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 $\mathcal{A} \rhd \{ \mathbf{P} \} \texttt{pp} \{ \mathbf{Q} \}$ and express that for all states S that fulfill $\mathbf{P}$, if the execution of the program part 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]:

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

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.

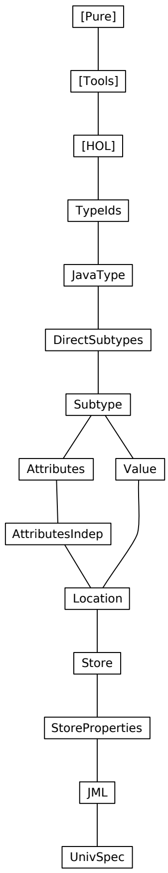
- In PVS, function arguments were sometimes restricted to subtypes. In Isabelle/HOL, unintended usage of functions is left unspecified.
- In PVS, the program-independent theories were parameterized by the datatypes that were generated for the program to be verified. In Isabelle/HOL, we just build on 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 × 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*

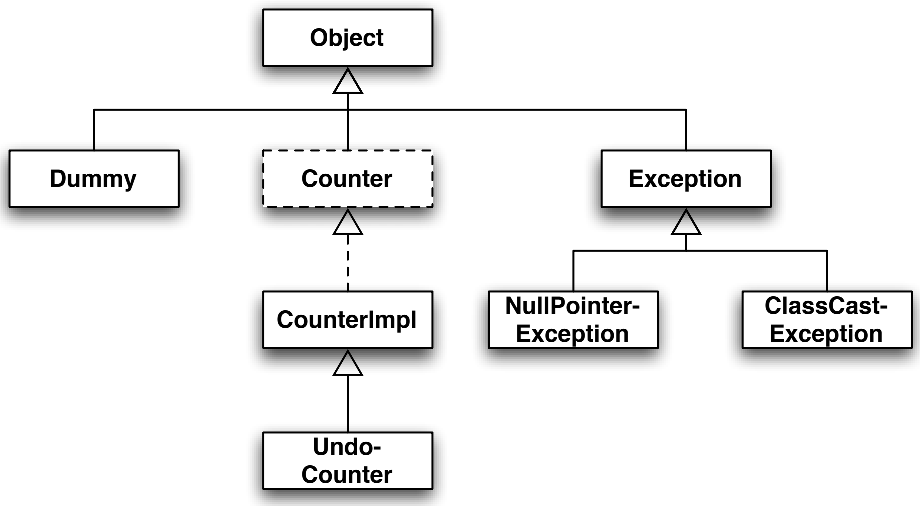
end

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*
⟨*proof*⟩

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 $(x,y) \in r^+$
⟨*proof*⟩

lemma *acyclic-imp-antisym-rtrancl*: *acyclic* $r \Longrightarrow$ *antisym* (r^*)

⟨*proof*⟩

lemma *acyclic-trancl-rtrancl*:
  assumes *acyclic*: *acyclic r*
  shows $(x,y) \in r^+ = ((x,y) \in r^* \wedge x \neq y)$
⟨*proof*⟩

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
  $A \preceq B == (A,B) \in \textit{direct-subtype}^*$

abbreviation
  widen-strict-syntax :: *Javatype* ⇒ *Javatype* ⇒ *bool* (‹- ≺ -› [71,71] 70)
where — transitive closure of direct subtype relation
  $A \prec B == (A,B) \in \textit{direct-subtype}^+$

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*⟩

lemma *antisym-rtrancl-direct-subtype*: *antisym* $(\textit{direct-subtype}^*)$
⟨*proof*⟩

lemma *widen-strict-to-widen*: $C \prec D = (C \preceq D \wedge C \neq D)$
⟨*proof*⟩

The widening relation on Javatype is reflexive.

lemma *widen-refl* [*iff*]: $X \preceq X$ ⟨*proof*⟩

The widening relation on Javatype is transitive.

