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 **J**ava **I**nteractive **V**erification **E**nvironment, Jive.

Contents

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 $\mathcal{A} \rhd \{ \mathbf{P} \} \ \mathtt{pp} \ \{ \mathbf{Q} \}$ and express that for all states S that fulfill $\mathbf{P}$, if the execution of the program part $\mathtt{pp}$ terminates, the state that is reached when *pp* has been evaluated in S must fulfill $\mathbf{Q}$. The so-called assumptions $\mathcal{A}$ are used to prove recursive methods.

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

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

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

A large number of the rules and axioms of the Hoare logic is related to the structure of the program part that is currently being examined. Besides these, the logic also contains rules that manipulate the pre- or postcondition of the examined subgoal without affecting the current program part selection. A prominent member of this kind of rules is the rule of consequence[1]:

$$\frac{\mathbf{PP} \Rightarrow \mathbf{P} \qquad \mathcal{A} \rhd \{ \mathbf{P} \} \ \mathtt{pp} \ \{ \mathbf{Q} \} \qquad \mathbf{Q} \Rightarrow \mathbf{QQ}}{\mathcal{A} \rhd \{ \mathbf{PP} \} \ \mathtt{pp} \ \{ \mathbf{QQ} \}}$$

It plays a special role in the Hoare logic because it additionally requires implications between stronger and weaker conditions to be proven. If a JIVE proof contains an application of the rule of consequence, the implication is attached to the proof tree node that documents this rule application; these attachments are called lemmas. JIVE sends these lemmas to an associated

[1] In JIVE, the rule of consequence is part of a larger rule which serves several purposes at once. Since we want to focus on the rule of consequence, we left out the parts that are irrelevant in this context.

general purpose theorem prover where the user is required to prove them. Currently, Jive supports Isabelle/HOL as associated prover. It is required that all lemmas that are attached to any node of a proof tree are proven before the initial goal of the proof tree is accepted as being proven.

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

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

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

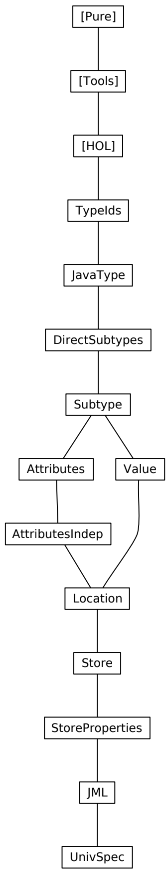
- In PVS, function arguments were sometimes restricted to subtypes. In Isabelle/HOL, unintended usage of functions is left unspecified.
- In PVS, the program-independent theories were parameterized by the datatypes that were generated for the program to be verified. In Isabelle/HOL, we just build on the generated theories. This makes the whole setting easier. The drawback is that we have to run the theories for each program we want to verify. But the proof scripts are designed in a way that they will work if the basic program-dependent theories are generated in the proper way. Since we can create an image of a proof session before starting actual verification we do not run into time problems either.
- The subtype relation is based on the direct subtype relation between classes and interfaces. We prove that subtyping forms a partial order. In the PVS version subtyping was expressed by axioms that described the subtype relation for the types appearing in the Java program to be verified.

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

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

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

2 Theory Dependencies



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

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

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

Isa_⟨Prog⟩:
- TypeIds.thy
- DirectSubtypes.thy
- UnivSpec.thy

Isa_⟨Prog⟩_Store:
- Attributes.thy

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

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

```
ML {*
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:

```
    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 $\times$ TypeId" and if we quantify over all objects, we get a lot of objects that do not exist, namely all objects that bear an interface type identifier or abstract class identifier. This is not very helpful. Therefore, we separate the three identifier kinds from each other.

end

5 Java-Type

theory *JavaType* **imports** ../*Isa-Counter*/*TypeIds*
begin

This theory formalizes the types that appear in a Java program. Note that the types defined by the classes and interfaces are formalized via their identifiers. This way, this theory is program-independent.

We only want to formalize one-dimensional arrays. Therefore, we describe the types that can be used as element types of arrays. This excludes the `null` type and array types themselves. This way, we get a finite number of types in our type hierarchy, and the subtype relations can be given explicitly (see Sec. 6). If desired, this can be extended in the future by using Javatype as argument type of the *ArrT* type constructor. This will yield infinitely many types.

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

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

We need a function that widens *Arraytype* to *Javatype*.

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

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

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

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

$isclass\ (InterfaceT\ i) = False$

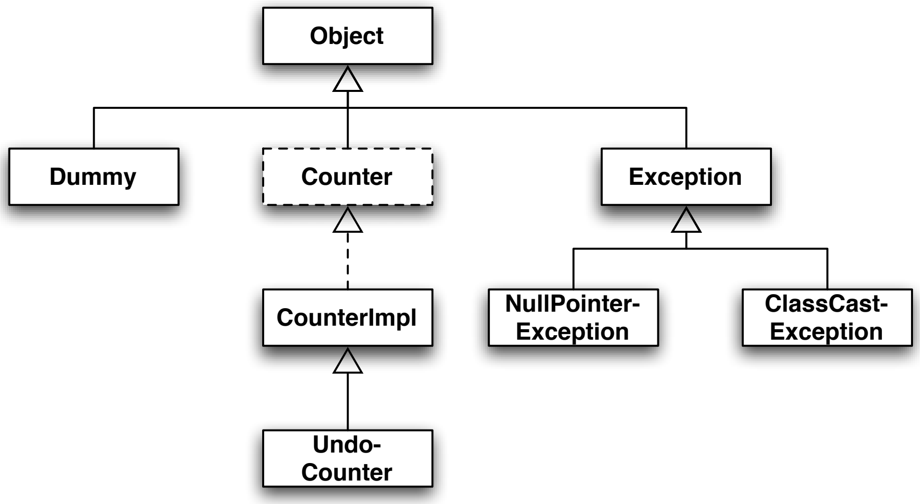
end

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*

(*CClassT UndoCounter*, *CClassT CounterImpl*) ∈ *direct-subtype*
(*CClassT CounterImpl*, *InterfaceT Counter*) ∈ *direct-subtype*
(*CClassT NullPointerException*, *CClassT Exception*) ∈ *direct-subtype*
(*CClassT ClassCastException*, *CClassT Exception*) ∈ *direct-subtype*

(*NullT*, *ArrT BoolAT*) ∈ *direct-subtype*
(*NullT*, *ArrT IntgAT*) ∈ *direct-subtype*
(*NullT*, *ArrT ShortAT*) ∈ *direct-subtype*
(*NullT*, *ArrT ByteAT*) ∈ *direct-subtype*
(*ArrT BoolAT*, *CClassT Object*) ∈ *direct-subtype*
(*ArrT IntgAT*, *CClassT Object*) ∈ *direct-subtype*
(*ArrT ShortAT*, *CClassT Object*) ∈ *direct-subtype*
(*ArrT ByteAT*, *CClassT Object*) ∈ *direct-subtype*

(*NullT*, *ArrT* (*AClassAT Dummy*)) ∈ *direct-subtype*
(*NullT*, *ArrT* (*CClassAT UndoCounter*)) ∈ *direct-subtype*
(*NullT*, *ArrT* (*CClassAT NullPointerException*)) ∈ *direct-subtype*
(*NullT*, *ArrT* (*CClassAT ClassCastException*)) ∈ *direct-subtype*

(*ArrT* (*CClassAT Object*), *CClassT Object*) ∈ *direct-subtype*

(*ArrT* (*AClassAT Dummy*), *ArrT* (*CClassAT Object*)) ∈ *direct-subtype*
(*ArrT* (*CClassAT CounterImpl*), *ArrT* (*InterfaceAT Counter*)) ∈ *direct-subtype*
(*ArrT* (*InterfaceAT Counter*), *ArrT* (*CClassAT Object*)) ∈ *direct-subtype*
(*ArrT* (*CClassAT Exception*), *ArrT* (*CClassAT Object*)) ∈ *direct-subtype*
(*ArrT* (*CClassAT UndoCounter*), *ArrT* (*CClassAT CounterImpl*)) ∈ *direct-subtype*
(*ArrT* (*CClassAT NullPointerException*), *ArrT* (*CClassAT Exception*)) ∈ *direct-subtype*
(*ArrT* (*CClassAT ClassCastException*), *ArrT* (*CClassAT Exception*)) ∈ *direct-subtype*
by (*simp-all add*: *direct-subtype-def*)

