where}
\begin{itemize}
\item isNewArr \(t\) = (case \(t\) of
\item new-instance \(C\) \Rightarrow False
\item new-array \(T\) \(l\) \Rightarrow True)
\end{itemize}

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

The function \(\text{typeofNew}\) yields the type of the newly created created element.

\textbf{definition} typeofNew :: New \Rightarrow Javatype \hspace{1em} \text{where}
\begin{itemize}
\item typeofNew \(n\) = (case \(n\) of
\item new-instance \(C\) \Rightarrow CClassT \(C\)
\item new-array \(T\) \(l\) \Rightarrow ArrT \(T\))
\end{itemize}

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

In our store model, all objects\(^2\) 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 \texttt{newOID}) and arrays (function \texttt{newAID}) as well as a function that maps locations to their contents (function \texttt{vals}).

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

The function \texttt{aliveImpl} determines for a given value whether it is alive in a given store.

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

The store itself is defined as new type. The store ensures and maintains the following properties:

- All stored values are alive; for all locations whose values are not alive, the store yields the location type’s init value; and all stored values are of the correct type (i.e. of the type of the location they are stored in).

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

\begin{verbatim}
typedef Store = Store

unfolding Store-def
apply (rule exI [where ?x=λ newOID = (λC. 0),
               newAID = (λT. 0),
               vals = (λl. init (ltype l)) []])
apply (auto simp add: aliveImpl-def init-def NullT-leaf-array split: Javatype.splits)
done
\end{verbatim}

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

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

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

```plaintext
consts access:: Store ⇒ Location ⇒ Value (-@- [71,71] 70)
alive:: Value ⇒ Store ⇒ bool
alloc:: Store ⇒ New ⇒ Store
new:: Store ⇒ New ⇒ Value
update:: Store ⇒ Location ⇒ Value ⇒ Store
```

nonterminal smodifybinds and smodifybind

syntax

```
-smodifybind :: ['a, 'a] ⇒ smodifybind ((2- := / -))
:: smodifybind ⇒ smodifybinds (-)
:: CTypeId ⇒ smodifybind ⇒ smodifybinds (-)
-smodifybinds:: [smodifybind, smodifybinds] ⇒> smodifybinds (-,/ -)
-sModify :: ['a, smodifybinds] ⇒ 'a (-/(-)) [900,0] 900
translations
-sModify s (-smodifybinds b bs) == -sModify (-sModify s b) bs
s(x:=y) == CONST update s x y
s(c) == CONST alloc s c
```

With this syntactic setup we can write chains of (array) updates and allocations like in the following term:

```
```

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

```plaintext
defs alive-def:
alive x s ≡ aliveImpl x (Rep-Store s)
```

```plaintext
defs access-def:
access s l ≡ vals (Rep-Store s) l
```

```plaintext
defs update-def:
update s l v ≡ if alive (ref l) s ∧ alive v s ∧ typeof v ≤ ltype l
then Abs-Store ((Rep-Store s)(vals:=(vals (Rep-Store s))(l:=v)))
else s
```

```plaintext
defs alloc-def:
alloc s t ≡
(case t of
  new-instance C
```
12.4 Derived Properties of the Store


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

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

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

definition
\text{wts : Store } \Rightarrow bool \text{ where}
\text{wts } OS = (\forall (l::Location). (\text{typeof } (OS@@l)) \leq (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 \text{alive-init} [simp,intro]: alive (init T) s
\text{by (cases } T \text{) (simp-all add: alive-def aliveImpl-def)}

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

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

lemma \text{alive-arr-len} [simp]:
\text{isArrV } x \Rightarrow \text{alive (ref } (\text{arr-len } x)) s = \text{alive } x s
\text{by (cases } x \text{) (simp-all)}

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

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

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

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

The store is well-typed by construction.

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

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

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

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

lemma vals-update-in-Store:
  assumes alive-l: alive (ref l) s
  assumes alive-y: alive y s
  assumes type-conform: typeof y \leq ltype l
  shows (Rep-Store s(vals := vals (Rep-Store s))(l := y)) \in Store
(is ?s-upd \in Store)
proof –
  have s: Rep-Store s \in 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 `s-upd `s-upd l = init (ltype l)
proof (intro allI impl)
  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
ultimately show ?thesis
  by (simp add: Store-def)
eral updating invariant [simp]: alive x (s(l:=y)) = alive x s

proof (rule update-induct)
  show alive x s = alive x s.

next
  assume alive (ref l) s alive y s typeof y ≤ ltype l
  hence Rep-Store
    (Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := y))))
    = Rep-Store s(vals := (vals (Rep-Store s))(l := y))
  by (rule vals-update-in-Store [THEN Abs-Store-inverse])
  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

access updating other [simp]:

assumes neq-l-m: l ≠ m
shows s(l:=x)@@m = s@@m

proof (rule update-induct)
  show s@@m = s@@m.

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

qed

Updating access same [simp]:

assumes alive-l: alive (ref l) s
assumes alive-x: alive x s
assumes widen-x-l: typeof x ≤ ltype l
shows s(l:=x)@@l = x

proof
  from alive-l alive-x widen-x-l
  have Rep-Store
    (Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := x))))
    = Rep-Store s(vals := (vals (Rep-Store s))(l := x))
  by (rule vals-update-in-Store [THEN Abs-Store-inverse])
  hence Abs-Store (Rep-Store s(vals := (vals (Rep-Store s))(l := x)))@@l = x
  by (simp add: access-def)
  with alive-l alive-x widen-x-l
  show ?thesis
  by (simp add: update-def)

qed
12.4 Derived Properties of the Store

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

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

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

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

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

lemma alloc-new-instance-in-Store:
  (Rep-Store s∥newOID := λD. if C = D then Suc (newOID (Rep-Store s) C) else newOID (Rep-Store s) D∥) ∈ Store

(is ?s-alloc ∈ Store)