lemma *widen-trans* [*trans*] :
  assumes *a-b*: $a \preceq b$
  shows $\bigwedge c.\ b \preceq c \Longrightarrow a \preceq c$
  ⟨*proof*⟩

The widening relation on Javatype is antisymmetric.

lemma *widen-antisym* [*trans*]:
  assumes *a-b*: $a \preceq b$
  assumes *b-c*: $b \preceq a$
  shows $a = b$
  ⟨*proof*⟩

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*⟩

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*: $(a{::}Javatype) \leq b \Longrightarrow a \leq b$
  ⟨*proof*⟩

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

⟨*ML*⟩

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*
⟨*proof*⟩

lemma *Object-root-abs*: *AClassT C* $\leq$ *CClassT Object*
⟨*proof*⟩

lemma *Object-root-int*: *InterfaceT C* $\leq$ *CClassT Object*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *NullT-leaf-abs* [*simp*]: *NullT* $\leq$ *AClassT C*
⟨*proof*⟩

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

lemma *NullT-leaf-array*: *NullT* $\leq$ *ArrT C*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*]:
$CClassT\ c \leq dtype\ f \Longrightarrow cls\ (catt\ c\ f) = c$
⟨*proof*⟩

lemma *att-catt* [*simp*]:
$CClassT\ c \leq dtype\ f \Longrightarrow att\ (catt\ c\ f) = f$
⟨*proof*⟩

The following lemmas are just a demonstration of simplification.

lemma *rtype-att-catt*:
$CClassT\ c \leq dtype\ f \Longrightarrow rtype\ (att\ (catt\ c\ f)) = rtype\ f$
⟨*proof*⟩

lemma *widen-cls-dtype-att* [*simp*,*intro*]:
$(CClassT\ (cls\ cf) \leq dtype\ (att\ cf))$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *typeof-init-widen* [*simp*,*intro*]: *typeof* (*init T*) ≤ *T*
⟨*proof*⟩

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
  ⟨proof⟩
```

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
  ⟨proof⟩
```

```
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
  ⟨proof⟩
```

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*
⟨*proof*⟩

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* ≤ *dtype f*⟧ ⟹ *ref* (*r*..*f*) = *r*
⟨*proof*⟩

lemma *obj-arr-loc* [*simp*]: *isArrV r* ⟹ *ref* (*r*.[*i*]) = *r*
⟨*proof*⟩

lemma *obj-arr-len* [*simp*]: *isArrV r* ⟹ *ref* (*arr-len r*) = *r*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

[2] In the following, the term "objects" includes arrays. This keeps the explanations compact.