end

7 Widening the Direct Subtype Relation

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

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

7.1 Auxiliary lemmas

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

lemma *distinct-rtrancl-into-trancl*:
assumes *neq-x-y*: $x \neq y$
assumes *x-y-rtrancl*: $(x,y) \in r^*$
shows $(x,y) \in r^+$
using *x-y-rtrancl neq-x-y*
proof (*induct*)
assume $x \neq x$ **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
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
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)
```

qed
next
assume $(x,y) \in r^* \wedge x \neq y$
thus $(x,y) \in r^+$
by (*auto intro*: *distinct-rtrancl-into-trancl*)
qed

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
$A \prec 1\ B == (A,B) \in$ *direct-subtype*

abbreviation
widen-syntax :: *Javatype* ⇒ *Javatype* ⇒ *bool* (‹- ⪯ -› [71,71] 70)
where — reflexive transitive closure of direct subtype relation
$A \preceq B == (A,B) \in$ *direct-subtype**

abbreviation
widen-strict-syntax :: *Javatype* ⇒ *Javatype* ⇒ *bool* (‹- ≺ -› [71,71] 70)
where — transitive closure of direct subtype relation
$A \prec B == (A,B) \in$ *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** $x \prec x \Longrightarrow False$
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*: $C \prec D = (C \preceq D \wedge C \neq D)$
using *acyclic-direct-subtype* **by** (*rule acyclic-trancl-rtrancl*)

The widening relation on Javatype is reflexive.

lemma *widen-refl* [*iff*]: $X \preceq X$ **..**

The widening relation on Javatype is transitive.

lemma *widen-trans* [*trans*] :
  assumes *a-b*: $a \preceq b$
  shows $\bigwedge c.\ b \preceq c \Longrightarrow a \preceq c$
  by (*insert a-b*, *rule rtrancl-trans*)

The widening relation on Javatype is antisymmetric.

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

7.4 Javatype Ordering Properties

The type class *ord* allows us to overwrite the two comparison operators $<$ and $\leq$. 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 $\leq$ are defined in such a way that $(x < y) = (x \leq y \wedge x \neq y)$ holds. This proof can easily be achieved by using the lemmas proved above and the definition of *less-Javatype-def*.

instantiation *Javatype*:: *order*
begin

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

definition
  less-Javatype-def: $A < B \equiv A \leq B \wedge \neg\ B \leq (A{::}Javatype)$

instance proof
  fix $x\ y\ z$:: *Javatype*
  {
    show $x \leq x$
      by (*simp add*: *le-Javatype-def*)
  next
    assume $x \leq y$ $y \leq z$
    then show $x \leq z$
      by (*unfold le-Javatype-def*) (*rule rtrancl-trans*)
  next
    assume $x \leq y$ $y \leq x$
    then show $x = y$
      apply (*unfold le-Javatype-def*)
      apply (*rule widen-antisym*)
      apply *assumption* +
      done
  next
    show $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
      by (*simp add*: *less-Javatype-def*)
  }
qed

end

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*: $(a::Javatype) \leq b \Longrightarrow a \leq b$
by *simp*

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

```
ML ‹
local val ss = simpset-of @{context} in

fun widen-tac ctxt =
  resolve-tac ctxt @{thms widen-asm} THEN'
  simp-tac (put-simpset ss ctxt addsimps @{thms le-Javatype-def}) THEN'
  Method.insert-tac ctxt @{thms direct-subtype-widened} THEN'
  simp-tac (put-simpset (simpset-of @{theory-context Transitive-Closure}) ctxt)

end
›

declaration ‹fn - =>
  Simplifier.map-ss (fn ss => ss addSolver (mk-solver widen widen-tac))
›
```

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

7.6 Properties of the Subtype Relation

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

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

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

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

lemma *Object-root-array*: $ArrT\ C \leq CClassT\ Object$

proof (*cases C*)
  fix *x*
  assume *c*: $C = CClassAT\ x$
  show $ArrT\ C \leq CClassT\ Object$
    using *c* **by** (*cases x*, *simp-all*)
next
  fix *x*
  assume *c*: $C = AClassAT\ x$
  show $ArrT\ C \leq CClassT\ Object$
    using *c* **by** (*cases x*, *simp-all*)
next
  fix *x*
  assume *c*: $C = InterfaceAT\ x$
  show $ArrT\ C \leq CClassT\ Object$
    using *c* **by** (*cases x*, *simp-all*)
next
  assume *c*: $C = BoolAT$
  show $ArrT\ C \leq CClassT\ Object$
    using *c* **by** *simp*
next
  assume *c*: $C = IntgAT$
  show $ArrT\ C \leq CClassT\ Object$
    using *c* **by** *simp*
next
  assume *c*: $C = ShortAT$
  show $ArrT\ C \leq CClassT\ Object$
    using *c* **by** *simp*
next
  assume *c*: $C = ByteAT$
  show $ArrT\ C \leq CClassT\ Object$
    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*: $CClassT\ Object \leq c$
  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*]: $NullT \leq CClassT\ C$
  by (*cases C*, *simp-all*)

lemma *NullT-leaf-abs* [*simp*]: $NullT \leq AClassT\ C$
  by (*cases C*, *simp-all*)

lemma *NullT-leaf-int* [*simp*]: $NullT \leq InterfaceT\ C$
  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*

```
          | UndoCounter'save ⇒ IntgT
          | Dummy'dummy ⇒ NullT
          | Counter'dummy ⇒ NullT)
```

lemma *rtype-simps* [*simp*]:
rtype CounterImpl'value = IntgT
rtype UndoCounter'save = IntgT
rtype Dummy'dummy = NullT
rtype Counter'dummy = NullT
by (*simp-all add*: *rtype-def rtype-def rtype-def*)

With the datatype *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 *CounterImpl* cannot have a *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 *not-treach-ref-impl-not-reach* in theory StoreProperties. If the store provides locations for every combination of class name and field name, we cannot rule out reachability of certain pointer chains that go through “wrong” locations. That is why we decided to introduce the new type *CAttId*.

While *AttId* describes which fields are declared in which classes and interfaces, *CAttId* describes which objects of which classes may contain which fields at run-time. Thus, *CAttId* makes the inheritance of fields visible in the formalization.

There is only one such datatype because only objects of concrete classes can be created at run-time, thus only instance fields of concrete classes can occupy memory.

datatype *CAttId = CounterImpl'CounterImpl'value | UndoCounter'CounterImpl'value*
| UndoCounter'UndoCounter'save
| CounterImpl'Counter'dummy | UndoCounter'Counter'dummy

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

definition *catt*:: *CTypeId ⇒ AttId ⇒ CAttId* **where**

```
catt C f =
  (case C of
     CounterImpl ⇒ (case f of
                 CounterImpl'value ⇒ CounterImpl'CounterImpl'value
               | UndoCounter'save ⇒ undefined
               | Dummy'dummy ⇒ undefined
               | Counter'dummy ⇒ CounterImpl'Counter'dummy)
   | UndoCounter ⇒ (case f of
                 CounterImpl'value ⇒ UndoCounter'CounterImpl'value
               | UndoCounter'save ⇒ UndoCounter'UndoCounter'save
               | Dummy'dummy ⇒ undefined
               | Counter'dummy ⇒ UndoCounter'Counter'dummy)
   | Object ⇒ undefined
   | Exception ⇒ undefined
   | ClassCastException ⇒ undefined
   | NullPointerException ⇒ undefined
)
```

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

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

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

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

Selection of the field name.

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

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

end

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

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

The following lemmas are just a demonstration of simplification.

lemma *rtype-att-catt*:
  $CClassT\ c \leq dtype\ f \Longrightarrow rtype\ (att\ (catt\ c\ f)) = rtype\ f$
  by *simp*

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

end

10 Value

theory *Value* **imports** *Subtype* **begin**

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

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

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

The objects of each class are identified by a unique ID. We use elements of type *nat* here, but in general it is sufficient to use an infinite type with a successor function and a comparison predicate.

type-synonym *ObjectId* = *nat*

The definition of the datatype *Value*. Values can be of the Java types boolean, int, short and byte. Additionally, they can be an object reference, an array reference or the value null.

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

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

10.1 Discriminator Functions

To test values, we define the following discriminator functions.

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

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

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

lemma *isIntgV-simps* [*simp*]:
isIntgV (*boolV b*) = *False*
isIntgV (*intgV i*) = *True*
isIntgV (*shortV s*) = *False*

isIntgV (*byteV by*) = *False*
isIntgV (*objV C a*) = *False*
isIntgV (*arrV T a*) = *False*
isIntgV (*nullV*) = *False*
by (*simp-all add*: *isIntgV-def*)

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

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

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

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

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

| *byteV by* ⇒ *False*
| *objV C a* ⇒ *True*
| *arrV T a* ⇒ *True*
| *nullV* ⇒ *True*)

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

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

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

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

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

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


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

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

10.2 Selector Functions