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

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

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

proof –

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

have \( \forall \ l. \text{aliveImpl (vals ?s-alloc l) ?s-alloc} \)
proof
fix \( l \) show \( \text{aliveImpl (vals ?s-alloc l) ?s-alloc} \)
proof (cases \( l = \text{arrLenLoc T (newAID (Rep-Store s) T}) \))
  case \( \text{True} \)
  thus \( ?\text{thesis} \)
  by (simp add: \text{aliveImpl-def split: Value.splits})
next
  case \( \text{False} \)
from \( s \) have \( \forall \ l. \text{aliveImpl (vals (Rep-Store s) l) (Rep-Store s)} \)
by (simp add: \text{Store-def})
with \( \text{False show } ?\text{thesis} \)
by (auto intro: \text{less-SucI simp add: aliveImpl-def split: Value.splits})
qed
qed
moreover
have \( \forall \ l. \neg \text{aliveImpl (ref l) ?s-alloc } \rightarrow \text{vals ?s-alloc l} = \text{init (ltype l)} \)
proof (intro allI \text{impl})
  fix \( l \)
  assume \( \text{unalive: } \neg \text{aliveImpl (ref l) ?s-alloc} \)
show \( \text{vals ?s-alloc l} = \text{init (ltype l)} \)
proof (cases \( l = \text{arrLenLoc T (newAID (Rep-Store s) T}) \))
  case \( \text{True} \)
  with \( \text{unalive show } ?\text{thesis by (simp add: aliveImpl-def)} \)
next
  case \( \text{False} \)
from \( \text{unalive} \)
have \( \neg \text{aliveImpl (ref l) (Rep-Store s)} \)
by (simp add: \text{aliveImpl-def split: Value.splits split-if-asm})
with \( s \) have \( \text{vals (Rep-Store s) l} = \text{init (ltype l)} \)
by (simp add: \text{Store-def})
with \( \text{False show } ?\text{thesis} \)
  by simp
qed
qed
moreover
from \( s \) have \( \forall \ l. \text{typeof (vals ?s-alloc l)} \leq \text{ltype l} \)
  by (simp add: \text{Store-def})
ultimately
show \( ?\text{thesis} \)
  by (simp add: \text{Store-def})
qed
lemma new-alive-alloc [simp,intro]: alive (new s t) (s(t))
proof (cases t)
  case new-instance thus ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
      alloc-new-instance-in-Store [THEN Abs-Store-inverse])
next
  case new-array thus ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
      alloc-new-array-in-Store [THEN Abs-Store-inverse])
qed

lemma value-class-inhabitants:
(\forall x. typeof x = CClassT typeId \rightarrow P x) = (\forall a. P (objV 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. typeof x = ArrT typeId \rightarrow P x) = (\forall a. P (arrV 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))
(is (\forall a. ?A a) = (\forall a. ?B a))
proof
  assume \forall a. ?A a thus \forall a. ?B a
    by fastforce
next
  assume B: \forall a. ?B a
    proof
      fix a from B show ?A a
        by (cases a) simp-all
Lemma `all-le-eq-imp-eq`: \( \forall c \in \mathbb{N}. (a < d) = (a < c) \rightarrow (d = c) \)

Proof (induct \( d \))
- Case \( 0 \) thus ?case by fastforce
- Next
  - Case \( \text{Suc } n \ c \)
  - Thus ?case
    - By (cases \( c \)) (auto simp add: le-Suc-eq)

Qed

Lemma `all-le-eq`: \( \forall a \in \mathbb{N}. (a < d) = (a < c) = (d = c) \)

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

Lemma `typeof-new`: `typeof (new s t)` = `typeofNew t`

By (cases \( t \)) (simp-all add: new-def typeofNew-def)

Store12

Lemma `new-eq`: \( \text{new } s_1 t = \text{new } s_2 t \) = \( \forall x. \text{typeof } x = \text{typeofNew } t \rightarrow \text{alive } x s_1 = \text{alive } x s_2 \)

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⟨t:=x⟩)` = `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 \ \lor \ \ (x = \text{new } s \ t))`

Proof
  - Assume `alive-alloc`: `alive x (s⟨t⟩)`
  - Show `alive x s \ \lor \ \ x = \text{new } s \ t`
  - Proof (cases \( t \))
    - Case `(new-instance C)`
      - With `alive-alloc` show ?thesis
        - By (cases \( x \)) (auto split: split-if-asm simp add: alive-def new-def alloc-def aliveImpl-def)
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: split-if-asm
      simp add: alive-def new-def alloc-def aliveImpl-def
      alloc-new-array-in-Store THEN Abs-Store-inverse])
  qed
next
assume alive x s \lor x = new s t
then show alive x (s(t))
proof
  assume alive x s thus ?thesis by (rule alive-alloc-propagation)
next
  assume new: x = new s t show ?thesis
  proof (cases t)
    case new-instance with new show ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
      alloc-new-instance-in-Store THEN Abs-Store-inverse])
  next
    case new-array with new show ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
      alloc-new-array-in-Store THEN Abs-Store-inverse])
  qed
  qed
qed

lemma alive-alloc-cases [consumes 1]:
[[ alive x (s(t)); alive x s \Rightarrow P; x = new s t \Rightarrow P ]]
\Rightarrow 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 \rightarrow l \neq arr-len (new s t)
  shows s(t)@@l = init (ltype l)
proof
  from ref-new
  have \neg 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: access-def alloc-def)
    from True show ?thesis
    proof (cases rule: alive-alloc-cases)
      assume alive-l-s: alive (ref l) s
      with new-unalive-old-Store
      have l-not-new: ref l ≠ new s t
        by fastforce
      hence vals (Rep-Store (s(t))) l = s@@l
        by (cases t)
        (auto simp add: alloc-def new-def access-def
        alloc-new-instance-in-Store [THEN Abs-Store-inverse]
        alloc-new-array-in-Store [THEN Abs-Store-inverse])
      with access-alloc-vals
      show ?thesis
      by simp
    next
    assume ref-new: ref l = new s t
    with no-arr-len-new
    have s(t)@@l = init (ltype l)
      by (simp add: access-new)
    moreover
    from ref-new have s@@l = init (ltype l)
      by simp
    ultimately
    show ?thesis by simp
    qed
  next
  case False
  hence s(t)@@l = init (ltype l)
    by (simp)
  moreover
  from False have ¬ alive (ref l) s
12.4 Derived Properties of the Store

by (auto simp add: alive-alloc-propagation)
hence s@l = init (ltype l)
by simp
ultimately show \(?thesis by simp
qed
qed