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
  ⟨proof⟩
```

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.

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *alive-arr-loc* [*simp*]:
isArrV x ⟹ *alive* (*ref* (*x.*[*i*])) *s* = *alive x s*
⟨*proof*⟩

lemma *alive-arr-len* [*simp*]:
isArrV x ⟹ *alive* (*ref* (*arr-len x*)) *s* = *alive x s*
⟨*proof*⟩

lemma *ref-arr-len-new* [*simp*]:
ref (*arr-len* (*new s* (*new-array T n*))) = *new s* (*new-array T n*)
⟨*proof*⟩

lemma *ref-arr-loc-new* [*simp*]:
ref ((*new s* (*new-array T n*)).[*i*]) = *new s* (*new-array T n*)
⟨*proof*⟩

lemma *ref-loc-new* [*simp*]: *CClassT C* ≤ *dtype f*
⟹ *ref* ((*new s* (*new-instance C*))..*f*) = *new s* (*new-instance C*)
⟨*proof*⟩

lemma *access-type-safe* [*simp*,*intro*]: *typeof* (*s*@@*l*) ≤ *ltype l*
⟨*proof*⟩

The store is well-typed by construction.

lemma *always-welltyped-store*: *wts OS*
⟨*proof*⟩

Store8

lemma *alive-access* [*simp*,*intro*]: *alive* (*s*@@*l*) *s*
⟨*proof*⟩

Store3

lemma *access-unalive* [*simp*]:
assumes *unalive*: ¬ *alive* (*ref l*) *s*
shows *s*@@*l* = *init* (*ltype l*)
⟨*proof*⟩

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⟩)*
⟨*proof*⟩

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*⟩

Store6

lemma *alive-update-invariant* [*simp*]: *alive x* (*s*⟨*l*:=*y*⟩) = *alive x s*
⟨*proof*⟩

Store1

lemma *access-update-other* [*simp*]:
assumes *neq-l-m*: *l* ≠ *m*
shows *s*⟨*l*:=*x*⟩@@*m* = *s*@@*m*
⟨*proof*⟩

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*⟨*l*:=*x*⟩@@*l* = *x*
⟨*proof*⟩

Store4

lemma *update-unalive-val* [*simp,intro*]: ¬ *alive x s* ⟹ *s*⟨*l*:=*x*⟩ = *s*
⟨*proof*⟩

lemma *update-unalive-loc* [*simp,intro*]: ¬ *alive* (*ref l*) *s* ⟹ *s*⟨*l*:=*x*⟩ = *s*
⟨*proof*⟩

lemma *update-type-mismatch* [*simp,intro*]: ¬ *typeof x* ≤ *ltype l* ⟹ *s*⟨*l*:=*x*⟩ = *s*
⟨*proof*⟩

Store9

lemma *alive-primitive* [*simp,intro*]: *isprimitive* (*typeof x*) ⟹ *alive x s*
⟨*proof*⟩

Store10

lemma *new-unalive-old-Store* [*simp*]: ¬ *alive* (*new s t*) *s*
⟨*proof*⟩

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

(**is** *?s-alloc* ∈ *Store*)
⟨*proof*⟩

lemma *alloc-new-array-in-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 n*))⦈) ∈ *Store*
(**is** *?s-alloc* ∈ *Store*)
⟨*proof*⟩

lemma *new-alive-alloc* [*simp*,*intro*]: *alive* (*new s t*) (*s*⟨*t*⟩)
⟨*proof*⟩

lemma *value-class-inhabitants*:
(∀ *x*. *typeof x* = *CClassT typeId* ⟶ *P x*) = (∀ *a*. *P* (*objV typeId a*))
(**is** (∀ *x*. *?A x*) = *?B*)
⟨*proof*⟩

lemma *value-array-inhabitants*:
(∀ *x*. *typeof x* = *ArrT typeId* ⟶ *P x*) = (∀ *a*. *P* (*arrV typeId a*))
(**is** (∀ *x*. *?A x*) = *?B*)
⟨*proof*⟩

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*⟩

lemma *all-le-eq-imp-eq*: ⋀ *c*::*nat*. (∀ *a*. (*a* < *d*) = (*a* < *c*)) ⟶ (*d* = *c*)
⟨*proof*⟩

