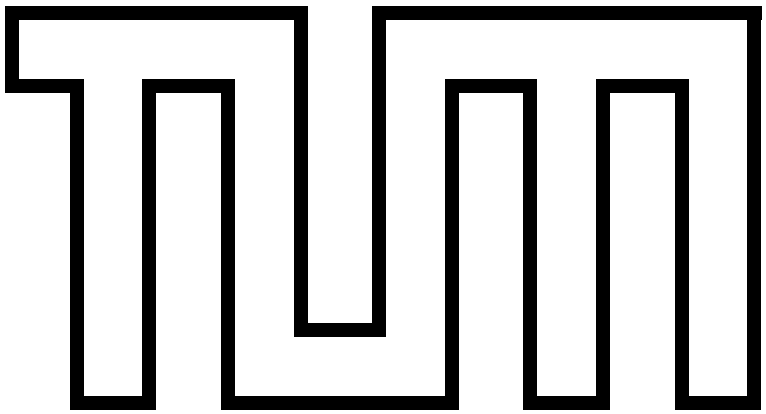


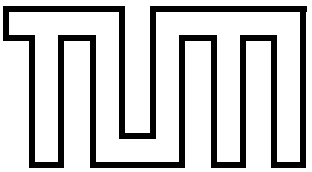
TECHNISCHE UNIVERSITÄT MÜNCHEN

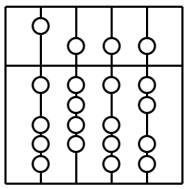
FAKULTÄT FÜR INFORMATIK

An Isabelle Formalization of the Universe Type System

Diplomarbeit in Informatik

Martin Klebermaß





FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN
UNIVERSITÄT MÜNCHEN

Lehrstuhl IV
Software & Systems Engineering

An Isabelle Formalization of the Universe Type System

Diplomarbeit in Informatik

Martin Klebermaß

Aufgabensteller	:	Prof. Tobias Nipkow
Betreuer	:	Prof. Peter Müller
Betreuer	:	Dipl.-Ing. Werner M. Dietl
Abgabedatum	:	April 11, 2007

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den April 11, 2007

(Martin Klebermaß)

Abstract

Today still very often faulty software is delivered. In contrast to other engineering tasks no one wonders about that. But there are a lot of areas where faulty software could generate high costs.
To solve that problem the correctness of software should be guaranteed.
The Universe Type System [11] helps to make this step more simple. It allows to structure the heap memory into contexts. Those contexts can be used to reason about software.
As the type system will be used to prove the correctness of programs, the type system itself has to be shown sound.
The goal of this diploma thesis is to formalize the Universe Type System using a Featherweight Java like syntax, extended with field updates. The formalization will be proven using the theorem prover Isabelle/HOL. It is based on the Isabelle/HOL Featherweight Java Implementation [5], the paper about generic universe types [3] and the Universe Java - paper [2]. The proof is separated into a proof of preserving the topology, and a proof of the encapsulation of the type system.
Using Isabelle/HOL, it was possible to show the type preservation of the semantics, as well as the owner as modifier property. Also a new interpretation of the universe modifiers for arrays has been shown sound.

Contents

1 Introduction **13**
- 1.1 Problems of aliasing in object orientation . . . 13
- 1.2 Universe Type System . . . 13
  - 1.2.1 Heap structure . . . 13
  - 1.2.2 Owner-as-modifier vs Owner-as-dominator . . . 14
  - 1.2.3 Modifiers . . . 15
  - 1.2.4 Topological vs. Encapsulation Type System . . . 16
- 1.3 Isabelle/HOL . . . 16
- 1.4 Type systems and how to prove properties of them . . . 17
  - 1.4.1 Featherweight Java . . . 18
  - 1.4.2 Jinja . . . 18
- 1.5 Overview . . . 18

2 Type system **19**
- 2.1 Static checking . . . 19
  - 2.1.1 Syntax . . . 19
  - 2.1.2 Viewpoint adaptation . . . 21
  - 2.1.3 Subtyping . . . 22
  - 2.1.4 Type rules . . . 25
  - 2.1.5 Well-formedness rules . . . 26
- 2.2 Runtime model . . . 27
  - 2.2.1 Heap definition . . . 27
  - 2.2.2 Auxiliary functions for the runtime environment . . . 28
  - 2.2.3 Well typedness of the runtime environment . . . 29
  - 2.2.4 Bigstep semantics . . . 30
- 2.3 Topological proofs . . . 31
  - 2.3.1 Correct viewpoint adaptation . . . 32
  - 2.3.2 Type preservation . . . 32
- 2.4 Encapsulation system . . . 33

3 Formalization of arrays **35**
- 3.1 New definition of arrays . . . 35
- 3.2 Static checking . . . 36
  - 3.2.1 Syntax . . . 36
  - 3.2.2 Viewpoint adaptation . . . 37
  - 3.2.3 Subtyping . . . 37
  - 3.2.4 Type system . . . 39
- 3.3 Runtime model . . . 39
  - 3.3.1 Heap definition and well typedness of the environment . . . 39
  - 3.3.2 Bigstep semantics . . . 40
- 3.4 Proofs . . . 40

4 Isabelle Formalization **45**
4.1 Overview over theory files . . . 45
4.2 Formalization . . . 45
4.2.1 Syntax definition . . . 46
4.2.2 Viewpoint adaptation . . . 48
4.2.3 Typing relation . . . 49
4.2.4 Subtyping relation . . . 50
4.2.5 Static helper functions . . . 52
4.2.6 Expression typing . . . 54
4.2.7 Static lemmas . . . 59
4.2.8 Heap model . . . 60
4.2.9 Runtime model . . . 63
4.2.10 Welltypedness definition . . . 65
4.2.11 Bigstep formalization . . . 67
4.2.12 Topological typesystem proofs . . . 70
4.2.13 Encapsulating typing rules . . . 77
4.2.14 Owner as modifier property . . . 80
4.3 Problems and the resolution . . . 83