Store13

lemma Store-eqI:
  assumes eq-alive: \( \forall x. \) alive \( x \) \( s1 = alive x s2 \)
  assumes eq-access: \( \forall l. \) s1@@l = s2@@l
  shows s1 = s2
proof (cases s1 = s2)
  case True thus \(?thesis .
next
case False note neq-s1-s2 = this
show \(?thesis
proof (cases newOID (Rep-Store s1) = newOID (Rep-Store s2))
  case False
  have \( \exists C. \) newOID (Rep-Store s1) \( C \neq newOID (Rep-Store s2) C \)
  proof (rule ccontr)
    assume \( \neg (\exists C. \) newOID (Rep-Store s1) \( C \neq newOID (Rep-Store s2) C \)
    then have newOID (Rep-Store s1) = newOID (Rep-Store s2)
      by (blast intro: ext)
  with False show False ..
qed
with eq-alive obtain C
  where newOID (Rep-Store s1) \( C \neq newOID (Rep-Store s2) C \)
  \( \forall a. \) alive (objV C a) \( s1 = alive (objV C a) s2 \)
  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 \( \exists T. \) newAID (Rep-Store s1) \( T \neq newAID (Rep-Store s2) T \)
  proof (rule ccontr)
    assume \( \neg (\exists T. \) newAID (Rep-Store s1) \( T \neq newAID (Rep-Store s2) T \)
    then have newAID (Rep-Store s1) = newAID (Rep-Store s2)
      by (blast intro: ext)
  with False show False ..
qed
with eq-alive obtain T
  where newAID (Rep-Store s1) \( T \neq newAID (Rep-Store s2) T \)
  \( \forall a. \) alive (arrV T a) \( s1 = alive (arrV T a) s2 \)
  by auto
then show \(?thesis
by (simp add: all-le-eq alive-def aliveImpl-def)
next
case True note eq-newAID = this
show \(?thesis
proof (cases vals (Rep-Store s1) = vals (Rep-Store s2))
  case True
  with eq-newOID eq-newAID
have \((\text{Rep-Store } s1) = (\text{Rep-Store } s2)\)
  by (cases \(s1\), cases \(s2\)) simp
hence \(s1 = s2\)
  by (simp add: \text{Rep-Store-inject})
with \(\text{neq-s1-s2}\) show \(?\text{thesis}\)
  by simp
next
  case False
  have \(\exists \ l. \text{vals } (\text{Rep-Store } s1) l \neq \text{vals } (\text{Rep-Store } s2) l\)
    proof (rule ccontr)
      assume \(\neg (\exists l. \text{vals } (\text{Rep-Store } s1) l \neq \text{vals } (\text{Rep-Store } s2) l)\)
      hence \(\text{vals } (\text{Rep-Store } s1) = \text{vals } (\text{Rep-Store } s2)\)
        by (blast intro: ext)
      with False show False ..
    qed
  then obtain \(l\)
    where \(\text{vals } (\text{Rep-Store } s1) l \neq \text{vals } (\text{Rep-Store } s2) l\)
      by auto
    with eq-access have False
      by (simp add: access-def)
    thus \(?\text{thesis}\) ..
  qed
  qed
qed

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 \(\text{neq-l-new}: \text{ref } l \neq \text{new } s t\)
   assumes \(\text{neq-x-new}: x \neq \text{new } s t\)
   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 \(\dots \ = (alive \ y \ s \lor (y = \text{new } s t))\)
            by (rule alive-alloc-exhaust)
          also have \(\dots \ = (alive \ y \ (s(l:=x)) \lor y = \text{new } s t)\)
            by (simp only: alive-update-invariant)
          also have \(\dots \ = (alive \ y \ (s(l:=x)) \lor y = \text{new } (s(l:=x)) \ t)\)
            proof
              have \(\text{new } s t = \text{new } (s(l:=x)) \ t\)
                by simp
              thus \(?\text{thesis}\) by simp
            qed
          qed
        qed
      qed
    next
      fix \(k\)
show \( s(t)(l := x)@@k = s(l := x)(t)@@k \)

**proof** (cases \( l = k \))

- **case** \( False \) **note** \( neq-l-k = \text{this} \)
  **show** ?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 \( \ldots = s(t := x)@@k \) **by** simp
      also from \( True \)
      have \( \ldots = s(l := x)(t)@@k \) **by** simp
      finally **show** ?thesis .

- **next**

- **case** \( False \)
  then obtain \( T \ n \) **where**
  \( t \colon :=\text{new-array T n and} \ k \colon =\text{arr-len (new s t)} \)
  **by** (cases \( t \)) **auto**
  from \( k \) **have** \( k' \colon =\text{arr-len (new (s\{l := x\}) (new-array T n)} \)
  **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** ?thesis .

**qed**

**next**

- **case** \( True \) **note** \( eq-l-k = \text{this} \)
  **have** lemma-3-1:
  \( \text{ref l \neq new s t \Rightarrow alive (ref l) (s(t)) = alive (ref l) s} \)
  **by** (simp add: alive-alloc-exhaust)
  **have** lemma-3-2:
  \( x \neq new s t \Rightarrow alive x (s(t)) = alive x s \)
  **by** (simp add: alive-alloc-exhaust)
  **have** lemma-3-3:
  \( s(l := x, t)@@l = s(l := x)@@l \)
  **proof** –
  from \( neq-l-new \) **have** \( \text{ref l \neq new (s(l := x)) t} \)
  **by** simp
  hence \( \text{isNewArr t \rightarrow l \neq \text{arr-len (new (s(l := x)) t)} \)
  **by** (cases \( t \)) **auto**
  thus ?thesis
  **by** (simp)
**qed**

**show** ?thesis

**proof** (cases alive \( x \ s \))

- **case** \( True \) **note** \( alive-x = \text{this} \)
  **show** ?thesis
  **proof** (cases alive \( \text{ref l} \ s \))
case True note alive-l = this
show ?thesis
proof (cases typeof x ≤ ltype l)
case True
  with alive-l alive-x
  have s(l:=x)@@l = x
    by (rule update-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 ¬ alive (ref l) (s(t))
    by (simp add: lemma-3-1)
  then have s(t)(l:=x)@@l = init (ltype l)
    by simp
  also from not-alive-l have ... = s(l:=x)@@l
    by simp
  also have ... = s(l:=x)(t)@@l
    by (simp add: lemma-3-3)
  finally show ?thesis by (simp add: eq-l-k)
  qed
next
  case False note not-alive-x = this
  from not-alive-x neq-x-new have ¬ alive x (s(t))
    by (simp add: lemma-3-2)
  then have s(t)(l:=x)@@l = s(t)@@l
    by (simp)
  also have ... = s@@l
  proof
    from neq-l-new
    haveisNewArr t → l ≠ arr-len (new s t)
      by (cases t) auto
    thus ?thesis
      by (simp)
    qed
  also from not-alive-x have ... = s(l:=x)@@l
    by (simp)
  also have ... = s(l:=x)(t)@@l
by (simp add: lemma-3-3)

finally show ?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 $reachS$ 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 $reach$ decides whether a location is reachable from a value in a store.