lemma *all-le-eq*: (∀ *a*::*nat*. (*a* < *d*) = (*a* < *c*)) = (*d* = *c*)
⟨*proof*⟩

Store11

lemma *typeof-new*: *typeof* (*new s t*) = *typeofNew t*
⟨*proof*⟩

Store12

lemma *new-eq*: (*new s1 t* = *new s2 t*) =
  (∀ *x*. *typeof x* = *typeofNew t* ⟶ *alive x s1* = *alive x s2*)
⟨*proof*⟩

lemma *new-update* [*simp*]: *new* (*s*⟨*l*:=*x*⟩) *t* = *new s t*
⟨*proof*⟩

lemma *alive-alloc-propagation*:
  assumes *alive-s*: *alive x s* **shows** *alive x* $(s\langle t\rangle)$
⟨*proof*⟩

Store7

lemma *alive-alloc-exhaust*: *alive x* $(s\langle t\rangle) = ($*alive x s* $\vee$ $(x =$ *new s t*$))$
⟨*proof*⟩

lemma *alive-alloc-cases* [*consumes 1*]:
  ⟦*alive x* $(s\langle t\rangle)$; *alive x s* $\Longrightarrow$ *P*; *x=new s t* $\Longrightarrow$ *P*⟧
  $\Longrightarrow$ *P*
  ⟨*proof*⟩

lemma *aliveImpl-vals-independent*: *aliveImpl x* (*s*⦇*vals* := *z*⦈) = *aliveImpl x s*
  ⟨*proof*⟩

lemma *access-arr-len-new-alloc* [*simp*]:
  s⟨*new-array T l*⟩@@*arr-len* (*new s* (*new-array T l*)) = *intgV* (*int l*)
  ⟨*proof*⟩

lemma *access-new* [*simp*]:
  assumes *ref-new*: *ref l* = *new s t*
  assumes *no-arr-len*: *isNewArr t* $\longrightarrow$ *l* $\neq$ *arr-len* (*new s t*)
  shows *s*⟨*t*⟩@@*l* = *init* (*ltype l*)
⟨*proof*⟩

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* $\longrightarrow$ *l* $\neq$ *arr-len* (*new s t*)
  shows *s*⟨*t*⟩@@*l* = *s*@@*l*
⟨*proof*⟩

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*⟩

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* $\neq$ *new s t*
  assumes *neq-x-new*: *x* $\neq$ *new s t*
  shows *s*⟨*t*⟩⟨*l*:=*x*⟩ = *s*⟨*l*:=*x*⟩⟨*t*⟩
⟨*proof*⟩

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*
  ⟨*proof*⟩

corollary *unreachable-Null-simp* [*simp*]:
  ¬ *s*⊢ *l reachable-from nullV*
  ⟨*proof*⟩

corollary *unreachable-NullE* [*elim*]:
  s⊢ *l reachable-from nullV* ⟹ *P*
  ⟨*proof*⟩

lemma *reachObjLoc* [*simp*,*intro*]:
  C=*cls cf* ⟹ *s*⊢ *objLoc cf a reachable-from objV C a*
  ⟨*proof*⟩

lemma *reachArrLoc* [*simp*,*intro*]: *s*⊢ *arrLoc T a i reachable-from arrV T a*
  ⟨*proof*⟩

lemma *reachArrLen* [*simp*,*intro*]: *s*⊢ *arrLenLoc T a reachable-from arrV T a*
  ⟨*proof*⟩

lemma *unreachStatic* [*simp*]: ¬ *s*⊢ *staticLoc f reachable-from x*
⟨*proof*⟩

lemma *unreachStaticE* [*elim*]: $s\vdash$ *staticLoc f reachable-from x* $\Longrightarrow P$
⟨*proof*⟩

lemma *reachable-from-ArrLoc-impl-Arr* [*simp*,*intro*]:
assumes *reach-loc*: $s\vdash$ *l reachable-from* (*s@@arrLoc T a i*)
shows $s\vdash$ *l reachable-from* (*arrV T a*)
⟨*proof*⟩

lemma *reachable-from-ObjLoc-impl-Obj* [*simp*,*intro*]:
assumes *reach-loc*: $s\vdash$ *l reachable-from* (*s@@objLoc cf a*)
assumes *C*: *C=cls cf*
shows $s\vdash$ *l reachable-from* (*objV C a*)
⟨*proof*⟩

Lemma 3.2 (i)

lemma *reach-update* [*simp*]:
assumes *unreachable-l-x*: $\neg$ $s\vdash$ *l reachable-from x*