5 Conclusion **85**
5.1 Results . . . 85
5.2 Future work . . . 85

A Appendix: Viewpoint adaptation of the ownership modifier **87**
A.1 Existing Functions in GUT . . . 87
A.2 UJ . . . 88
A.3 Comparison of the methods in both papers . . . 89
A.3.1 Cases a and b . . . 89
A.3.2 Cases c and d . . . 89
A.3.3 Case e . . . 91

List of Figures

1.1 Structuring the heap . . . 14
1.2 Owner-as-dominator vs Owner-as-modifier . . . 15
1.3 Isabelle/HOL - Proof General . . . 17

2.1 Derivation tree of an expression . . . 21
2.2 Syntax tree . . . 22
2.3 Viewpoint adaptation . . . 23
2.4 Subclassing . . . 23
2.5 Order of the ownership modifiers . . . 23
2.6 Subtyping of the universe modifer . . . 24
2.7 Subtyping . . . 25
2.8 Typing rules . . . 26
2.9 Wellformedness rules . . . 27
2.10 Heap definition . . . 28
2.11 The dynType function . . . 29
2.12 Wellformed model . . . 30
2.13 Bigstep semantics . . . 31
2.14 Adaptation from a viewpoint . . . 32
2.15 Adaptation to a viewpoint . . . 32
2.16 Type safety . . . 33
2.17 Encapsulation rules . . . 34
2.18 Owner-as-Modifier property . . . 34

3.1 Old and new definition of arrays . . . 36
3.2 Mapping old to new array system . . . 37
3.3 Transformation rules for array definition . . . 37
3.4 Syntax extended with arrays . . . 38
3.5 Viewpoint adaptation with arrays . . . 38
3.6 Array subtyping rules . . . 39
3.7 Syntax rules with arrays . . . 41
3.8 Heap with arrays . . . 41
3.9 The dynType function for arrays . . . 42
3.10 Welltyped address . . . 42
3.11 Bigstep semantics with arrays . . . 43

1 Introduction

Reasoning about software is one of the key problems in informatics. Today programming theory and the possibilities of proving the correctness of software still lack behind practical programming situations. They either limit the sort of software that could be developed, or it is complicated to prove the correctness of the program. At least in some areas, where errors in the software programs would generate a lot of damage or costs, computer checked code has become more and more important.
One argument to introduce object oriented programming was to reduce complexity and to encourage reusability. The problem with reasoning about software programs is that for each written class the correctness of some properties have to be shown in every context the class is used.
One part to handle that problem is to use the Universe Type System[10]. Other parts, like invariant semantics or models are not handled in this paper.

This chapter will make you familiar with related information to understand the formalization.

1.1 Problems of aliasing in object orientation

While reasoning about object-oriented programs aliasing can make the work really complicated. Aliasing is a part of many programming languages. It gives the possibility of accessing an object using more than one name. An example would be two variables mapping to the same object. If the object behind one of those variables is changed, the object behind the other one is changed too (because it is the same) [6, 11].
This could break invariants during program execution and thus can make reasoning about a program more complicated If a property is defined for a variable, it has to be guaranteed that this property is not destroyed by modifying the object using an alias variable that maps to that object.

1.2 Universe Type System

The Universe Type System describes a possibility how to enforce a structure in the heap and to enforce the owner-as-modifier property for object oriented programming languages [9, 4].

1.2.1 Heap structure

Why should the heap be structured?
In contrast to classes in Java, which have a structure tree for the inheritance, building upon each other, the object store does not have any order. New objects are just created at a free address.

Instead of having no structure, using the universe type system the objects can be given an ownership information, which make it possible to structure the objects. Using this structure and some restrictions gives one the ability to get control over objects directly owned by an object. In figure 1.1 one can see a model for an unstructured heap and a structured heap. In the case of the structured heap access will be limited by this structuring information. There will be some limitations like access is only allowed inside the structure but not outside. Those restrictions can be divided into two concepts: Owner-as-modifier and Owner-as-dominator.