inductive $reach$ :: $Store \Rightarrow Location \Rightarrow Value \Rightarrow bool$
  (⊢ reachable′ from \[91,91,91\])

for $s$ :: $Store$
where
  Immediate: $ref\ l \neq nullV \Rightarrow s\vdash l \text{ reachable-from} (ref\ l)$

| Indirect: $[s\vdash l \text{ reachable-from} (s\@@k); \quad ref\ k \neq nullV]$
  \quad $\Rightarrow s\vdash l \text{ reachable-from} (ref\ k)$

Note that we explicitly exclude $nullV$ as legal reference for reachability. Keep in mind that static fields are not associated to any object, therefore $ref$ yields $nullV$ if invoked on static fields (see the definition of the function $ref$, 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 $nullV$ 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 $x \neq nullV$

using $reach$ by (induct) auto

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

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

lemma reachObjLoc [simp,intro]:
\( C = \text{cls} \ cf \implies s^+ \text{objLoc} \ cf \ a \ \text{reachable-from} \ \text{objV} \ C \ a \)

by (iprover intro: reach.Immediate [of \text{objLoc} \ cf \ a, simplified])

lemma reachArrLoc [simp, intro]: \( s^+ \text{arrLoc} \ T \ a \ i \ \text{reachable-from} \ \text{arrV} \ T \ a \)

by (rule reach.Immediate [of \text{arrLoc} \ T \ a \ i, simplified])

lemma reachArrLen [simp, intro]: \( s^+ \text{arrLenLoc} \ T \ a \ \text{reachable-from} \ \text{arrV} \ T \ a \)

by (rule reach.Immediate [of \text{arrLenLoc} \ T \ a, simplified])

lemma unreachStatic [simp]: \( \neg \ s^+ \text{staticLoc} \ f \ \text{reachable-from} \ x \)

proof
  
  \{
  \begin{align*}
  \text{fix} \ y & \ \text{assume} \ s^+ \text{reachable-from} \ x \ y = \text{staticLoc} \ f \\
  \text{then have} \ False & \ \text{by induct auto}
  \end{align*}
  \}

  \text{thus} \ ?\text{thesis}
    
  \text{by auto}

qed

lemma unreachStaticE [elim]: \( s^+ \text{staticLoc} \ f \ \text{reachable-from} \ x \implies P \)

by (simp add: unreachStatic)

lemma reachable-from-ArrLoc-impl-Arr [simp, intro]:

\begin{align*}
\begin{align*}
\text{assumes} & \ \text{reach-loc:} \ s^+ l \ \text{reachable-from} \ \(s@\text{arrLoc} \ T \ a \ i\) \\
\text{shows} & \ s^+ l \ \text{reachable-from} \ \text{arrV} \ T \ a \\
\text{using} & \ \text{reach.Indirect [OF reach-loc]} \\
\text{by} & \ \text{simp}
\end{align*}
\end{align*}

lemma reachable-from-ObjLoc-impl-Obj [simp, intro]:

\begin{align*}
\begin{align*}
\text{assumes} & \ \text{reach-loc:} \ s^+ l \ \text{reachable-from} \ \(s@\text{objLoc} \ cf \ a\) \\
\text{assumes} & \ C: \ C = \text{cls} \ cf \\
\text{shows} & \ s^+ l \ \text{reachable-from} \ \text{objV} \ C \ a \\
\text{using} & \ C \ \text{reach.Indirect [OF reach-loc]} \\
\text{by} & \ \text{simp}
\end{align*}
\end{align*}

Lemma 3.2 (i)

lemma reach-update [simp]:

\begin{align*}
\begin{align*}
\text{assumes} & \ \text{unreachable-l-x:} \ \neg s^+ l \ \text{reachable-from} \ x \\
\text{shows} & \ s(l := y)^+ k \ \text{reachable-from} \ x = s^+ k \ \text{reachable-from} \ x \\
\text{proof} & \\
\text{assume} & \ s^+ k \ \text{reachable-from} \ x \\
\text{from} & \ \text{this unreachable-l-x} \\
\text{show} & \ s(l := y)^+ k \ \text{reachable-from} \ x \\
\text{proof} & \ (induct) \\
\text{case} & \ (\text{Immediate} \ k) \\
\text{have} & \ \text{ref} k \neq \text{nullV} \ \text{by fact} \\
\text{then show} & \ s(l := y)^+ k \ \text{reachable-from} \ (\text{ref} \ k) \\
\text{by} & \ \text{(rule reach.Immediate)}
\end{align*}
\end{align*}
\end{align*}

next

\begin{align*}
\begin{align*}
\text{case} & \ (\text{Indirect} \ k \ m) \\
\text{have} & \ \text{hyp:} \ \neg s^+ l \ \text{reachable-from} \ \(s@m\) \\
\text{implies} & \ s(l := y)^+ k \ \text{reachable-from} \ \(s@m\) \ \text{by fact} \\
\text{have} & \ \text{ref} m \neq \text{nullV and} \ \neg s^+ l \ \text{reachable-from} \ (\text{ref} \ m) \ \text{by fact} \\
\end{align*}
\end{align*}
\end{align*}
13.1 Reachability of a Location from a Reference

hence \( l \neq m \rightarrow s \vdash l \text{ reachable-from } (s \cdot m) \)
by (auto intro: reach.intros)
with hyp have \( s(l := y) \vdash k \text{ reachable-from } (s(l := y) \cdot m) \)
by simp
then show \( s(l := y) \vdash k \text{ reachable-from } (\text{ref } m) \)
by (rule reach.Indirect) (rule Indirect.hyps)
qed

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

Lemma 3.2 (ii)

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

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

qed
lemma val-ArrLen-IntgT: isArrLenLoc l \implies \text{typeof}(s@\langle l \rangle) = \text{IntgT}

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

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

Lemma 3.2 (v)

lemma reach.alloc [simp]: s(t)\vdash l \text{ reachable-from } x = s\vdash l \text{ reachable-from } x