```
definition aI :: Value ⇒ JavaInt where
aI v = (case v of
            boolV  b   ⇒ undefined
          | intgV  i   ⇒ i
          | shortV sh  ⇒ undefined
          | byteV  by  ⇒ undefined
          | objV   C a ⇒ undefined
          | arrV   T a  ⇒ undefined
          | nullV       ⇒ undefined)
lemma aI-simps [simp]:
aI (intgV i) = i
by (simp add: aI-def)


definition aB :: Value ⇒ JavaBoolean where
aB v = (case v of
            boolV  b   ⇒ b
          | intgV  i   ⇒ undefined
          | shortV sh  ⇒ undefined
          | byteV  by  ⇒ undefined
          | objV   C a ⇒ undefined
          | arrV   T a  ⇒ undefined
          | nullV       ⇒ undefined)
lemma aB-simps [simp]:
aB (boolV b) = b
by (simp add: aB-def)
```

definition *aSh* :: *Value* ⇒ *JavaShort* **where**
aSh v = (*case v of*
  boolV b ⇒ *undefined*
| *intgV i* ⇒ *undefined*
| *shortV sh* ⇒ *sh*
| *byteV by* ⇒ *undefined*
| *objV C a* ⇒ *undefined*
| *arrV T a* ⇒ *undefined*
| *nullV* ⇒ *undefined*)
lemma *aSh-simps* [*simp*]:
aSh (*shortV sh*) = *sh*
by (*simp add*: *aSh-def*)

definition *aBy* :: *Value* ⇒ *JavaByte* **where**
aBy v = (*case v of*
  boolV b ⇒ *undefined*
| *intgV i* ⇒ *undefined*
| *shortV s* ⇒ *undefined*
| *byteV by* ⇒ *by*
| *objV C a* ⇒ *undefined*
| *arrV T a* ⇒ *undefined*
| *nullV* ⇒ *undefined*)
lemma *aBy-simps* [*simp*]:
aBy (*byteV by*) = *by*
by (*simp add*: *aBy-def*)

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

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

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

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

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

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

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

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

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

10.3 Determining the Type of a Value

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

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

abbreviation *tau-syntax* :: *Value* ⇒ *Javatype* (‹τ -›)
  where τ *v* == *typeof v*

lemma *typeof-simps* [*simp*]:
(τ (*boolV b*)) = *BoolT*
(τ (*intgV i*)) = *IntgT*

(τ (*shortV sh*)) = *ShortT*
(τ (*byteV by*)) = *ByteT*
(τ (*objV c a*)) = *CClassT c*
(τ (*arrV t a*)) = *ArrT t*
(τ (*nullV*))   = *NullT*
  by (*simp-all add*: *typeof-def*)

10.4 Default Initialization Values for Types

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

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

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

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

```
  assume c: T = NullT
  show (τ (init T)) ≤ T
    using c by simp
next
  fix x
  assume c: T = CClassT x
  show (τ (init T)) ≤ T
    using c by (cases x, simp-all)
next
  fix x
  assume c: T = AClassT x
  show (τ (init T)) ≤ T
    using c by (cases x, simp-all)
next
  fix x
  assume c: T = InterfaceT x
  show (τ (init T)) ≤ T
    using c by (cases x, simp-all)
next
  fix x
  assume c: T = ArrT x
  show (τ (init T)) ≤ T
    using c
  proof (cases x)
    fix y
    assume c2: x = CClassAT y
    show (τ (init T)) ≤ T
      using c c2 by (cases y, simp-all)
  next
    fix y
    assume c2: x = AClassAT y
    show (τ (init T)) ≤ T
      using c c2 by (cases y, simp-all)
  next
    fix y
    assume c2: x = InterfaceAT y
    show (τ (init T)) ≤ T
      using c c2 by (cases y, simp-all)
  next
    assume c2: x = BoolAT
    show (τ (init T)) ≤ T
      using c c2 by simp
  next
    assume c2: x = IntgAT
    show (τ (init T)) ≤ T
      using c c2 by simp
  next
    assume c2: x = ShortAT
    show (τ (init T)) ≤ T
      using c c2 by simp
  next
    assume c2: x = ByteAT
    show (τ (init T)) ≤ T
      using c c2 by simp
```

qed
qed

end

11 Location

theory *Location*
imports *AttributesIndep ../Isabelle/Value*
begin

A storage location can be a field of an object, a static field, the length of an array, or the contents of an array.

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

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

The function *ltype* yields the content type of a location.

definition *ltype*:: *Location* ⇒ *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*)

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.

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*)

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 cff 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 *objLoc* location or a *staticLoc* location. But this will only complicate the actual proofs and we can already easily perform the distinction whether a field is static or not in the JIVE-frontend and therefore keep the verification simpler.

lemma *ref-loc* [*simp*]: $[\![isObjV\ r;\ typeof\ r \leq dtype\ f]\!] \Longrightarrow ref\ (r..f) = r$
  apply (*case-tac r*)
  apply (*case-tac* [!] *f*)
  apply (*simp-all*)
  done

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

lemma *obj-arr-len* [*simp*]: $isArrV\ r \Longrightarrow ref\ (arr\text{-}len\ r) = r$
  by (*cases r*) *simp-all*

end

12 Store

theory *Store*
imports *Location*
begin

12.1 New

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

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

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

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

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

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

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

lemma *typeofNew-simps*:
typeofNew (*new-instance C*) = *CClassT C*
typeofNew (*new-array T l*) = *ArrT T*
  by (*simp-all add*: *typeofNew-def*)

12.2 The Definition of the Store

In our store model, all objects[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 *newOID*) and arrays (function *newAID*) as well as a function that maps locations to their contents (function *vals*).

record *StoreImpl* = *newOID* :: *CTypeId* ⇒ *ObjectId*
newAID :: *Arraytype* ⇒ *ObjectId*
vals :: *Location* ⇒ *Value*

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

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*)

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).

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*)}

typedef *Store* = *Store*
unfolding *Store-def*
apply (*rule exI* [**where** *?x*=⦇ *newOID* = (λ*C*. *0*),
newAID = (λ*T*. *0*),
vals = (λ*l*. *init* (*ltype l*)) ⦈])
apply (*auto simp add*: *aliveImpl-def init-def NullT-leaf-array split*: *Javatype.splits*)
done

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

[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.

```
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:: [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 *s⟨new-instance Node, x := y, z := intgV 3, new-array IntgAT 3, a.[i] := intgV 4, k := boolV True⟩*.

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

```
overloading alive ≡ alive
begin
  definition alive where alive x s ≡ aliveImpl x (Rep-Store s)
end

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

overloading
  access ≡ access
  update ≡ update
  alloc ≡ alloc
  new ≡ new
begin

definition access
  where access s l ≡ vals (Rep-Store s) l
```

definition *update*
 where *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

definition *alloc*
 where *alloc s t* ≡
 (case t of
 new-instance C
 ⇒ *Abs-Store*
 ((Rep-Store s)(|newOID := λ D. if C=D
 then Suc (newOID (Rep-Store s) C)
 else newOID (Rep-Store s) D|))
 | *new-array T l*
 ⇒ *Abs-Store*
 ((Rep-Store s)(|newAID := λ S. if T=S
 then Suc (newAID (Rep-Store s) T)
 else newAID (Rep-Store s) S,
 vals := (vals (Rep-Store s))
 (arrLenLoc T (newAID (Rep-Store s) T)
 := intgV (int l))|)))

definition *new*
 where *new s t* ≡
 (case t of
 new-instance C ⇒ *objV C (newOID (Rep-Store s) C)*
 | *new-array T l* ⇒ *arrV T (newAID (Rep-Store s) T))*

end

The predicate *wts* tests whether the store is well-typed.

definition
wts :: Store ⇒ *bool* **where**
wts OS = (∀ *(l::Location) . (typeof (OS@@l))* ≤ *(ltype l))*

12.4 Derived Properties of the Store

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

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

lemma *alive-loc [simp]:*
 ⟦*isObjV x; typeof x* ≤ *dtype f*⟧ ⟹ *alive (ref (x..f)) s = alive x s*
 by *(cases x) (simp-all)*

lemma *alive-arr-loc [simp]:*
 isArrV x ⟹ *alive (ref (x.[i])) s = alive x s*
 by *(cases x) (simp-all)*

lemma *alive-arr-len [simp]:*

isArrV x ⟹ *alive* (*ref* (*arr-len x*)) *s* = *alive x s*
by (*cases x*) (*simp-all*)

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

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

lemma *ref-loc-new* [*simp*]: *CClassT C* ≤ *dtype f*
⟹ *ref* ((*new s* (*new-instance C*))..*f*) = *new s* (*new-instance C*)
by (*simp add*: *new-def*)

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

The store is well-typed by construction.