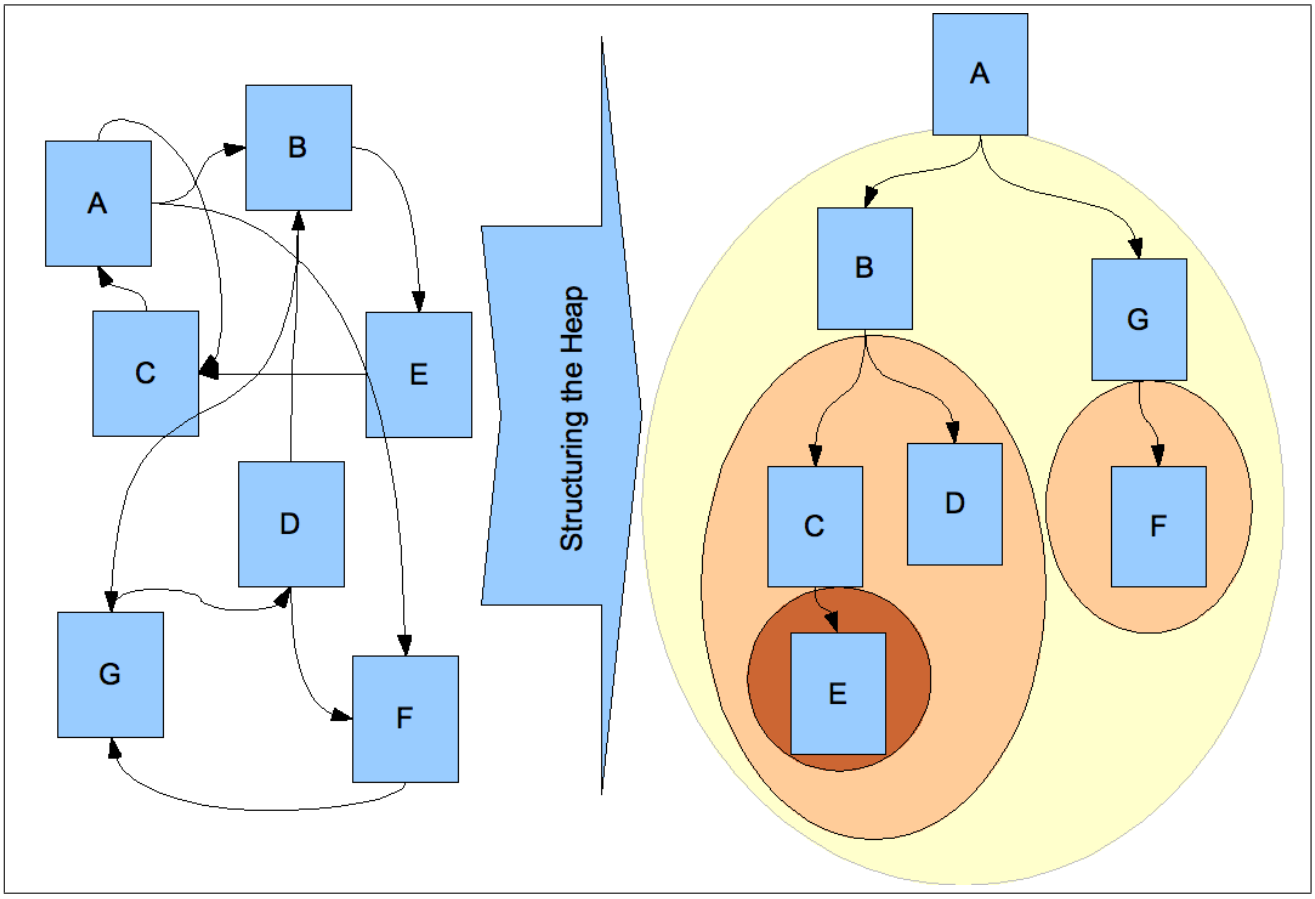


Figure 1.1: Structuring the heap

1.2.2 Owner-as-modifier vs Owner-as-dominator

In both concepts objects have owners, and in this way the hierarchy is built. The main difference in the two concepts is the read access to fields. The owner-as-dominator property forbids every access to an object, even read access. The advantage for that is a complete encapsulation of objects and what happens inside the object. Thus an object has complete control over the information and the access to its child objects [1].
Although it is very powerful in the sense of encapsulation, the owner-as-dominator property restricts the programmer in the way how to implement a software program, reduces the possibility of reusing objects and blows up the single classes with information not needed.

For this reason the owner-as-modifier property gives the possibility of relaxing access limitations to only control write access to objects (everyone can read every object) and still gives the possibility of defining properties for a class as modification of objects can be controlled. With this property a lot of assertions about a class can be proven once and reused in other classes as assumptions without proving it in every situation again.
In contrast to the owner-as-dominator property, where all access (even read access) has to be done with respect to the owner of an object, the owner-as-modifier property gives the possibility of allowing aliasing and thus lets you create more flexible programs.

Figure 1.2 compares both properties.