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

next
assume $s \vdash l \text{ reachable-from } x$
thus $s(t) \vdash l \text{ reachable-from } x$
proof (induct)
case (Immediate $l$)
thus $s(t) \vdash l \text{ reachable-from ref } l$
by (rule reach.intros)
next
case (Indirect $l$ $k$)
have reach-$k$: $s \langle t \rangle \vdash l \text{ reachable-from } (s @@ k)$ by fact
moreover
have $s(t) @@ k = s @@ k$
proof −
from reach-$k$ have isRefV: isRefV (s @@ k)
by (rule reachable-isRef)
have $\neg \text{ArrLenLoc } k$
proof (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 $\neg \text{thesis}$
by (rule access-alloc')
qed
ultimately have $s(t) \vdash l \text{ reachable-from } (s(t) @@ k)$
by simp
thus $s(t) \vdash l \text{ reachable-from ref } k$
by (rule reach.intros) (rule Indirect.hyps)
qed

Lemma 3.2 (vi)

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

hence isRefV $x$
by (rule reachable-isRef)
with prim show False
by (cases $x$) simp-all
qed

Lemma 3.2 (iii)

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

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

next
case (Indirect \( m \text{ k} \))
  have \( k \text{-not-Null} \): \( \text{ref } k \neq \text{nullV} \) by fact
  have \( not-m-g \): \( \neg s^m m \text{ reachable-from } y \) by fact
  have \( not-m-k \): \( \neg s^m m \text{ reachable-from } \text{ref } k \) by fact
  have hyp: \[ \neg \vdash s m \text{ reachable-from } y; \neg s^m m \text{ reachable-from } (s(l := y)@@k) \]
  \[ \rightarrow \text{False by fact} \]
  have \( m \text{-udp-k} \): \( s(l := y) \vdash m \text{ reachable-from } (s(l := y)@@k) \) by fact
  show False
  proof (cases \( l = k \))
    case False
    then have \( s(l := y)@@k = s @@ k \) by simp
    moreover
    from \( not-m-k \text{-} k \text{-not-Null} \) have \( \neg s^m m \text{ reachable-from } (s@@k) \)
    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 \( \text{ref } l \) \( s \) \( \land \) alive \( y \) \( s \) \( \land \) typeof \( y \leq \text{ltype } l \))
      case True
      with eq-l-k have \( s(l := y)@@k = y \)
      by simp
      with \( not-m-y \) hyp show False by simp

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

qed

Lemma 3.2 (vii).

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

lemma ref-reach-unalive:
  assumes unalive-x: \( \neg \text{alive } x \) \( s \)
  assumes l-x: \( s^l l \text{ reachable-from } x \)
shows \( x = \text{ref } l \)
using \( l-x \) \text{unalive-}\!x
proof induct
  case (Immediate \( l \))
  show \( \text{ref } l = \text{ref } l \)
    by simp
next
  case (Indirect \( l \) \( k \))
  have \( \text{ref } k \neq \text{null}\!V \) by fact
  have \( \neg \text{alive } (\text{ref } k) \) \( s \) by fact
  hence \( s@k = \text{init } (l\text{type } k) \) by simp
  moreover have \( s = l \) reachable-from \( (s@k) \) by fact
ultimately have \( \text{False} \) by simp
thus ?case ..
qed

lemma \( \text{loc-new-reach} \):
  assumes \( l: \text{ref } l = \text{new } s \) \( t \)
  assumes \( l-x: s = l \) reachable-from \( x \)
  shows \( x = \text{new } s \) \( t \)
using \( l-x \) \( l \)
proof induct
  case (Immediate \( l \))
  show ?case by fact
next
  case (Indirect \( l \) \( k \))
  hence \( s@k = \text{new } s \) \( t \) by iprover
  moreover have \( \neg \text{alive } (s@k) \) \( s \)
    by simp
  moreover have \( \text{alive } (s@k) \) \( s \)
    by simp
ultimately have \( \text{False} \) by simp
thus ?case ..
qed

Lemma 3.2 (viii)

lemma \( \text{alive-reach-alive} \):
  assumes \( \text{alive-}\!x: \text{alive } x \) \( s \)
  assumes \( \text{reach-}\!l: s = l \) reachable-from \( x \)
  shows \( \text{alive } (\text{ref } l) \) \( s \)
using \( \text{reach-}\!l \) \( \text{alive-}\!x \)
proof (induct)
  case (Immediate \( l \))
  show ?case by fact
next
  case (Indirect \( l \) \( k \))
  have \( \text{hyp: alive } (s@k) \) \( s \Rightarrow \text{alive } (\text{ref } l) \) \( s \) by fact
  moreover have \( \text{alive } (s@k) \) \( s \) by simp
ultimately
  show \( \text{alive } (\text{ref } l) \) \( s \)
    by iprover
qed
Lemma 3.2 (ix)

Lemma \( \text{reach} \):

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

Shows \( s1 \vdash l \text{ reachable-from } x = s2 \vdash l \text{ reachable-from } x \)

Proof

Assume \( s1 \vdash l \text{ reachable-from } x \)

From this reach-impl-access-eq

Show \( s2 \vdash l \text{ reachable-from } x \)

Proof (induct)

Case (Immediate \( l \))

Show \( s2 \vdash l \text{ reachable-from } \text{ref } l \)

By (rule reach.intros) (rule Immediate.hyps)

Next

Case (Indirect \( l \) \( k \))

Have hyp: \( \forall l. s1 \vdash l \text{ reachable-from } (s1 @ @ k) \rightarrow s1 @ @ l = s2 @ @ l \)

\( \implies s2 \vdash l \text{ reachable-from } (s1 @ @ k) \) by fact

Have k-not-Null: \( \text{ref } k \neq \text{nullV} \) by fact

Have reach-impl-access-eq:

\( \forall l. s1 \vdash l \text{ reachable-from } \text{ref } k \rightarrow s1 @ @ l = s2 @ @ l \) by fact

Have \( s1 \vdash l \text{ reachable-from } (s1 @ @ k) \) by fact

With k-not-Null

Have \( s1 @ @ k = s2 @ @ k \) by (iprover intro: reach-impl-access-eq [rule-format] reach.intros)

Moreover from reach-impl-access-eq k-not-Null

Have \( \forall l. s1 \vdash l \text{ reachable-from } (s1 @ @ k) \rightarrow s1 @ @ l = s2 @ @ l \)

By (iprover intro: reach.intros)

Then have \( s2 \vdash l \text{ reachable-from } (s1 @ @ k) \)

By (rule hyp)

Ultimately have \( s2 \vdash l \text{ reachable-from } (s2 @ @ k) \)