shows $s\langle l{:=}y\rangle\vdash$ *k reachable-from x* $=$ $s\vdash$ *k reachable-from x*
⟨*proof*⟩

Lemma 3.2 (ii)

lemma *reach2*:
$\neg$ $s\vdash$ *l reachable-from x* $\Longrightarrow$ $\neg$ $s\langle l{:=}y\rangle\vdash$ *l reachable-from x*
⟨*proof*⟩

Lemma 3.2 (iv)

lemma *reach4*: $\neg$ $s \vdash$ *l reachable-from* (*ref k*) $\Longrightarrow k \neq l \lor$ (*ref k*) $=$ *nullV*
⟨*proof*⟩

lemma *reachable-isRef*:
assumes *reach*: $s\vdash$*l reachable-from x*
shows *isRefV x*
⟨*proof*⟩

lemma *val-ArrLen-IntgT*: *isArrLenLoc l* $\Longrightarrow$ *typeof* (*s@@l*) $=$ *IntgT*
⟨*proof*⟩

lemma *access-alloc′* [*simp*]:
assumes *no-arr-len*: $\neg$ *isArrLenLoc l*
shows $s\langle t\rangle$@@l $=$ *s@@l*
⟨*proof*⟩

Lemma 3.2 (v)

lemma *reach-alloc* [*simp*]: $s\langle t\rangle\vdash$ *l reachable-from x* $=$ $s\vdash$ *l reachable-from x*
⟨*proof*⟩

Lemma 3.2 (vi)

lemma *reach6*: *isprimitive*(*typeof x*) $\Longrightarrow$ $\neg$ $s \vdash$ *l reachable-from x*
⟨*proof*⟩

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*⟩

Lemma 3.2 (vii).

lemma *unreachable-from-init* [*simp*,*intro*]: ¬ *s*⊢ *l reachable-from* (*init T*)
⟨*proof*⟩

lemma *ref-reach-unalive*:
assumes *unalive-x*:¬ *alive x s*
assumes *l-x*: *s*⊢ *l reachable-from x*
shows *x* = *ref l*
⟨*proof*⟩

lemma *loc-new-reach*:
assumes *l*: *ref l* = *new s t*
assumes *l-x*: *s*⊢ *l reachable-from x*
shows *x* = *new s t*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*

⟨*proof*⟩

lemma *disjI2*: ⟦⋀ *l*. *s*⊢ *l reachable-from y* ⟹ ¬ *s*⊢ *l reachable-from x*⟧
⟹ *disj x y s*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩)
⟨*proof*⟩

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*⟩

Lemma 3.3 (iii)

lemma *disj3*: **assumes** *alive-x-s*: *alive x s*
shows *disj x* (*new s t*) (*s*⟨*t*⟩)
⟨*proof*⟩

Lemma 3.3 (iv)

lemma *disj4*: ⟦*disj* (*objV C a*) *y s*; *CClassT C* ≤ *dtype f* ⟧
⟹ *disj* (*s*@@(*objV C a*)..*f*) *y s*
⟨*proof*⟩

lemma *disj4'*: ⟦*disj* (*arrV T a*) *y s* ⟧
⟹ *disj* (*s*@@(*arrV T a*).[*i*]) *y s*
⟨*proof*⟩

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*;
$\bigwedge$ *l*. *s*⊢ *l reachable-from x* $\Longrightarrow$ *s*@@*l* = *t*@@*l*
$]\!]$ $\Longrightarrow$ *s* ≡[*x*] *t*
⟨*proof*⟩

Lemma 3.4 (i) in [PH97].

lemma *xeq1-refl*: *s* ≡[*x*] *s*
⟨*proof*⟩

Lemma 3.4 (i)

lemma *xeq1-sym'*:
assumes *s-t*: *s* ≡[*x*] *t*
shows *t* ≡[*x*] *s*
⟨*proof*⟩

lemma *xeq1-sym*: *s* ≡[*x*] *t* = *t* ≡[*x*] *s*
⟨*proof*⟩

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*⟩

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*⟩

Lemma 3.4 (iii)

lemma *xeq3*:
assumes *unreach-l*: ¬ *s*⊢ *l reachable-from x*
shows *s* ≡[*x*] *s*⟨*l*:=*y*⟩
⟨*proof*⟩

Lemma 3.4 (iv)

lemma *xeq4*: **assumes** *not-new*: *x* ≠ *new s t*
shows *s* ≡[*x*] *s*⟨*t*⟩
⟨*proof*⟩

Lemma 3.4 (v)