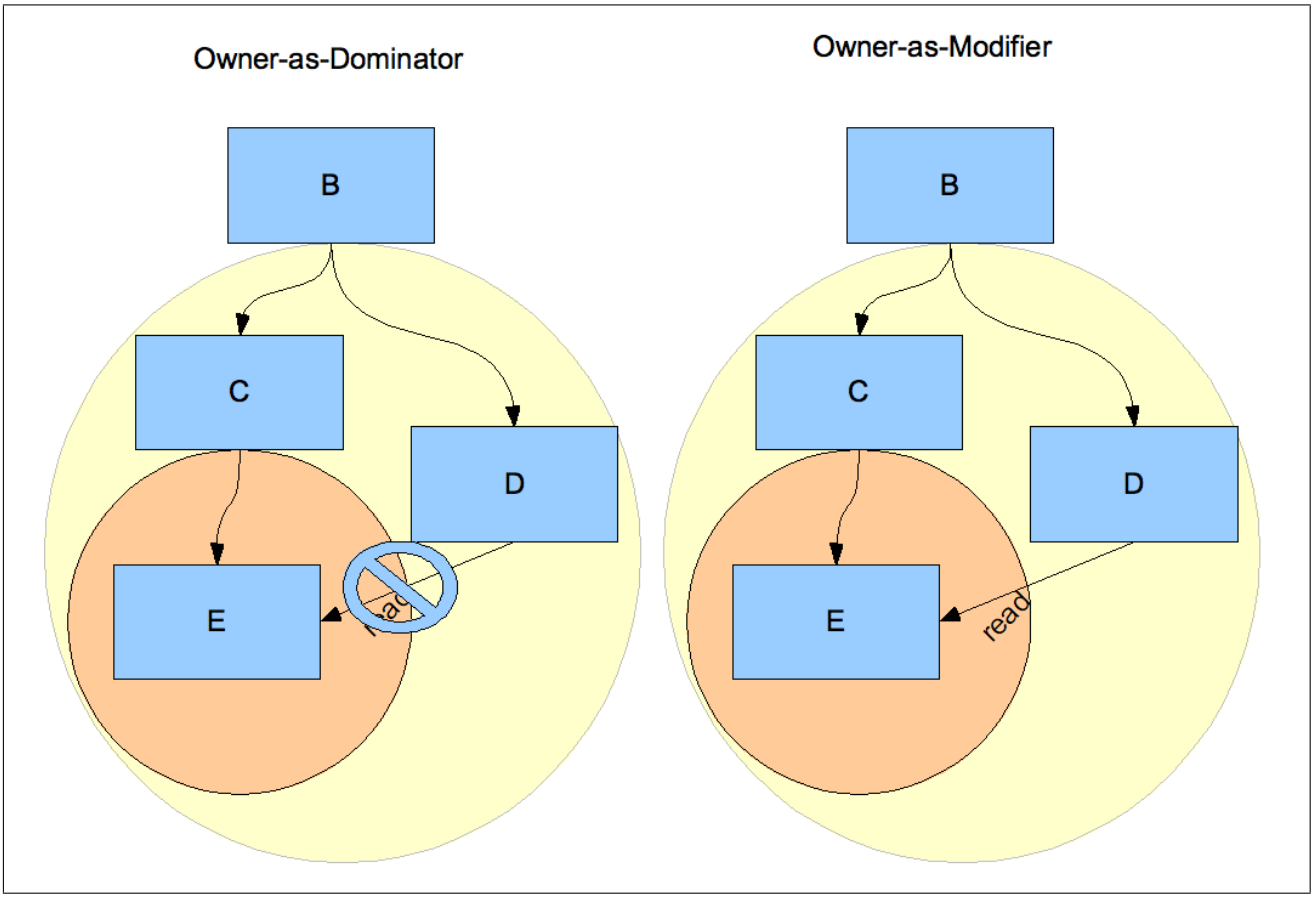


Figure 1.2: Owner-as-dominator vs Owner-as-modifier

1.2.3 Modifiers

To achieve those properties the type system has to be extended by a new syntax to give objects an ownership information. It is done by using the annotations *rep*, *peer* and any_u to describe the ownership relation for the types in expressions and field and method declarations of classes.
The *rep* annotation is used to describe that the variable is owned by the object in which the declaration is done. A *peer* modifier means that the variable and the class it is defined in have the same owner and any_u does not tell anything about the owner. A more specific explanation will be given during the formalization.
There are different ways to annotate the modifiers in the Java source code:

```
//Annotation of universe modifiers
class C{
        peer S a ; // Java keyword
        /*@ rep @*/ T b ; // JML specification
        /*@ any @*/ T c ; // JML specification, requires parsing
}
```

Besides those three ownership modifiers there are two internally used modifiers called $this_u$ and *unknown*. A variable with the $this_u$ ownership modifier is the current viewpoint. The *unknown* modifier is used by the adaptation of the viewpoint to represent a modifier that can not be expressed in the current viewpoint because it is a specific but unknown type. The primary use for the *unknown*

modifier is during field update and method call.
The underscored "u" in some modifiers are used to distinguish it from other parts of the formalization, because the variable name for the current viewpoint is called *this* and there also exists a runtime part of the "any" called any_a.
A detailed description of the universe modifiers can be found in an article from Dietl and Müller about ownership properties in JML [4].

1.2.4 Topological vs. Encapsulation Type System

In this formalization the Universe Type System will be split into two parts. One part is the topological type system, which guarantees that the object hierarchy is preserved and that an evaluated expression will result in a heap and an object respecting the expected type as result for the expression, also called type preservation.
The other part is the encapsulation type system, set on top of the topological one, which enforces the owner-as-modifier property by the use of pure and encapsulated expressions.

1.3 Isabelle/HOL

Isabelle/HOL is a tool, to assist in proving theorems using Higher Order Logic. It is being developed by Lawrence C. Paulson (University of Cambridge, UK) and Tobias Nipkow (Technical University of Munich, Germany). As it allows to introduce new notations, theorems can be written down in a way not too cryptical for a human to read and to understand and to keep the notations of prior work in most cases. Besides, there already exists a huge database of theorems and rules for different problems which can be used in the proofs[13].
Apart from using Isabelle/HOL to show mathematical proofs, it can be used to reason about semantics of a programming language and to prove interesting properties for this semantics. For this case the semantics of a programming language can be described in an abstract way in Isabelle/HOL using inference rules and inductive sets.
Isabelle/HOL is built on top of ML (Meta-Language). The theorem prover is written inside the ML environment. It is just the small core of Isabelle/HOL, the rest is shown as theorems using the theorem prover.
Those theorems are written down in a structured way and stored in so called ".thy" files, containing a requirement list of other theories, definition of functions, sets and types, and lemmas that have to be shown.
When a theory file will be proven using the requirements list a dependency tree of theories can be built with all required theories in it. It is not allowed to have loops in this tree.
There are two ways to work with Isabelle/HOL files. One way is a graphical front-end. It is called Proof General and is an extension to Emacs (Illustration 1.3). Using the proof general, the state of a proof can be shown, and during the proof steps you can walk along a proof to see which parts show what property. It is the normal environment to write down the proofs. The other way is the use of a command line tool to generate latex and pdf documents. To be able to generate a document out of a document all lemmas in a theory file have to be shown correct.
To show the correctness of a lemma, either a tactic script can be used, telling Isabelle/HOL in a "programming"-like way what theories and provers (like a simplifier) have to be applied to the lemma, or in a more "mathematical" or structured way using Intelligible Semi-Automated Reasoning (ISAR) [12]. It depends on the person which way he likes better, normally the structured proofs are more robust against changes, so for bigger proofs it would be a good idea to use ISAR. Special subgoals can be shown using tactic scripts.