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

Store8

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

Store3

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

lemma *update-induct*:
assumes *skip*: *P s*
assumes *update*: ⟦*alive* (*ref l*) *s*; *alive v s*; *typeof v* ≤ *ltype l*⟧ ⟹
P (*Abs-Store* ((*Rep-Store s*)⦇*vals*:=(*vals* (*Rep-Store s*))(*l*:=*v*)⦈))
shows *P* (*s*⟨*l*:=*v*⟩)

```
  using update skip
  by (simp add: update-def)

lemma vals-update-in-Store:
  assumes alive-l: alive (ref l) s
  assumes alive-y: alive y s
  assumes type-conform: typeof y ≤ ltype l
  shows (Rep-Store s(|vals := (vals (Rep-Store s))(l := y)|)) ∈ Store
  (is ?s-upd ∈ Store)
proof −
  have s: Rep-Store s ∈ Store
    by (rule Rep-Store)
  have alloc-eq: newOID ?s-upd = newOID (Rep-Store s)
    by simp
  have ∀ l. aliveImpl (vals ?s-upd l) ?s-upd
  proof
    fix k
    show aliveImpl (vals ?s-upd k) ?s-upd
    proof (cases k=l)
      case True
      with alive-y show ?thesis
        by (simp add: alloc-eq alive-def aliveImpl-def split: Value.splits)
    next
      case False
      from s have ∀ l. aliveImpl (vals (Rep-Store s) l) (Rep-Store s)
        by (simp add: Store-def)
      with False show ?thesis
        by (simp add: aliveImpl-def split: Value.splits)
    qed
  qed
  moreover
  have ∀ l. ¬ aliveImpl (ref l) ?s-upd ⟶ vals ?s-upd l = init (ltype l)
  proof (intro allI impI)
    fix k
    assume unalive: ¬ aliveImpl (ref k) ?s-upd
    show vals ?s-upd k = init (ltype k)
    proof −
      from unalive alive-l
      have k≠l
        by (auto simp add: alive-def aliveImpl-def split: Value.splits)
      hence vals ?s-upd k = vals (Rep-Store s) k
        by simp
      moreover from unalive
      have ¬ aliveImpl (ref k) (Rep-Store s)
        by (simp add: aliveImpl-def split: Value.splits)
      ultimately show ?thesis
        using s by (simp add: Store-def)
    qed
  qed
  moreover
  have ∀ l. typeof (vals ?s-upd l) ≤ ltype l
  proof
    fix k show typeof (vals ?s-upd k) ≤ ltype k
    proof (cases k=l)
```

case *True*
with *type-conform* **show** *?thesis*
by *simp*
next
case *False*
hence *vals ?s-upd k = vals (Rep-Store s) k*
by *simp*
with *s* **show** *?thesis*
by (*simp add*: *Store-def*)
qed
qed
ultimately show *?thesis*
by (*simp add*: *Store-def*)
qed

Store6

lemma *alive-update-invariant* [*simp*]: *alive x* (*s*⟨*l*:=*y*⟩) = *alive x s*
proof (*rule update-induct*)
show *alive x s* = *alive x s*..
next
assume *alive* (*ref l*) *s alive y s typeof y* ≤ *ltype l*
hence *Rep-Store*
(*Abs-Store* (*Rep-Store s*⦇*vals* := (*vals* (*Rep-Store s*))(*l* := *y*)⦈))
= *Rep-Store s*⦇*vals* := (*vals* (*Rep-Store s*))(*l* := *y*)⦈
by (*rule vals-update-in-Store* [*THEN Abs-Store-inverse*])
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

Store1

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

Store2

lemma *update-access-same* [*simp*]:
assumes *alive-l*: *alive* (*ref l*) *s*
assumes *alive-x*: *alive x s*
assumes *widen-x-l*: *typeof x* ≤ *ltype l*

shows $s\langle l{:=}x\rangle@@l = x$
proof −
from *alive-l alive-x widen-x-l*
have *Rep-Store*
$(Abs\text{-}Store\ (Rep\text{-}Store\ s(|vals := (vals\ (Rep\text{-}Store\ s))(l := x)|)))$
$= Rep\text{-}Store\ s(|vals := (vals\ (Rep\text{-}Store\ s))(l := x)|)$
by (*rule vals-update-in-Store* [*THEN Abs-Store-inverse*])
hence $Abs\text{-}Store\ (Rep\text{-}Store\ s(|vals := (vals\ (Rep\text{-}Store\ s))(l := x)|))@@l = x$
by (*simp add*: *access-def*)
with *alive-l alive-x widen-x-l*
show *?thesis*
by (*simp add*: *update-def*)
qed

Store4

lemma *update-unalive-val* [*simp*,*intro*]: $\neg\ alive\ x\ s \Longrightarrow s\langle l{:=}x\rangle = s$
by (*simp add*: *update-def*)

lemma *update-unalive-loc* [*simp*,*intro*]: $\neg\ alive\ (ref\ l)\ s \Longrightarrow s\langle l{:=}x\rangle = s$
by (*simp add*: *update-def*)

lemma *update-type-mismatch* [*simp*,*intro*]: $\neg\ typeof\ x \le ltype\ l \Longrightarrow s\langle l{:=}x\rangle = s$
by (*simp add*: *update-def*)

Store9

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

Store10

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

lemma *alloc-new-instance-in-Store*:
$(Rep\text{-}Store\ s(|newOID := \lambda D.\ if\ C = D$
$then\ Suc\ (newOID\ (Rep\text{-}Store\ s)\ C)$
$else\ newOID\ (Rep\text{-}Store\ s)\ D|)) \in Store$
(**is** *?s-alloc* $\in$ *Store*)
proof −
have *s*: $Rep\text{-}Store\ s \in Store$
by (*rule Rep-Store*)
hence $\forall\ l.\ aliveImpl\ (vals\ (Rep\text{-}Store\ s)\ l)\ (Rep\text{-}Store\ s)$
by (*simp add*: *Store-def*)
then
have $\forall\ l.\ aliveImpl\ (vals\ ?s\text{-}alloc\ l)\ ?s\text{-}alloc$
by (*auto intro*: *less-SucI simp add*: *aliveImpl-def split*: *Value.splits*)
moreover
have $\forall\ l.\ \neg\ aliveImpl\ (ref\ l)\ ?s\text{-}alloc \longrightarrow vals\ ?s\text{-}alloc\ l = init\ (ltype\ l)$
proof (*intro allI impI*)
fix *l*
assume $\neg\ aliveImpl\ (ref\ l)\ ?s\text{-}alloc$
hence $\neg\ aliveImpl\ (ref\ l)\ (Rep\text{-}Store\ s)$
by (*simp add*: *aliveImpl-def split*: *Value.splits if-split-asm*)
with *s* **have** $vals\ (Rep\text{-}Store\ s)\ l = init\ (ltype\ l)$

```
    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 :=
                 λS. if T = S
                     then Suc (newAID (Rep-Store s) T)
                     else newAID (Rep-Store s) S,
               vals := (vals (Rep-Store s))
                       (arrLenLoc T
                         (newAID (Rep-Store s) T) :=
                         intgV (int n))|)) ∈ Store
(is ?s-alloc ∈ Store)
proof −
  have s: Rep-Store s ∈ Store
    by (rule Rep-Store)
  have ∀ l. aliveImpl (vals ?s-alloc l) ?s-alloc
  proof
    fix l show aliveImpl (vals ?s-alloc l) ?s-alloc
    proof (cases l = arrLenLoc T (newAID (Rep-Store s) T))
      case True
      thus ?thesis
        by (simp add: 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 (auto intro: less-SucI simp add: aliveImpl-def split: Value.splits)
    qed
  qed
  moreover
  have ∀ l. ¬ aliveImpl (ref l) ?s-alloc ⟶ vals ?s-alloc l = init (ltype l)
  proof (intro allI impI)
    fix l
    assume unalive: ¬ aliveImpl (ref l) ?s-alloc
    show vals ?s-alloc l = init (ltype l)
    proof (cases l = arrLenLoc T (newAID (Rep-Store s) T))
      case True
      with unalive show ?thesis by (simp add: aliveImpl-def)
    next
      case False
      from unalive
      have ¬ aliveImpl (ref l) (Rep-Store s)
        by (simp add: aliveImpl-def split: Value.splits if-split-asm)
```

```
      with s have vals (Rep-Store s) l = init (ltype l)
        by (simp add: Store-def)
      with False show ?thesis
        by simp
    qed
  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 new-alive-alloc [simp,intro]: alive (new s t) (s⟨t⟩)
proof (cases t)
  case new-instance thus ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
                  alloc-new-instance-in-Store [THEN Abs-Store-inverse])
next
  case new-array thus ?thesis
    by (simp add: alive-def aliveImpl-def new-def alloc-def
                  alloc-new-array-in-Store [THEN Abs-Store-inverse])
qed
```