lemma *xeq5*: *s* ≡[*x*] *t* $\Longrightarrow$ *s*⊢ *l reachable-from x* = *t*⊢ *l reachable-from x*
⟨*proof*⟩

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 $lessalive\ s\ t = ((\forall\ x.\ alive\ x\ s \longrightarrow s \equiv[x]\ t) \land (\forall\ f.\ s@@staticLoc\ f = t@@staticLoc\ f))$

We define an introduction rule for the new operator.

lemma *lessaliveI*:
$[\![\bigwedge x.\ alive\ x\ s \Longrightarrow s \equiv[x]\ t;\ \bigwedge f.\ s@@staticLoc\ f = t@@staticLoc\ f]\!]$
$\Longrightarrow s \ll t$
⟨*proof*⟩

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

lemma *lessalive-refl*: $s \ll s$
⟨*proof*⟩

lemma *lessalive-trans* [*trans*]:
assumes *s-t*: $s \ll t$
assumes *t-w*: $t \ll w$
shows $s \ll w$
⟨*proof*⟩

lemma *lessalive-antisym*:
assumes *s-t*: $s \ll t$
assumes *t-s*: $t \ll s$
shows $s = t$
⟨*proof*⟩

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

instantiation *Store* :: *order*
begin

definition
le-Store-def: $s \le t \longleftrightarrow s \ll t$

definition
less-Store-def: $(s::Store) < t \longleftrightarrow s \le t \land \neg\ t \le s$

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

Lemma 3.5 (i)

instance ⟨*proof*⟩

end

Lemma 3.5 (ii)

lemma *lessalive2*: ⟦$s \ll t$; *alive* x s⟧ $\Longrightarrow$ *alive* x t
⟨*proof*⟩

Lemma 3.5 (iii)

lemma *lessalive3*:
assumes *s-t*: $s \ll t$
assumes *alive*: *alive* x s $\vee$ $\neg$ *alive* x t
shows $s \equiv[x]\ t$
⟨*proof*⟩

Lemma 3.5 (iv)

lemma *lessalive-update* [*simp*,*intro*]:
assumes *s-t*: $s \ll t$
assumes *unalive-l*: $\neg$ *alive* (*ref l*) t
shows $s \ll t\langle l{:=}x\rangle$
⟨*proof*⟩

lemma *Xequ4'*:
assumes *alive*: *alive* x s
shows $s \equiv[x]\ s\langle t\rangle$
⟨*proof*⟩

Lemma 3.5 (v)

lemma *lessalive-alloc* [*simp*,*intro*]: $s \ll s\langle t\rangle$
⟨*proof*⟩

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: $U \le T \Longrightarrow$ *treach* T U
| *Attribute*: ⟦*treach* T S; $S \le$ *dtype* f; $U \le$ *rtype* f⟧ $\Longrightarrow$ *treach* T U
| *ArrLength*: *treach* (*ArrT AT*) *IntgT*
| *ArrElem*: *treach* (*ArrT AT*) (*at2jt AT*)
| *Trans* [*trans*]: ⟦*treach* T U; *treach* U V⟧ $\Longrightarrow$ *treach* T V

lemma *treach-ref-l* [*simp*,*intro*]:
assumes *not-Null*: *ref l* $\neq$ *nullV*
shows *treach* (*typeof* (*ref l*)) (*ltype l*)

⟨*proof*⟩

lemma *treach-ref-l′* [*simp*,*intro*]:
 assumes *not-Null*: *ref l* ≠ *nullV*
 shows *treach* (*typeof* (*ref l*)) (*typeof* (*s*@@*l*))
⟨*proof*⟩

lemma *reach-impl-treach*:
 assumes *reach-l*: *s* ⊢ *l reachable-from x*
 shows *treach* (*typeof x*) (*ltype l*)
⟨*proof*⟩

lemma *not-treach-ref-impl-not-reach*:
 assumes *not-treach*: ¬ *treach* (*typeof x*) (*typeof* (*ref l*))
 shows ¬ *s* ⊢ *l reachable-from x*
⟨*proof*⟩

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*⟩

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.