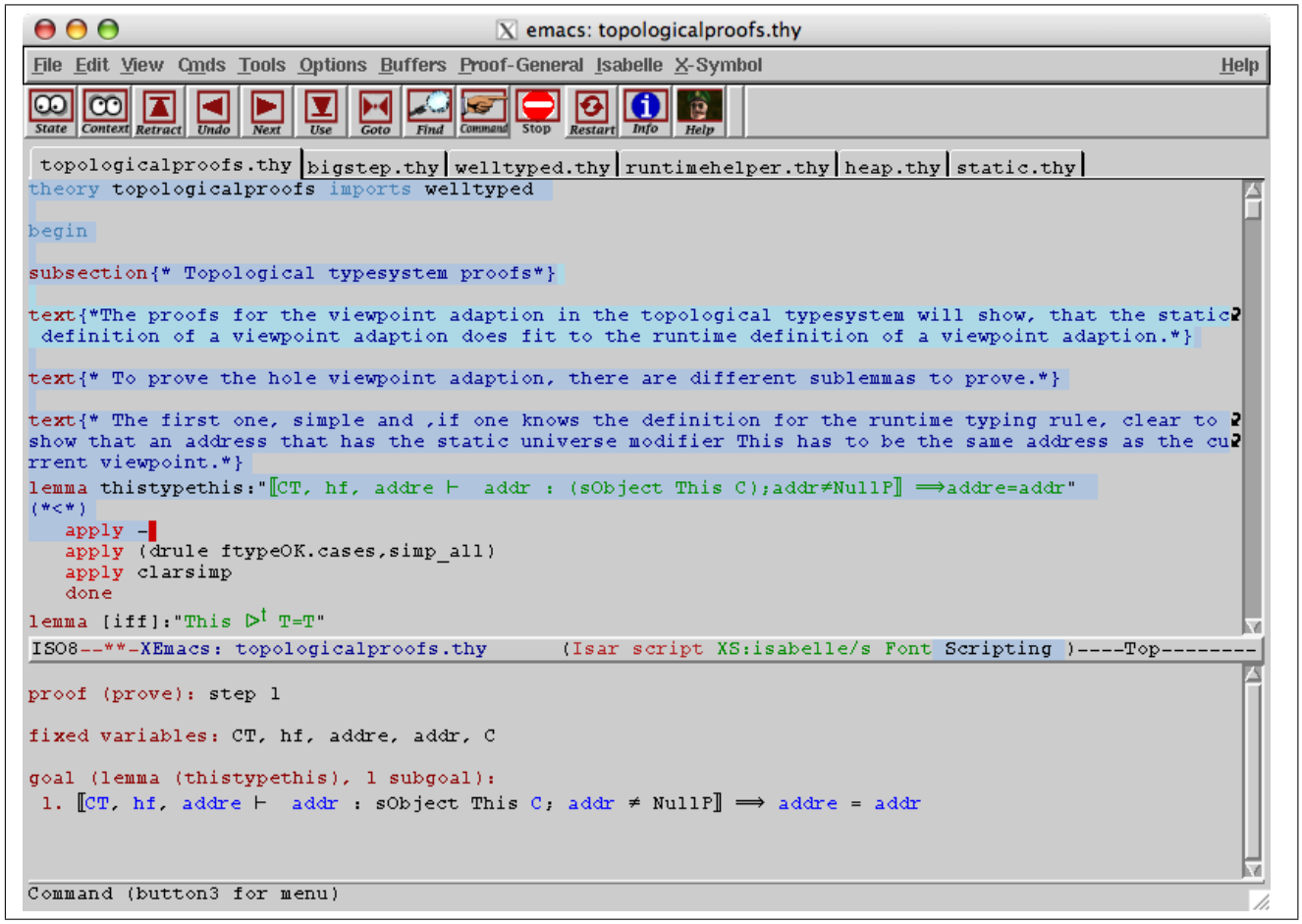


Figure 1.3: Isabelle/HOL - Proof General

1.4 Type systems and how to prove properties of them

A type is a description of the properties of a set of values. In Java, for example, there exist different types for numbers, strings or for classes.
A type system defines the types and how they are used in a programming language. Depending on the definition of a type system, properties could be assumed during runtime. Those are enforced through the type system statically.

```
// A not well typed assignment
int i=0;
char c='C';
i=c; //should be forbidden by the type system
```

One of those assertions is the type safety, meaning that in a well typed program, when an expression is evaluated the resulting object has the same or a more specific type then the type that expression will result by applying the typing rules to it. This is called type preservation. Besides that, a well typed program will not get stuck during program execution. A type system could also be used to optimize programs, because a compiler knows how to handle specific variables. For the formalization of the Universe Type System only the property of the type preservation is of interest. Depending on the type system other properties can be also enforced, like the owner-as-modifier property.

To guarantee those assertions, the type system has to be proven sound. This can be achieved by the following steps:

- Formalize the syntax and the typing rules.
- Formalize the semantics of expression evaluation.
- Formalize the assertions. Afterwards show them with respect to the definitions.