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

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

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

```
lemma le-Suc-eq: (∀ a. (a < Suc n) = (a < Suc m)) = (∀ a. (a < n) = (a < m))
 (is (∀ a. ?A a) = (∀ a. ?B a))
proof
  assume ∀ a. ?A a thus ∀ a. ?B a
    by fastforce
next
  assume B: ∀ a. ?B a
  show ∀ a. ?A a
  proof
    fix a
    from B show ?A a
      by (cases a) simp-all
  qed
qed

lemma all-le-eq-imp-eq: ⋀ c::nat. (∀ a. (a < d) = (a < c)) ⟶ (d = c)
proof (induct d)
  case 0 thus ?case by fastforce
next
  case (Suc n c)
  thus ?case
    by (cases c) (auto simp add: le-Suc-eq)
qed

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

Store11

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

Store12

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

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

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

```
    (simp-all add: alive-def aliveImpl-def alloc-def
                alloc-new-array-in-Store [THEN Abs-Store-inverse])
qed
```

Store7

```
lemma alive-alloc-exhaust: alive x (s⟨t⟩) = (alive x s ∨ (x = new s t))
proof
  assume alive-alloc: alive x (s⟨t⟩)
  show alive x s ∨ x = new s t
  proof (cases t)
    case (new-instance C)
    with alive-alloc show ?thesis
      by (cases x) (auto split: if-split-asm
                    simp add: alive-def new-def alloc-def aliveImpl-def
                       alloc-new-instance-in-Store [THEN Abs-Store-inverse])
  next
    case (new-array T l)
    with alive-alloc show ?thesis
      by (cases x) (auto split: if-split-asm
                    simp add: alive-def new-def alloc-def aliveImpl-def
                    alloc-new-array-in-Store [THEN Abs-Store-inverse])
  qed
next
  assume alive x s ∨ x = new s t
  then show alive x (s⟨t⟩)
  proof
    assume alive x 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 ⟹ P; x=new s t ⟹ P⟧
   ⟹ P
  by (auto simp add: alive-alloc-exhaust)

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

lemma access-arr-len-new-alloc [simp]:
  s⟨new-array T l⟩@@arr-len (new s (new-array T l)) = intgV (int l)
  by (subst access-def)
    (simp add: new-def alloc-def alive-def
            alloc-new-array-in-Store [THEN Abs-Store-inverse] access-def)
```

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

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

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

```
      have s⟨t⟩@@l = init (ltype l)
        by (simp add: access-new)
      moreover
      from ref-new have s@@l = init (ltype l)
        by simp
      ultimately
      show ?thesis by simp
    qed
  next
    case False
    hence s⟨t⟩@@l = init (ltype l)
      by (simp)
    moreover
    from False have ¬ alive (ref l) s
      by (auto simp add: alive-alloc-propagation)
    hence s@@l = init (ltype l)
      by simp
    ultimately show ?thesis by simp
  qed
qed
```

Store13

```
lemma Store-eqI:
  assumes eq-alive: ∀ x. alive x s1 = alive x s2
  assumes eq-access: ∀ 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 ∃ C. newOID (Rep-Store s1) C ≠ newOID (Rep-Store s2) C
    proof (rule ccontr)
      assume ¬ (∃ C. newOID (Rep-Store s1) C ≠ 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 ≠ newOID (Rep-Store s2) C
            ∀ 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 ∃ T. newAID (Rep-Store s1) T ≠ newAID (Rep-Store s2) T
      proof (rule ccontr)
        assume ¬ (∃ T. newAID (Rep-Store s1) T ≠ 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 ≠ newAID (Rep-Store s2) T
         ∀ 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 (Rep-Store s1) = (Rep-Store s2)
        by (cases Rep-Store s1,cases Rep-Store s2) simp
      hence s1=s2
        by (simp add: Rep-Store-inject)
      with neq-s1-s2 show ?thesis
        by simp
    next
      case False
      have ∃ l. vals (Rep-Store s1) l ≠ vals (Rep-Store s2) l
      proof (rule ccontr)
        assume ¬ (∃ l. vals (Rep-Store s1) l ≠ vals (Rep-Store s2) l)
        hence vals (Rep-Store s1) = vals (Rep-Store s2)
          by (blast intro: ext)
        with False show False ..
      qed
      then obtain l
        where vals (Rep-Store s1) l ≠ vals (Rep-Store s2) l
        by auto
      with eq-access have False
        by (simp add: access-def)
      thus ?thesis ..
    qed
  qed
 qed
qed
```

Lemma 3.1 in [Poetzsch-Heffter97]. The proof of this lemma is quite an impressive demostration of readable Isar proofs since it closely follows the textual proof.

```
lemma comm:
  assumes neq-l-new: ref l ≠ new s t
  assumes neq-x-new: x ≠ new s t
  shows s⟨t⟩⟨l:=x⟩ = s⟨l:=x⟩⟨t⟩
proof (rule Store-eqI [rule-format])
  fix y
  show alive y (s⟨t⟩⟨l:=x⟩) = alive y (s⟨l:=x⟩⟨t⟩)
  proof −
    have alive y (s⟨t⟩⟨l:=x⟩) = alive y (s⟨t⟩)
      by (rule alive-update-invariant)
    also have ... = (alive y s ∨ (y = new s t))
      by (rule alive-alloc-exhaust)
```

```
    also have ... = (alive y (s⟨l:=x⟩) ∨ y = new s t)
      by (simp only: alive-update-invariant)
    also have ... = (alive y (s⟨l:=x⟩) ∨ y = new (s⟨l:=x⟩) t)
    proof –
      have new s t = new (s⟨l:=x⟩) t
        by simp
      thus ?thesis by simp
    qed
    also have ... = alive y (s⟨l:=x⟩⟨t⟩)
      by (simp add: alive-alloc-exhaust)
    finally show ?thesis .
  qed
next
  fix k
  show s⟨t⟩⟨l := x⟩@@k = s⟨l := x⟩⟨t⟩@@k
  proof (cases l=k)
    case False note neq-l-k = this
    show ?thesis
    proof (cases isNewArr t ⟶ k ≠ 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 ... = s@@k by simp
      also from neq-l-k
      have ... = s⟨l:=x⟩@@k by simp
      also from True
      have ... = s⟨l := x⟩⟨t⟩@@k by simp
      finally show ?thesis .
    next
      case False
      then obtain T n where
        t: t=new-array T n and k: k=arr-len (new s (new-array T n))
        by (cases t) auto
      from k have k': k=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 ... = intgV (int n)
        by simp
      also from t k'
      have ... = s⟨l := x⟩⟨t⟩@@k
        by (simp del: new-update)
      finally show ?thesis .
    qed
  next
    case True note eq-l-k = this
    have lemma-3-1:
      ref l ≠ new s t ⟹ alive (ref l) (s⟨t⟩) = alive (ref l) s
      by (simp add: alive-alloc-exhaust)
    have lemma-3-2:
      x ≠ new s t ⟹ 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 ref l ≠ new (s⟨l:=x⟩) t
    by simp
  hence isNewArr t ⟶ l ≠ arr-len (new (s⟨l:=x⟩) t)
    by (cases t) auto
  thus ?thesis
    by (simp)
qed
show ?thesis
proof (cases alive x s)
  case True note alive-x = this
  show ?thesis
  proof (cases alive (ref l) s)
    case True note alive-l = this
    show ?thesis
    proof (cases typeof x ≤ 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
      have isNewArr t ⟶ l ≠ arr-len (new s t)
        by (cases t) auto
      thus ?thesis
        by (simp)
    qed
    also from not-alive-x have ... = s⟨l:=x⟩@@l
      by (simp)
    also have ... = s⟨l:=x⟩⟨t⟩@@l
      by (simp add: lemma-3-3)
    finally show ?thesis by (simp add: eq-l-k)
  qed
 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 ⇒ Location ⇒ Value ⇒ bool
   (‹-⊢ - reachable′-from -› [91,91,91]90)
  for s :: Store
where
  Immediate: ref l ≠ nullV ⟹ s⊢ l reachable-from (ref l)
| Indirect: ⟦s⊢ l reachable-from (s@@k); ref k ≠ nullV⟧
            ⟹ s⊢ l 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⊢ l reachable-from x shows x≠nullV
  using reach by (induct) auto

corollary unreachable-Null-simp [simp]:
  ¬ s⊢ l reachable-from nullV
  by (iprover dest: unreachable-Null)

corollary unreachable-NullE [elim]:
  s⊢ l reachable-from nullV ⟹ P
  by (simp)

lemma reachObjLoc [simp,intro]:
  C=cls cf ⟹ s⊢ objLoc cf a reachable-from objV C a
  by (iprover intro: reach.Immediate [of objLoc cf a,simplified])

lemma reachArrLoc [simp,intro]: s⊢ arrLoc T a i reachable-from arrV T a
  by (rule reach.Immediate [of arrLoc T a i,simplified])

lemma reachArrLen [simp,intro]: s⊢ arrLenLoc T a reachable-from arrV T a
  by (rule reach.Immediate [of arrLenLoc T a,simplified])

lemma unreachStatic [simp]: ¬ s⊢ staticLoc f reachable-from x
proof −
  {
    fix y assume s⊢ y reachable-from x y=staticLoc f
    then have False
      by induct auto
  }
  thus ?thesis
    by auto
qed

lemma unreachStaticE [elim]: s⊢ staticLoc f reachable-from x ⟹ P
  by (simp add: unreachStatic)