By simp

Thus \( s2 \vdash l \text{ reachable-from } \text{ref } k \)

By (rule reach.intros) (rule Indirect.hyps)

Qed

Next

Assume \( s2 \vdash l \text{ reachable-from } x \)

From this reach-impl-access-eq

Show \( s1 \vdash l \text{ reachable-from } x \)

Proof (induct)

Case (Immediate \( l \))

Show \( s1 \vdash l \text{ reachable-from } \text{ref } l \)

By (rule reach.intros) (rule Immediate.hyps)

Next

Case (Indirect \( l \) \( k \))

Have hyp: \( \forall l. s1 \vdash l \text{ reachable-from } (s2 @ @ k) \rightarrow s1 @ @ l = s2 @ @ l \)

\( \implies s1 \vdash l \text{ reachable-from } (s2 @ @ k) \) by fact

Have k-not-Null: \( \text{ref } k \neq \text{nullV} \) by fact

Have reach-impl-access-eq:

\( \forall l. s1 \vdash l \text{ reachable-from } \text{ref } k \rightarrow s1 @ @ l = s2 @ @ l \) by fact

Have \( s1 \vdash k \text{ reachable-from } \text{ref } k \)

By (rule reach.intros) (rule Indirect.hyps)

With reach-impl-access-eq

Have eq-k: \( s1 @ @ k = s2 @ @ k \)

By simp

From reach-impl-access-eq k-not-Null


have \( \forall l. s_1 \vdash l \text{ reachable-from } (s_1 \oplus \oplus k) \rightarrow s_1 \oplus \oplus l = s_2 \oplus \oplus l \)

by (iprover intro: reach.intro)

then

have \( \forall l. s_1 \vdash l \text{ reachable-from } (s_2 \oplus \oplus k) \rightarrow s_1 \oplus \oplus l = s_2 \oplus \oplus l \)

by (simp add: eq-k)

thus \( s_1 \vdash l \text{ reachable-from } ref k \)

by (rule reach.intro) (rule Indirect.hyps)

qed

13.2 Reachability of a Reference from a Reference

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

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

\[
\text{s} \vdash \text{Ref y reachable-from x} = \left( \exists l. \text{s} \vdash l \text{ reachable-from } x \land y = \text{ref l} \right)
\]

**Notation** \( \text{rreach} (\vdash \text{Ref - reachable \'} \text{-from - [91,91,91]0}) \)

13.3 Disjointness of Reachable Locations

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

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

\[
\text{disj\ x\ y\ s\ =}\ (\forall l. \neg \text{s} \vdash l \text{ reachable-from } x \lor \neg \text{s} \vdash l \text{ reachable-from } y)
\]

**Lemma** \( \text{disjI1} :: \left[ \left\{ \text{\forall l. \neg \text{s \vdash l reachable-from x \rightarrow \neg s \vdash l reachable-from y} \right\} \right] \Rightarrow \text{disj\ x\ y\ s} \)

by (simp add: disj-def)

**Lemma** \( \text{disjI2} :: \left[ \left\{ \text{\forall l. \neg \text{s \vdash l reachable-from y \rightarrow \neg s \vdash l reachable-from x} \right\} \right] \Rightarrow \text{disj\ x\ y\ s} \)

by (auto simp add: disj-def)

**Lemma** \( \text{disj-cases \ [consumes 1]} :: \)

assumes \( \text{disj\ x\ y\ s} \)

assumes \( \text{\forall l. \neg s \vdash l \text{ reachable-from } x \Rightarrow P} \)

assumes \( \text{\forall l. \neg s \vdash l \text{ reachable-from } y \Rightarrow P} \)

shows \( P \)

using \( \text{assms} \) by (auto simp add: disj-def)

Lemma 3.3 (i) in [PH97]

**Lemma** \( \text{disjI} :: \left[ \text{\forall l. \neg s \vdash l \text{ reachable-from } x, \neg s \vdash l \text{ reachable-from } y} \right] \Rightarrow \text{disj\ x\ y\ (s\{l:=z\})} \)

by (auto simp add: disj-def)

Lemma 3.3 (ii)
lemma \textit{disj2}:
\begin{itemize}
\item assumes \textit{disj-x-y}: \textit{disj x y s}
\item assumes \textit{disj-x-z}: \textit{disj x z s}
\item assumes \textit{unreach-l-x}: \neg s \vdash \textit{l reachable-from x}
\end{itemize}
\begin{itemize}
\item shows \textit{disj x y} (s (\textit{l := z}))
\end{itemize}
\begin{proof}
\begin{itemize}
\item fix \textit{k}
\item assume \textit{reach-k-x}: s (\textit{l := z}) \vdash \textit{k reachable-from x}
\item show \neg s (\textit{l := z}) \vdash \textit{k reachable-from y}
\end{itemize}
\begin{itemize}
\item from \textit{unreach-l-x} \textit{reach-k-x}
\item have \textit{reach-s-k-x}: s \vdash \textit{k reachable-from x}
\item by simp
\item with \textit{disj-x-z}
\item have \neg s \vdash \textit{k reachable-from z}
\item by (simp add: \textit{disj-def})
\item moreover from \textit{reach-s-k-x} \textit{disj-x-y}
\item have \neg s \vdash \textit{k reachable-from y}
\item by (simp add: \textit{disj-def})
\item ultimately show \neg \textit{thesis}
\item by (rule \textit{reach3})
\end{itemize}
\end{proof}
\end{proof}
\end{lemma}

Lemma 3.3 (iii)

\begin{lemma} \textit{disj3}:
\begin{itemize}
\item assumes \textit{alive-x-s}: \textit{alive x s}
\item shows \textit{disj x} (new s t) (s (\textit{t}))
\end{itemize}
\begin{proof}
\begin{itemize}
\item fix \textit{l}
\item assume \textit{reach-l-x}: s \vdash \textit{l reachable-from x}
\item show \neg s \vdash \textit{l reachable-from new s t}
\end{itemize}
\begin{itemize}
\item assume \textit{reach-l-new}: s \vdash \textit{l reachable-from new s t}
\item have \textit{unalive-new}: \neg \textit{alive (new s t) s} by simp
\item from this \textit{reach-l-new}
\item have \textit{new s t = ref l}
\item by (rule \textit{ref-reach-unalive})
\item moreover from \textit{alive-x-s} \textit{reach-l-x}
\item have \textit{alive (ref l) s}
\item by (rule \textit{alive-reach-alive})
\item ultimately show \textit{False}
\item using \textit{unalive-new}
\item by simp
\end{itemize}
\end{proof}
\end{lemma}