1.4.1 Featherweight Java

Featherweight Java [7] is a very small subset of Java using only the basic parts of it. Sometimes it is called the lambda calculus for object oriented programming. Featherweight Java consists only of object creation, field read, casting, and method calls. Neither once created objects can be modified, nor does there exist a sequence of expressions or conditions.
As Featherweight Java does not modify created objects, a reduction semantics without a heap can be used to express the evaluation of expressions.
It is a good point to start and create a new language containing only the basic features needed to prove properties about the Universe Type System. Because the interesting part for the Universe Type system is the modification of objects, an expression for field updates has to be added to Featherweight Java. Apart from the modification of objects, which could not be done by the reduction semantics, also the reduction of method calls would not be possible, because there exists no "*rep rep*" type. Thus the heap has to be modeled and a new semantics has to be defined. The result is a relatively simple programming language, which can be used for the formalization and the proofs.
There is already a formalization of Featherweight Java in Isabelle/HOL, and it can be used as a good starting point [5].

1.4.2 Jinja

In contrast to Featherweight Java Prof. Tobias Nipkow formalized a very expressive Java subset called Jinja. There are a lot of aspects of Java implemented. Because it is already formalized in Isabelle this formalization of the Universe Type System followed the formalization of Jinja in some parts [8]. I did not decide to use Jinja directly to formalize the Universe Type System because the expressiveness of Jinja makes it necessary to show a lot of properties besides the core properties.

1.5 Overview

The thesis will be split up into three main parts: in the second chapter the type system will be described semi-formally. This will be done by first specifying the topological part of the type system, together with the definition of the adaptation of the viewpoint and the well-typedness of the classes. Afterwards a heap will be modeled to be able to describe the semantics.
With those definitions the first interesting property, the type preservation, can be described.
Having the topological information it can be extended with some encapsulation rules. They can be used to describe the owner-as-modifier property.
In chapter three is the description of the extension, the integration of arrays into the type system. Topological as well as encapsulation parts will be both extended.
The formalization of all those descriptions using Isabelle/HOL is written down in chapter four.
At last a summary of the results will be done in chapter five.
The appendix describes where the adaptation function is derived from.

2 Type system

In this chapter it will be described how the type system will be formalized.
Based on that semi-formal notation the Isabelle formalization should be easier to understand.

Section 2.1 describes the static parts of the topological system. It is extended by a runtime model in section 2.2. With those information in section 2.3 the main lemmas are introduced.
At last in section 2.4 the type system will be encapsulated to guarantee the owner-as-modifier property.

2.1 Static checking

It would take much effort to formalize a parser to generate an abstract structure from source code, so the abstract structure that is defined will be used directly for the definition of the syntax and the proofs.

```
//Wrong syntax
class C wird abgeleitet von D{ // "wird abgeleitet von" is not a syntax element
    ...
}
```

As the type system should prevent you from using a "wrong" (in the context of this semantics) syntax, most of the checks on an expression should be done with this abstract structure to prevent errors that could otherwise happen in the runtime environment. An example would be the assignment of a string to an integer variable.

```
//Right syntax but not well formed in the semantics of Java
class C extends C{ // a class cannot extend its own class
    ...
}
```

2.1.1 Syntax

The abstract syntax (see Figure 2.2) used in this paper consists of different parts. There exist the namespaces $VarName$, $MethodName$ and $ClassName$ for the variables, methods and classes. Fields and environment variables will have the same namespace.
The type $Type^s$ consists of an ownership modifier and a class part required to represent the type in the system.

The first part, the ownership modifier OM, is used to represent the ownership relation of types. The second part, the class, describes the behavior of an object. That is done like in every other object oriented language. All classes together are stored in the $ClassTable$, which is represented by a partial function mapping $ClassName$s to class definitions. For each class in a $ClassTable$ there exists a class definition naming the superclass and a list of fields for the class together with a list of method definitions. Besides this, the class $Object$ is defined in the ClassTable as the root class

without any methods or fields.
The list of fields is a list of $Type^s$ and $VarName$ pairs, telling that the field with the defined name has a specific type.

A method is specified with different parts. It consists of the method's signature, containing argument names and types, a return type and an expression that will be evaluated when the method is called.

The expression can consist of different parts:

- The Null expression,
- An environment variable, having the variable name as argument,
- A field read, consisting of a target expression and a field name,
- A field update, with an expression for the receiver object, a field name, and an expression for the object to be assigned,
- A method call, consisting of an expression for the receiver, the method names, and expressions for all arguments,
- An object creation, with the type of the object as argument,
- A type cast, with the type, the object should be casted to.

Using those expressions a derivation tree can be built which can be used later on for the proofs (see figure 2.1.

There must be an environment to evaluate the expression to a static type.
The static environment Γ^s is represented as a partial function mapping variable names to types and a mapping from the the variable name $this$ to the type of the current viewpoint.

With those declarations a simple abstract syntax is defined (see figure 2.2) which can be used for the further work.
The syntax used to formalize the type system using Isabelle/HOL will be modified slightly to fit the requirements for Isabelle/HOL and to simplify the proofs in some cases.
Although the semantics does not have constructors, control flow elements or exceptions, it is still enough to show the interesting parts of the type system.

The syntax does not describe what a program looks like. For the formalization it is not of interest how to get to an initial environment on which the transitions can be applied on. The only interesting thing is what a well formed environment is. The definition of a well formed environment state, as well as a well formed runtime environment will be given later on.
If one wants to formalize a way to get to this initial environment he has to do the following things: Define a classtable. Then write down the triple classtable, classname and methodname, with the method having no arguments. By using the classname an initial root object can be created. Afterwards the method will be called in the context of that object.