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

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

Lemma 3.2 (i)

```
lemma reach-update [simp]:
  assumes unreachable-l-x: ¬ s⊢ l reachable-from x
  shows s⟨l:=y⟩⊢ k reachable-from  x = s⊢ k reachable-from x
proof
```

```
  assume s⊢ k reachable-from x
  from this unreachable-l-x
  show s⟨l := y⟩⊢ k reachable-from x
  proof (induct)
    case (Immediate k)
    have ref k ≠ nullV by fact
    then show s⟨l := y⟩⊢ k reachable-from (ref k)
      by (rule reach.Immediate)
  next
    case (Indirect k m)
    have hyp: ¬ s⊢ l reachable-from (s@@m)
              ⟹ s⟨l:=y⟩ ⊢ k reachable-from (s@@m) by fact
    have ref m ≠ nullV and ¬ s⊢ l reachable-from (ref m) by fact+
    hence l≠m ¬ s⊢ l reachable-from (s@@m)
      by (auto intro: reach.intros)
    with hyp have s⟨l := y⟩ ⊢ k reachable-from (s⟨l := y⟩@@m)
      by simp
    then show s⟨l := y⟩⊢ k reachable-from (ref m)
      by (rule reach.Indirect) (rule Indirect.hyps)
  qed
next
  assume s⟨l := y⟩⊢ k reachable-from x
  from this unreachable-l-x
  show s⊢ k reachable-from x
  proof (induct)
    case (Immediate k)
    have ref k ≠ nullV by fact
    then show s ⊢ k reachable-from (ref k)
      by (rule reach.Immediate)
  next
    case (Indirect k m)
    with Indirect.hyps
    have hyp: ¬ s⊢ l reachable-from (s⟨l := y⟩@@m)
              ⟹ s⊢ k reachable-from (s⟨l := y⟩@@m) by simp
    have ref m ≠ nullV and ¬ s⊢ l reachable-from (ref m) by fact+
    hence l≠m ¬ s ⊢ l reachable-from (s@@m)
      by (auto intro: reach.intros)
    with hyp have s ⊢ k reachable-from (s@@m)
      by simp
    thus s⊢ k reachable-from (ref m)
      by (rule reach.Indirect) (rule Indirect.hyps)
  qed
qed
```

Lemma 3.2 (ii)

```
lemma reach2:
  ¬ s⊢ l reachable-from x ⟹ ¬ s⟨l:=y⟩⊢ l reachable-from x
  by (simp)
```

Lemma 3.2 (iv)

```
lemma reach4: ¬ s ⊢ l reachable-from (ref k) ⟹ k ≠ l ∨ (ref k) = nullV
  by (auto intro: reach.intros)

lemma reachable-isRef:
```

```
  assumes reach: s⊢l 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


lemma val-ArrLen-IntgT: isArrLenLoc l ⟹ typeof (s@@l) = IntgT
proof −
  assume isArrLen: isArrLenLoc l
  have T: typeof (s@@l) ≤ ltype l
    by (simp)
  also from isArrLen have I: ltype l = 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: ¬ isArrLenLoc l
  shows s⟨t⟩@@l = s@@l
proof −
  from no-arr-len
  have isNewArr t ⟶ l ≠ 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⟩⊢ l reachable-from x = s⊢ l reachable-from x
proof
  assume s⟨t⟩⊢ l reachable-from x
  thus s⊢ l reachable-from x
  proof (induct)
    case (Immediate l)
    thus s⊢ l reachable-from ref l
      by (rule reach.intros)
  next
    case (Indirect l k)
    have reach-k: s⊢ l reachable-from (s⟨t⟩@@k) by fact
    moreover
    have s⟨t⟩@@k = s@@k
    proof −
      from reach-k have isRef: isRefV (s⟨t⟩@@k)
        by (rule reachable-isRef)
      have ¬ isArrLenLoc k
```

```
      proof (rule ccontr,simp)
        assume isArrLenLoc k
        then have typeof (s⟨t⟩@@k) = IntgT
          by (rule val-ArrLen-IntgT)
        with isRef
        show False
          by (cases (s⟨t⟩@@k)) simp-all
      qed
      thus ?thesis
        by (rule access-alloc')
    qed
    ultimately have s⊢ l reachable-from (s@@k)
      by simp
    thus s⊢ l reachable-from ref k
      by (rule reach.intros) (rule Indirect.hyps)
  qed
next
  assume s⊢ l reachable-from x
  thus s⟨t⟩⊢ l reachable-from x
  proof (induct)
    case (Immediate l)
    thus s⟨t⟩⊢ l reachable-from ref l
      by (rule reach.intros)
  next
    case (Indirect l k)
    have reach-k: s⟨t⟩⊢ l reachable-from (s@@k) by fact
    moreover
    have s⟨t⟩@@k = s@@k
    proof −
      from reach-k have isRef: isRefV (s@@k)
        by (rule reachable-isRef)
      have ¬ isArrLenLoc 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 ?thesis
        by (rule access-alloc')
    qed
    ultimately have s⟨t⟩⊢ l reachable-from (s⟨t⟩@@k)
      by simp
    thus s⟨t⟩⊢ l reachable-from ref k
      by (rule reach.intros) (rule Indirect.hyps)
  qed
qed
```

Lemma 3.2 (vi)

```
lemma reach6: isprimitive(typeof x) ⟹ ¬ s ⊢ l reachable-from x
proof
  assume prim: isprimitive(typeof x)
```

```
  assume s ⊢ l reachable-from x
  hence isRefV x
    by (rule reachable-isRef)
  with prim show False
    by (cases x) simp-all
qed
```

Lemma 3.2 (iii)

```
lemma reach3:
  assumes k-y: ¬ s⊢ k reachable-from y
  assumes k-x: ¬ s⊢ k reachable-from x
  shows ¬ s⟨l:=y⟩⊢ k reachable-from x
proof
  assume s⟨l:=y⟩⊢ k reachable-from x
  from this k-y k-x
  show False
  proof (induct)
    case (Immediate l)
    have ¬ s⊢ l reachable-from ref l and ref l ≠ nullV by fact+
    thus False
      by (iprover intro: reach.intros)
  next
    case (Indirect m k)
    have k-not-Null: ref k ≠ nullV by fact
    have not-m-y: ¬ s⊢ m reachable-from y by fact
    have not-m-k: ¬ s⊢ m reachable-from ref k by fact
    have hyp: ⟦¬ s⊢ m reachable-from y; ¬ s⊢ m reachable-from (s⟨l := y⟩@@k)⟧
               ⟹ False by fact
    have m-upd-k: s⟨l := y⟩⊢ m reachable-from (s⟨l := y⟩@@k) by fact
    show False
    proof (cases l=k)
      case False
      then have s⟨l := y⟩@@k = s@@k by simp
      moreover
      from not-m-k k-not-Null have ¬ s⊢ m reachable-from (s@@k)
        by (iprover intro: reach.intros)
      ultimately show False
        using not-m-y hyp by simp
    next
      case True note eq-l-k = this
      show ?thesis
      proof (cases alive (ref l) s ∧ alive y s ∧ typeof y ≤ ltype l)
        case True
        with eq-l-k have s⟨l := y⟩@@k = y
          by simp
        with not-m-y hyp show False by simp
      next
        case False
        hence s⟨l := y⟩ = s
          by auto
        moreover
        from not-m-k k-not-Null have ¬ s⊢ m reachable-from (s@@k)
          by (iprover intro: reach.intros)
        ultimately show False
```

```
        using not-m-y hyp by simp
      qed
    qed
  qed
qed
```

Lemma 3.2 (vii).

```
lemma unreachable-from-init [simp,intro]: ¬ s⊢ l reachable-from (init T)
  using reach6 by (cases T) simp-all

lemma ref-reach-unalive:
  assumes unalive-x:¬ alive x s
  assumes l-x: s⊢ l reachable-from x
  shows x = ref l
using l-x unalive-x
proof induct
  case (Immediate l)
  show ref l = ref l
    by simp
next
  case (Indirect l k)
  have ref k ≠ nullV by fact
  have ¬ alive (ref k) s by fact
  hence s@@k = init (ltype k) by simp
  moreover have s⊢ l reachable-from (s@@k) by fact
  ultimately have False by simp
  thus ?case ..
qed

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

Lemma 3.2 (viii)