Lemma 3.3 (iv)

\begin{lemma} \textit{disj4}:
\begin{itemize}
\item \textit{disj (objV C a) y s; CClass C ≤ dtype f}
\item \textit{disj (s@@(objV C a))..f) y s}
\end{itemize}
\begin{proof}
by (auto simp add: \textit{disj-def})
\end{proof}
\end{lemma}

\begin{lemma} \textit{disj4}':
\begin{itemize}
\item \textit{disj (arrV T a) y s}
\item \textit{disj (s@@(arrV T a).[i]) y s}
\end{itemize}
\begin{proof}
by (auto simp add: \textit{disj-def})
\end{proof}
\end{lemma}
13.4 X-Equivalence

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

**Definition** $xeq :: Value \Rightarrow Store \Rightarrow Store \Rightarrow bool$ where

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

**Abbreviation** $xeq$-syntax :: $Store \Rightarrow Value \Rightarrow Store \Rightarrow bool$

\[
(-/ (==[\cdot]) / - [900,0,900] 900)
\]

**Notation** (xsymbols) $xeq$-syntax (/. (≡[\cdot]) / - [900,0,900] 900)

**Lemma** $xeqI$:

\[
\begin{align*}
\text{alive } x \ s &= \text{alive } x \ t \\
(\forall l. s \vdash l \text{ reachable-from } x \implies s@@l = t@@l)
\end{align*}
\]

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** $xeq1-sym'$:

\[
\text{assumes } s-t: s \equiv [x] t
\]

\[
\text{shows } t \equiv [x] s
\]

**Proof** –

\[
\begin{align*}
\text{from } s-t \ \text{have } \text{alive } x \ s &= \text{alive } x \ t \ \text{by (simp add : xeq-def)}
\end{align*}
\]

moreover

\[
\begin{align*}
\text{from } s-t \ \text{have } (\forall l. s \vdash l \text{ reachable-from } x \implies s@@l = t@@l) \\
\text{by (simp add : xeq-def)}
\end{align*}
\]

with reach9 [OF this]

\[
\begin{align*}
\text{have } (\forall l. t \vdash l \text{ reachable-from } x \implies t@@l = s@@l) \\
\text{by simp}
\end{align*}
\]

ultimately show \ ?thesis

\[
\text{by (simp add : xeq-def)}
\]

qed

**Lemma** $xeq1-sym$:

\[
s \equiv [x] t = t \equiv [x] s
\]

by (auto intro : xeq1-sym')

Lemma 3.4 (i)

**Lemma** $xeq1-trans$ [trans]:

\[
\text{assumes } s-t: s \equiv [x] t
\]

\[
\text{assumes } t-r: t \equiv [x] r
\]

\[
\text{shows } s \equiv [x] r
\]

**Proof** –

\[
\begin{align*}
\text{from } s-t t-r \ \text{have } \text{alive } x \ s &= \text{alive } x \ r \\
\text{by (simp add : xeq-def)}
\end{align*}
\]
Moreover

\[ \forall \ l. \ \text{si} \ l \ \text{reachable-from} \ x \rightarrow s\hat{a}l = r\hat{a}l \]

\textbf{proof (intro allI impI)}

fix \ l

\textbf{assume} reach-l: \si \ l \ \text{reachable-from} \ x

\textbf{show} s\hat{a}l = r\hat{a}l

\textbf{proof –}

from reach-l \ \text{s-t} \ \text{have} \ s\hat{a}l = t\hat{a}l

by (simp add: xeq-def)

\textbf{also have} t\hat{a}l = r\hat{a}l

\textbf{proof –}

from s-t \ \text{have} \ \forall \ l. \ \text{si} \ l \ \text{reachable-from} \ x \rightarrow s\hat{a}l = t\hat{a}l

by (simp add: xeq-def)

from reach9 [OF this] \ \text{reach-l} \ \text{have} \ tl \ l \ \text{reachable-from} \ x

by simp

with \ t-r \ \text{show} \ \text{thesis}

by (simp add: xeq-def)

\textbf{qed}

\textbf{finally show} \ \text{thesis}

\quad \textbf{qed}

\textbf{ultimately show} \ \text{thesis}

\quad \textbf{by (simp add: xeq-def)}

\textbf{qed}

\textbf{Lemma 3.4 (ii)}

\textbf{lemma xeq2:}

\textbf{assumes} xeq: \ \forall \ x. \ s \equiv [x] t

\textbf{assumes} static-eq: \ \forall \ f. \ s\hat{a}(\text{staticLoc} \ f) = t\hat{a}(\text{staticLoc} \ f)

\textbf{shows} \ s = t

\textbf{proof (rule Store-eqI)}

from xeq

\textbf{show} \ \forall \ x. \ \text{alive} \ x \ s = \text{alive} \ x \ t

by (simp add: xeq-def)

\textbf{next}

\textbf{show} \ \forall \ l. \ s\hat{a}l = t\hat{a}l

\textbf{proof}

fix \ l

\textbf{show} s\hat{a}l = t\hat{a}l

\textbf{proof (cases \ l)}

\textbf{case} (objLoc cf a)

\textbf{have} \ l = \text{objLoc} \ cf \ a \ \text{by fact}

\textbf{hence} \ si \ l \ \text{reachable-from} \ (\text{objV} \ (\text{cls} \ cf) \ a)

by simp

\textbf{with} \ \text{xeq} \ \text{show} \ \text{thesis}

by (simp add: xeq-def)

\textbf{next}

\textbf{case} (staticLoc \ f)

\textbf{have} \ l = \text{staticLoc} \ f \ \text{by fact}

\textbf{with} \ \text{static-eq} \ \text{show} \ \text{thesis}

by (simp add: xeq-def)

\textbf{next}

\textbf{case} (arrLenLoc \ T \ a)

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

next
\begin{itemize}
\item case (arrLoc T a i)
\item have $l = \texttt{arrLoc T a i}$ by fact
\item hence $s \vdash \texttt{l reachable-from (arrV T a)}$
by simp
with xeq show $?thesis$
by (simp add: xeq-def)
\end{itemize}

qed

Lemma 3.4 (iii)

lemma xeq3:
assumes \texttt{unreach-l: \neg s \vdash \texttt{l reachable-from x}}
shows $s \equiv[x] s(l:=y)$

\begin{proof}(rule xeqI)
show $\texttt{alive x s = alive x (s(l := y))}$
by simp
\end{proof}