Lookup functions

To make the inference rules in the formalization easier to read, some functions to look up information in the class table are required. At least the type of a field f in a class C has to be looked up, and

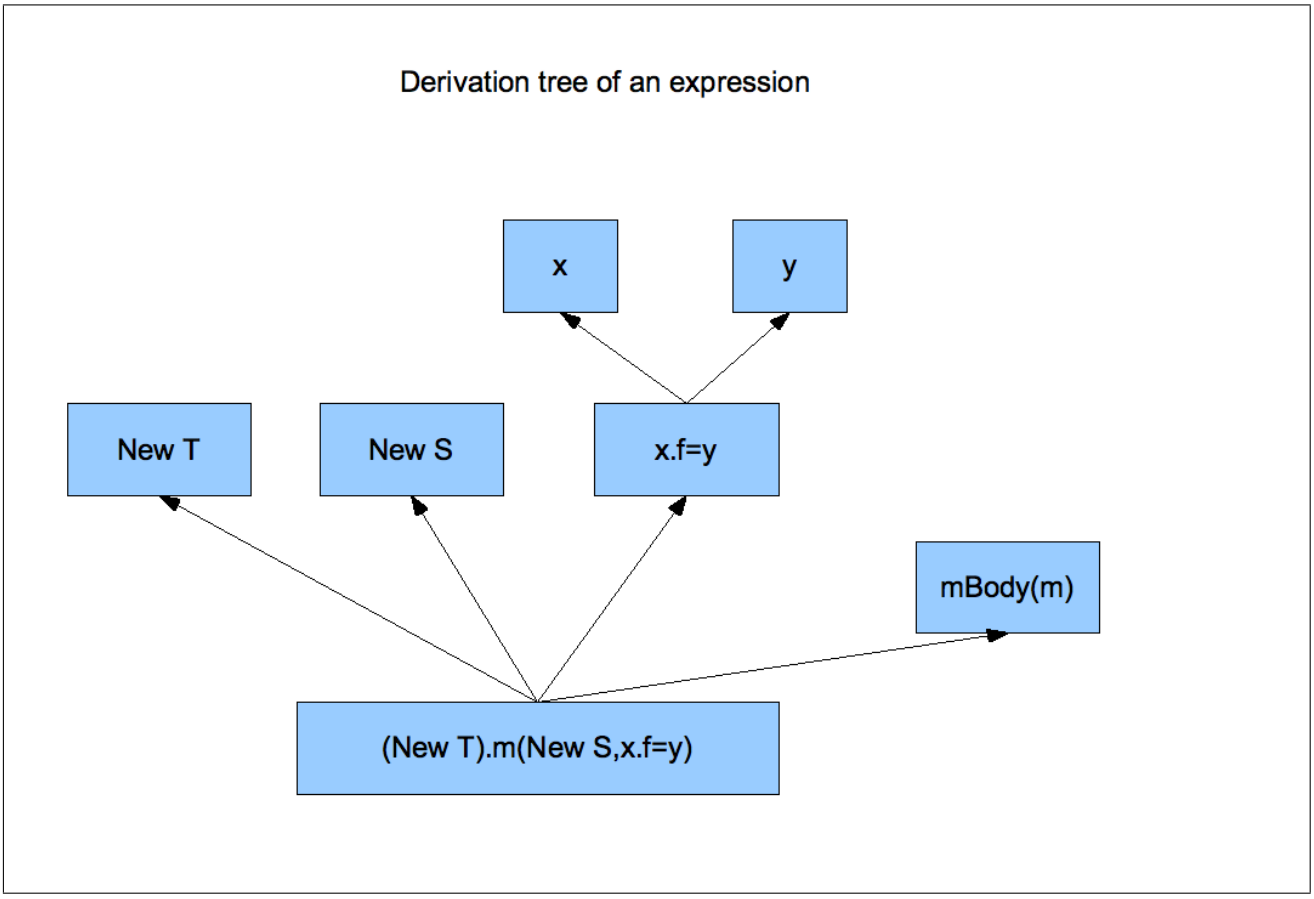


Figure 2.1: Derivation tree of an expression

the signature as well as the body of a method needs to be looked up frequently.
To get the super class of a class the super function will be used.

- $fType :: ClassTable \times ClassName \times VarName \rightarrow Type^s$
  $fType$ is a function to get the type of a field for a specific class. $fType$ is defined recursive so, it returns the type of the field in the class itself or, if not declared there, it returns the $fType$ of the superclass.

- $mType :: ClassTable \times ClassName \times MethodName \rightarrow (Purity, \overline{Type^s}, Type^s)$
  $mBody :: ClassTable \times ClassName \times MethodName \rightarrow (\overline{VarDef}, Expr)$
  The same rules apply to the function $mType$ which returns the signature of a method, containing the Purity, the argument types and the return type and for the function $mBody$, which returns the argument names and the expression of a method in a class.
  Both of these methods are defined recursively, too, so they will return a method definition either from the class itself or if not defined there of one of the super-classes.

- $super :: ClassTable \times ClassName \rightarrow ClassName$
  The function $super$ is used to extract the name of the superclass out of the class definition.