```
lemma alive-reach-alive:
  assumes alive-x: alive x s
  assumes reach-l: s ⊢ l reachable-from x
```

```
  shows alive (ref l) s
using reach-l alive-x
proof (induct)
  case (Immediate l)
  show ?case by fact
next
  case (Indirect l k)
  have hyp: alive (s@@k) s ⟹ alive (ref l) s by fact
  moreover have alive (s@@k) s by simp
  ultimately
  show alive (ref l) s
    by iprover
qed
```

Lemma 3.2 (ix)

```
lemma reach9:
  assumes reach-impl-access-eq: ∀ l. s1⊢l reachable-from x ⟶ (s1@@l = s2@@l)
  shows s1⊢ l reachable-from x = s2⊢ l reachable-from x
proof
  assume s1⊢ l reachable-from x
  from this reach-impl-access-eq
  show s2⊢ l reachable-from x
  proof (induct)
    case (Immediate l)
    show s2⊢ l reachable-from ref l
      by (rule reach.intros) (rule Immediate.hyps)
  next
    case (Indirect l k)
    have hyp: ∀ l. s1⊢ l reachable-from (s1@@k) ⟶ s1@@l = s2@@l
              ⟹ s2⊢ l reachable-from (s1@@k) by fact
    have k-not-Null: ref k ≠ nullV by fact
    have reach-impl-access-eq:
      ∀ l. s1⊢ l reachable-from ref k ⟶ s1@@l = s2@@l by fact
    have s1⊢ l 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 ∀ l. s1⊢ l reachable-from (s1@@k) ⟶ s1@@l = s2@@l
      by (iprover intro: reach.intros)
    then have s2⊢ l reachable-from (s1@@k)
      by (rule hyp)
    ultimately have s2⊢ l reachable-from (s2@@k)
      by simp
    thus s2⊢ l reachable-from ref k
      by (rule reach.intros) (rule Indirect.hyps)
  qed
next
  assume s2⊢ l reachable-from x
  from this reach-impl-access-eq
  show s1⊢ l reachable-from x
  proof (induct)
    case (Immediate l)
    show s1⊢ l reachable-from ref l
```

by (*rule reach.intros*) (*rule Immediate.hyps*)
next
case (*Indirect l k*)
have *hyp*: ∀ *l. s1*⊢ *l reachable-from* (*s2*@@*k*) ⟶ *s1*@@*l* = *s2*@@*l*
⟹ *s1*⊢ *l reachable-from* (*s2*@@*k*) **by** *fact*
have *k-not-Null*: *ref k* ≠ *nullV* **by** *fact*
have *reach-impl-access-eq*:
∀ *l. s1*⊢ *l reachable-from ref k* ⟶ *s1*@@*l* = *s2*@@*l* **by** *fact*
have *s1*⊢ *k reachable-from ref k*
by (*rule reach.intros*) (*rule Indirect.hyps*)
with *reach-impl-access-eq*
have *eq-k*: *s1*@@*k* = *s2*@@*k*
by *simp*
from *reach-impl-access-eq k-not-Null*
have ∀ *l. s1*⊢ *l reachable-from* (*s1*@@*k*) ⟶ *s1*@@*l* = *s2*@@*l*
by (*iprover intro*: *reach.intros*)
then
have ∀ *l. s1*⊢ *l reachable-from* (*s2*@@*k*) ⟶ *s1*@@*l* = *s2*@@*l*
by (*simp add*: *eq-k*)
with *eq-k hyp* **have** *s1*⊢ *l reachable-from* (*s1*@@*k*)
by *simp*
thus *s1*⊢ *l reachable-from ref k*
by (*rule reach.intros*) (*rule Indirect.hyps*)
qed
qed

13.2 Reachability of a Reference from a Reference

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

definition *rreach*:: *Store* ⇒ *Value* ⇒ *Value* ⇒ *bool*
(‹-⊢*Ref* - *reachable′-from* -› [*91*,*91*,*91*]*90*) **where**
s⊢*Ref y reachable-from x* = (∃ *l. s*⊢ *l reachable-from x* ∧ *y* = *ref l*)

13.3 Disjointness of Reachable Locations

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

definition *disj*:: *Value* ⇒ *Value* ⇒ *Store* ⇒ *bool* **where**
disj x y s = (∀ *l.* ¬ *s*⊢ *l reachable-from x* ∨ ¬ *s*⊢ *l reachable-from y*)

lemma *disjI1*: ⟦⋀ *l. s*⊢ *l reachable-from x* ⟹ ¬ *s*⊢ *l reachable-from y*⟧
⟹ *disj x y s*
by (*simp add*: *disj-def*)

lemma *disjI2*: ⟦⋀ *l. s*⊢ *l reachable-from y* ⟹ ¬ *s*⊢ *l reachable-from x*⟧
⟹ *disj x y s*
by (*auto simp add*: *disj-def*)

lemma *disj-cases* [*consumes 1*]:
assumes *disj x y s*

```
  assumes ⋀ l.  ¬ s⊢ l reachable-from x ⟹ P
  assumes ⋀ l.  ¬ s⊢ l reachable-from y ⟹ P
  shows P
  using assms by (auto simp add: disj-def)
```

Lemma 3.3 (i) in [PH97]

```
lemma disj1: ⟦disj x y s; ¬ s⊢ l reachable-from x; ¬ s⊢ l reachable-from y⟧
              ⟹ disj x y (s⟨l:=z⟩)
  by (auto simp add: disj-def)
```

Lemma 3.3 (ii)

```
lemma disj2:
  assumes disj-x-y: disj x y s
  assumes disj-x-z: disj x z s
  assumes unreach-l-x: ¬ s⊢ l reachable-from x
  shows disj x y (s⟨l:=z⟩)
proof (rule disjI1)
  fix k
  assume reach-k-x: s⟨l := z⟩⊢ k reachable-from x
  show ¬ s⟨l := z⟩⊢ k reachable-from y
  proof -
    from unreach-l-x reach-k-x
    have reach-s-k-x: s⊢ k reachable-from x
      by simp
    with disj-x-z
    have ¬ s⊢ k reachable-from z
      by (simp add: disj-def)
    moreover from reach-s-k-x disj-x-y
    have ¬ s⊢ k reachable-from y
      by (simp add: disj-def)
    ultimately show ?thesis
      by (rule reach3)
  qed
qed
```

Lemma 3.3 (iii)

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

```
    by simp
  qed
qed
```

Lemma 3.3 (iv)

```
lemma disj4: ⟦disj (objV C a) y s; CClassT C ≤ dtype f ⟧
       ⟹ disj (s@@(objV C a)..f) y s
  by (auto simp add: disj-def)

lemma disj4': ⟦disj (arrV T a) y s ⟧
       ⟹ disj (s@@(arrV T a).[i]) y s
  by (auto simp add: disj-def)
```

13.4 X-Equivalence

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

```
definition xeq:: Value ⇒ Store ⇒ Store ⇒ bool where
xeq x s t = (alive x s = alive x t ∧
      (∀ l. s⊢ l reachable-from x ⟶ s@@l = t@@l))

abbreviation xeq-syntax :: Store ⇒ Value ⇒ Store ⇒ bool
  (‹-/ (≡[-])/ -› [900,0,900] 900)
where s ≡[x] t == xeq x s t


lemma xeqI: ⟦alive x s = alive x t;
      ⋀ l. s⊢ l reachable-from x ⟹ s@@l = t@@l
      ⟧ ⟹ s ≡[x] t
  by (auto simp add: xeq-def)
```

Lemma 3.4 (i) in [PH97].