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

qed

Lemma 3.4 (iv)

lemma xeq4: assumes \texttt{not-new: x \neq new s t}
shows $s \equiv[t] s(t)$

\begin{proof}(rule xeqI)
from \texttt{not-new}
show $\texttt{alive x s = alive x (s(t))}$
by (simp add: alive-alloc-exhaust)
\end{proof}

next
\begin{itemize}
\item fix $l$
\item assume \texttt{reach-l: s \vdash \texttt{l reachable-from x}}
\item show $s @ @ l = s(t) @ @ l$
\item proof (cases isNewArr t \rightarrow l \neq \texttt{arr-len (new s t)})
\item case True
\item with \texttt{reach-l} show $?thesis$
by simp
\item next
\item case False
\item then obtain $T n$ where $t: t = \texttt{new-array T n}$ and $l: l = \texttt{arr-len (new s t)}$
by (cases t) auto
\item hence $\texttt{ref l = new s t}$
by simp
\item from this \texttt{reach-l}
\item have $x = \texttt{new s t}$
\end{itemize}
Lemma 3.4 (v)

\[\text{lemma } xeq5: s \equiv [x] t \implies s \vdash \text{reachable-from } x = t \vdash \text{reachable-from } x\]

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 \(\text{teq} \) [PH97, p. 55].

\[\text{definition } \text{teq} :: \text{Javatype} \Rightarrow \text{Store} \Rightarrow \text{Store} \Rightarrow \text{bool} \text{ where}\]
\[\text{teq } t \ s_1 \ s_2 = (\forall x. \text{typeof } x \leq t \implies a s_1 [x] = a s_2 [x])\]

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 \(\text{lessalive} \) [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.

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

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

We define an introduction rule for the new operator.

\[\text{lemma } \text{lessaliveI}: \]
\[\implies [\forall x. \text{alive } x \ s \implies s [x] t \land (\forall f. s @@ \text{staticLoc } f = t @@ \text{staticLoc } f)]\]

by (simp add: lessalive-def)

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

\[\text{lemma } \text{lessalive-refl}: s \ll s\]

by (simp add: lessalive-def xeq1-refl)

\[\text{lemma } \text{lessalive-trans [trans]}:\]
\[\text{assumes } s-t: s \ll t\]
\[\text{assumes } t-w: t \ll w\]
\[\text{shows } s \ll w\]
\[\text{proof } (\text{rule lessaliveI})\]
\[\text{fix } x\]
\[\text{assume } \text{alive-x-s: alive } x \ s\]
\[\text{with } s-t \text{ have } s [x] t\]

by (simp add: lessalive-def)

also
have \( t \equiv [x] \ w \)

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

finally show s \equiv [x] \ w.
next
fix f
from s-t t-w show s@staticLoc f = w@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 alive x s = alive x t by simp
      next
        fix l assume reach-s-x: \( s \vdash l \) 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 = init (ltype l)
          by simp
        also from reach-s-x x have \( t \vdash l \) reachable-from x
          by (auto intro: reach.Immediate unreachable-Null)
        with False x have t@l = init (ltype l)
          by simp
        finally show s@l = t@l
          by simp
        qed
    qed
  qed
next
from s-t show \( \forall f. \ s@staticLoc f = w@staticLoc f \)
This gives us a partial ordering on the store. Thus, the type Store can be added to the appropriate type class ord which lets us define the < and ≤ symbols, and to the type class order which axiomatizes partial orderings.

instantiation Store :: ord
begin

definition le-Store-def: s ≤ t ←→ s ≪ t

definition less-Store-def: (s::Store) < t ←→ s ≤ t ∧ ¬ t ≤ s

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

Lemma 3.5 (i)

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

end

Lemma 3.5 (ii)

lemma lessalive2: [s ≪ t; alive x s] ⇒ alive x t
by (simp add: lessalive-def xeq-def)

Lemma 3.5 (iii)

lemma lessalive3:
  assumes s-t: s ≪ t
  assumes alive: alive x s ∨ ¬ alive x t
  shows s ≡ [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

with alive have unalive-x-t: ¬ alive x t
  by simp

show thesis

proof (rule xeqI)
  from False alive show alive x s = alive x t
  by simp

next

fix l assume reach-s-x: s ⊢ l reachable-from x

with unalive-x-s have x: x = ref l
  by (rule ref-reach-unalive)

with unalive-x-s have s@@l = init (ltype l)
  by simp

also from reach-s-x x have t ⊢ l reachable-from x
  by (auto intro: reach.Immediate unreachable-Null)

with unalive-x-t x have t@@l = init (ltype l)
  by simp

finally show s@@l = t@@l
  by simp

qed

Lemma 3.5 (iv)

lemma lessalive-update [simp, intro]:
  assumes s-t: s ≪ t
  assumes unalive-l: ¬ alive (ref l) t
  shows s ≪ t(l:=x)

proof −
  from unalive-l have t(l:=x) = t
  by simp

with s-t show thesis by simp

qed

lemma Xequ4′:
  assumes alive: alive x s
  shows s ≡[x] s(t)

proof −
  from alive have x ≠ new s t
    by auto
  thus thesis
    by (rule xeq4)

qed

Lemma 3.5 (v)

lemma lessalive-alloc [simp, intro]: s ≪ s(t)
  by (simp add: lessalive-def Xequ4′)

13.7 Reachability of Types from Types

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

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

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

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

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

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

end

14 The Formalization of JML Operators

theory JML imports ../Isabelle-Store/StoreProperties begin
JML operators that are to be used in Hoare formulae can be formalized here.
definition instanceof :: Value ⇒ Javatype ⇒ bool (- @instanceof -)
where
  instanceof v t = (typeof v ≤ t)
end

15 The Universal Specification

theory UnivSpec imports ../Isabelle/JML begin
This theory contains the Isabelle formalization of the program-dependent specification. This
theory has to be provided by the user. In later versions of Jive, one may be able to generate it
from JML model classes.
definition aCounter :: Value ⇒ Store ⇒ JavaInt where
aCounter x s =
  (if x ∼= nullV & (alive x s) & typeof x = CClassT CounterImpl then
   aI ( s@@(x..CounterImpl'value) )
  else undefined)
end

References

[Jiv]  Jive project webpage.  http://softech.informatik.uni-kl.de/softech/content/
eforschung/e3490/index_ger.html.