```
lemma xeq1-refl: s ≡[x] s
  by (simp add: xeq-def)
```

Lemma 3.4 (i)

```
lemma xeq1-sym':
  assumes s-t: s ≡[x] t
  shows t ≡[x] s
proof –
  from s-t have alive x s = alive x t by (simp add: xeq-def)
  moreover
  from s-t have ∀ l. s⊢ l reachable-from x ⟶ s@@l = t@@l
    by (simp add: xeq-def)
  with reach9 [OF this]
  have ∀ l. t⊢ l reachable-from x ⟶ t@@l = s@@l
    by simp
  ultimately show ?thesis
    by (simp add: xeq-def)
qed
```

```
lemma xeq1-sym: s ≡[x] t = t ≡[x] s
  by (auto intro: xeq1-sym')
```

Lemma 3.4 (i)

```
lemma xeq1-trans [trans]:
  assumes s-t: s ≡[x] t
  assumes t-r: t ≡[x] r
  shows s ≡[x] r
proof −
  from s-t t-r
  have alive x s = alive x r
    by (simp add: xeq-def)
  moreover
  have ∀ l. s⊢ l reachable-from x ⟶ s@@l = r@@l
  proof (intro allI impI)
    fix l
    assume reach-l: s⊢ l reachable-from x
    show s@@l = r@@l
    proof −
      from reach-l s-t have s@@l=t@@l
        by (simp add: xeq-def)
      also have t@@l = r@@l
      proof −
        from s-t have ∀ l. s⊢ l reachable-from x ⟶ s@@l = t@@l
          by (simp add: xeq-def)
        from reach9 [OF this] reach-l have t⊢ l reachable-from x
          by simp
        with t-r show ?thesis
          by (simp add: xeq-def)
      qed
      finally show ?thesis .
    qed
  qed
  ultimately show ?thesis
    by (simp add: xeq-def)
qed
```

Lemma 3.4 (ii)

```
lemma xeq2:
  assumes xeq: ∀ x. s ≡[x] t
  assumes static-eq: ∀ f. s@@(staticLoc f) = t@@(staticLoc f)
  shows s = t
proof (rule Store-eqI)
  from xeq
  show ∀x. alive x s = alive x t
    by (simp add: xeq-def)
next
  show ∀l. s@@l = t@@l
  proof
    fix l
    show s@@l = t@@l
    proof (cases l)
      case (objLoc cf a)
      have l = objLoc cf a by fact
```

```
      hence s⊢ l reachable-from (objV (cls cf) a)
        by simp
      with xeq show ?thesis
        by (simp add: xeq-def)
    next
      case (staticLoc f)
      have l = staticLoc f by fact
      with static-eq show ?thesis
        by (simp add: xeq-def)
    next
      case (arrLenLoc T a)
      have l = arrLenLoc T a by fact
      hence s⊢ l reachable-from (arrV T a)
        by simp
      with xeq show ?thesis
        by (simp add: xeq-def)
    next
      case (arrLoc T a i)
      have l = arrLoc T a i by fact
      hence s⊢ l reachable-from (arrV T a)
        by simp
      with xeq show ?thesis
        by (simp add: xeq-def)
    qed
  qed
qed
```

Lemma 3.4 (iii)

```
lemma xeq3:
  assumes unreach-l: ¬ s⊢ l reachable-from x
  shows s ≡[x] s⟨l:=y⟩
proof (rule xeqI)
  show alive x s = alive x (s⟨l := y⟩)
    by simp
next
  fix k
  assume reach-k: s⊢ k reachable-from x
  with unreach-l have l≠k by auto
  then show s@@k = s⟨l := y⟩@@k
    by simp
qed
```

Lemma 3.4 (iv)

```
lemma xeq4: assumes not-new: x ≠ new s t
  shows s ≡[x] s⟨t⟩
proof (rule xeqI)
  from not-new
  show alive x s = alive x (s⟨t⟩)
    by (simp add: alive-alloc-exhaust)
next
  fix l
  assume reach-l: s⊢ l reachable-from x
  show s@@l = s⟨t⟩@@l
  proof (cases isNewArr t ⟶ l ≠ arr-len (new s t))
```

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

Lemma 3.4 (v)

lemma *xeq5*: $s \equiv[x]\ t \Longrightarrow s \vdash l$ *reachable-from* $x = t \vdash l$ *reachable-from* x
by (*rule reach9*) (*simp add: xeq-def*)

13.5 T-Equivalence

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

definition *teq*:: *Javatype* ⇒ *Store* ⇒ *Store* ⇒ *bool* **where**
$teq\ t\ s1\ s2 = (\forall\ x.\ typeof\ x \le t \longrightarrow s1 \equiv[x]\ s2)$

13.6 Less Alive

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

definition *lessalive*:: *Store* ⇒ *Store* ⇒ *bool* (‹-/ ≪ -› [70,71] 70)
where $lessalive\ s\ t = ((\forall\ x.\ alive\ x\ s \longrightarrow s \equiv[x]\ t) \wedge (\forall\ f.\ s@@staticLoc\ f = t@@staticLoc\ f))$

We define an introduction rule for the new operator.

lemma *lessaliveI*:
$[\![\bigwedge x.\ alive\ x\ s \Longrightarrow\ s \equiv[x]\ t;\ \bigwedge f.\ s@@staticLoc\ f = t@@staticLoc\ f]\!]$
$\Longrightarrow s \ll t$
by (*simp add*: *lessalive-def*)

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

lemma *lessalive-refl*: $s \ll s$
by (*simp add*: *lessalive-def xeq1-refl*)

lemma *lessalive-trans* [*trans*]:
assumes *s-t*: $s \ll t$

```
  assumes t-w: t ≪ w
  shows s ≪ w
proof (rule lessaliveI)
  fix x
  assume alive-x-s: alive x s
  with s-t have s ≡[x] t
    by (simp add: lessalive-def)
  also
  have t ≡[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 ≡[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 ≪ t
  assumes t-s: t ≪ s
  shows s = t
proof (rule xeq2)
  show ∀ x. s ≡[x] t
  proof
    fix x show 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
      show ?thesis
      proof (cases alive x t)
        case True
        with t-s show ?thesis
          by (subst xeq1-sym) (simp add: lessalive-def)
      next
        case False
        show ?thesis
        proof (rule xeqI)
          from False unalive-x-s show alive x s = alive x t by simp
        next
          fix l assume reach-s-x: s⊢ l reachable-from x
          with unalive-x-s have x: x = ref l
            by (rule ref-reach-unalive)
          with unalive-x-s have s@@l = init (ltype l)
            by simp
          also from reach-s-x x have t⊢ l reachable-from x
            by (auto intro: reach.Immediate unreachable-Null)
          with False x have t@@l = init (ltype l)
            by simp
```

```
          finally show s@@l = t@@l
            by simp
        qed
      qed
    qed
  qed
next
  from s-t show ∀f. s@@staticLoc f = t@@staticLoc f
    by (simp add: lessalive-def)
qed
```

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 $\leq$ symbols, and to the type class *order* which axiomatizes partial orderings.

```
instantiation Store :: order
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
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 *not-treach-ref-impl-not-reach* below.

```
inductive
  treach :: Javatype ⇒ Javatype ⇒ bool
where
  Subtype:       U ≤ T ⟹ treach T U
| Attribute:     ⟦treach T S; S ≤ dtype f; U ≤ rtype f⟧ ⟹ treach T U
| ArrLength:      treach (ArrT AT) IntgT
| ArrElem:        treach (ArrT AT) (at2jt AT)
| Trans [trans]: ⟦treach T U; treach U V⟧ ⟹ treach T V


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

lemma treach-ref-l' [simp,intro]:
  assumes not-Null: ref l ≠ nullV
  shows treach (typeof (ref l)) (typeof (s@@l))
```

proof –
  from *not-Null* **have** *treach* (*typeof* (*ref l*)) (*ltype l*) **by** (*rule treach-ref-l*)
  also have *typeof* (*s@@l*) ≤ *ltype l*
    by *simp*
  hence *treach* (*ltype l*) (*typeof* (*s@@l*))
    by (*rule treach.intros*)
  finally show *?thesis* **.**
qed

lemma *reach-impl-treach*:
  assumes *reach-l*: *s* ⊢ *l reachable-from x*
  shows *treach* (*typeof x*) (*ltype l*)
using *reach-l*
proof (*induct*)
  case (*Immediate l*)
  have *ref l* ≠ *nullV* **by** *fact*
  then show *treach* (*typeof* (*ref l*)) (*ltype l*)
    by (*rule treach-ref-l*)
next
  case (*Indirect l k*)
  have *treach* (*typeof* (*s@@k*)) (*ltype l*) **by** *fact*
  moreover
  have *ref k* ≠ *nullV* **by** *fact*
  hence *treach* (*typeof* (*ref k*)) (*typeof* (*s@@k*))
    by *simp*
  ultimately show *treach* (*typeof* (*ref k*)) (*ltype l*)
    by (*iprover intro*: *treach.Trans*)
qed

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

```
      finally have treach (typeof (ref k)) (typeof (ref l)) .
      with not-k-l show False ..
    qed
  qed
qed
```

Lemma 4.6 in [PH97, p. 107].

```
lemma treach1:
  assumes x-t: typeof x ≤ T
  assumes not-treach: ¬ treach T (typeof (ref l))
  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.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[MPH00] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2000.

[PH97] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.

[PHGR05] Arnd Poetzsch-Heffter, Jean-Marie Gaillourdet, and Nicole Rauch. A Hoare Logic for a Java Subset and its Proof of Soundness and Completeness. Internal report, University of Kaiserslautern, Germany, 2005. To appear.