2.1.2 Viewpoint adaptation

As the universe modifier represents the ownership of a variable in the context of the current viewpoint, when those variables are used in another context by accessing them using field access, method

v, f	$\in$	$VarName$	$::=$	$Set\ of\ VarNames \mid varthis$
m	$\in$	$MethodName$	$::=$	$Set\ of\ MethodNames$
C, D, E	$\in$	$ClassName$	$::=$	$Set\ of\ ClassNames \mid object$
u	$\in$	OM	$::=$	$peer \mid rep \mid any_u \mid this_u \mid unknown$
w	$\in$	$Purity$	$::=$	$pure \mid nonpure$
S, T	$\in$	$Type^s$	$::=$	$OM\ ClassName$
$vDef$	$\in$	$VarDef$	$::=$	$Type^s\ VarName$
e	$\in$	$Expr$	$::=$	$null \mid$
				$VarName \mid$
				$Expr.VarName \mid$
				$Expr.VarName = Expr \mid$
				$Expr.MethodName(\overline{Expr}) \mid$
				$new\ Type^s \mid$
				$(Type^s)Expr$
$mDef$	$\in$	$MethodDef$	$::=$	$Purity\ Type^s\ MethodName\ (\overline{VarDef})\ \{return\ Expr\}$
$cDef$	$\in$	$ClassDef$	$::=$	$Class\ ClassName\ extends\ ClassName\ (\overline{VarDef},\ \overline{MethodDef})$
CT	$\in$	$ClassTable$	$::=$	$ClassName \rightarrow\ ClassDef$
Γ^s	$\in$	Env^s	$::=$	$VarName \rightarrow\ Type^s$

Figure 2.2: Syntax tree

invocation or field update, the ownership information has to be adapted to the current viewpoint. For this reason the infix function $\triangleright$ takes two arguments, the first one as the ownership modifier of the receiving object and the second one as the ownership modifier of the field in the context of that class. The information is adapted to the current viewpoint.
When the adaptation can not express all details of the required information (a *rep* field of a *rep* object for example can not be expressed as "*rep rep*" object) the universe modifier *unknown* is used to express it instead of using the ownership modifier any_u for these cases. *unknown* in a result with the adaptation to a viewpoint, which is required in the case of field update or method invocation, indicates that the adaptation is not allowed.
The $\triangleright$ function has the signature
$\_\triangleright\ \_ :: OM \times\ OM \rightarrow OM$.
Because most of the time the result should be a $Type^s$ again, the infix function $\triangleright^t$ accepts $Type^s$ as the second argument and returns a $Type^s$ with a similar rule. To do the adaptation step, the ownership modifiers of the arguments are adapted using the $\triangleright$ function, and the result together with the type of the second argument is the resulting type. $\_\triangleright^t\ \_ :: OM \times\ Type^s \rightarrow Type^s$
u $\triangleright^t$ (u' C) =:: (u $\triangleright$ u') C
The result of a small analysis of different papers with adaptation of the the viewpoint (see Appendix A) is a lookup table, which can be seen in figure 2.3:

2.1.3 Subtyping

The next step in the formal definition is to define the well typedness of the static system. Well typedness means that the defined syntax follows some specified rules. Those preconditions about the syntax should limit the syntax in a way, so that the conclusions, the type preservation and the owner as modifier property, can be shown.

$u \triangleright u'$	$peer$	rep	$unknown$	any_u
$this_u$	$peer$	rep	$unknown$	any_u
$peer$	$peer$	$unknown$	$unknown$	any_u
rep	rep	$unknown$	$unknown$	any_u
$unknown$	$unknown$	$unknown$	$unknown$	any_u
any_u	$unknown$	$unknown$	$unknown$	any_u

Figure 2.3: Viewpoint adaptation

Because a $Type^s$ consists of two parts, the ownership modifier and a class, subtyping respects both parts of a $Type^s$. It has to be distinguished between subclassing (the class hierarchy has a certain order), the order of the ownership modifier and subtyping.

Subclassing

Subclassing is defined like most object-oriented programming languages (See figure 2.4). A class is a subclass of its superclass [sc-1-super]. Subclassing is reflexive [sc-2-refl] and transitive [sc-3-trans]. As the root class $Object$ has no superclass there is no subclass X with $CT \vdash Object[: X$ and X is not Object.

$$\frac{super(CT, C) = D}{CT \vdash C[: D}\ \text{sc-1-super}$$

$$\frac{}{CT \vdash C[: C}\ \text{sc-2-refl} \qquad \frac{CT \vdash C[: D \quad CT \vdash D[: E}{CT \vdash C[: E}\ \text{sc-3-trans}$$

Figure 2.4: Subclassing

Order of the universe modifiers

$$\frac{}{rep \leq any_u}\ \text{OM-order-1} \qquad \frac{}{peer \leq any_u}\ \text{OM-order-2}$$

$$\frac{}{this_u \leq peer}\ \text{OM-order-3} \qquad \frac{}{unknown \leq any_u}\ \text{OM-order-4}$$

$$\frac{u \leq u' \quad u' \leq u''}{u \leq u''}\ \text{OM-order-trans} \qquad \frac{}{u \leq u}\ \text{OM-order-refl}$$

Figure 2.5: Order of the ownership modifiers

There exists an order of the universe modifiers (see inference rules 2.5 and the subtyping relation 2.6):

- rep specifies the ownership information of an object more specific than any_u, because rep tells one that the current viewpoint is the owner of the object and any_u tells you nothing.

- The same holds for *peer* and any_u, with the difference that *peer* tells you that it has the same owner as the current viewpoint.

- The current viewpoint has the universe modifier $this_u$. As *peer* tells one that an object has the same owner as the current viewpoint, the current viewpoint must also be of the ownership modifier *peer*. Thus $this_u$ must be more specific than *peer*.

- *unknown* is a special case. You could say it gives you as much information as any_u, but that is not exactly true. It tells you at least that the type could not be expressed, so it could possibly be a specific but unknown owner. Per definition *unknown* will be declared as more specific than any_u.

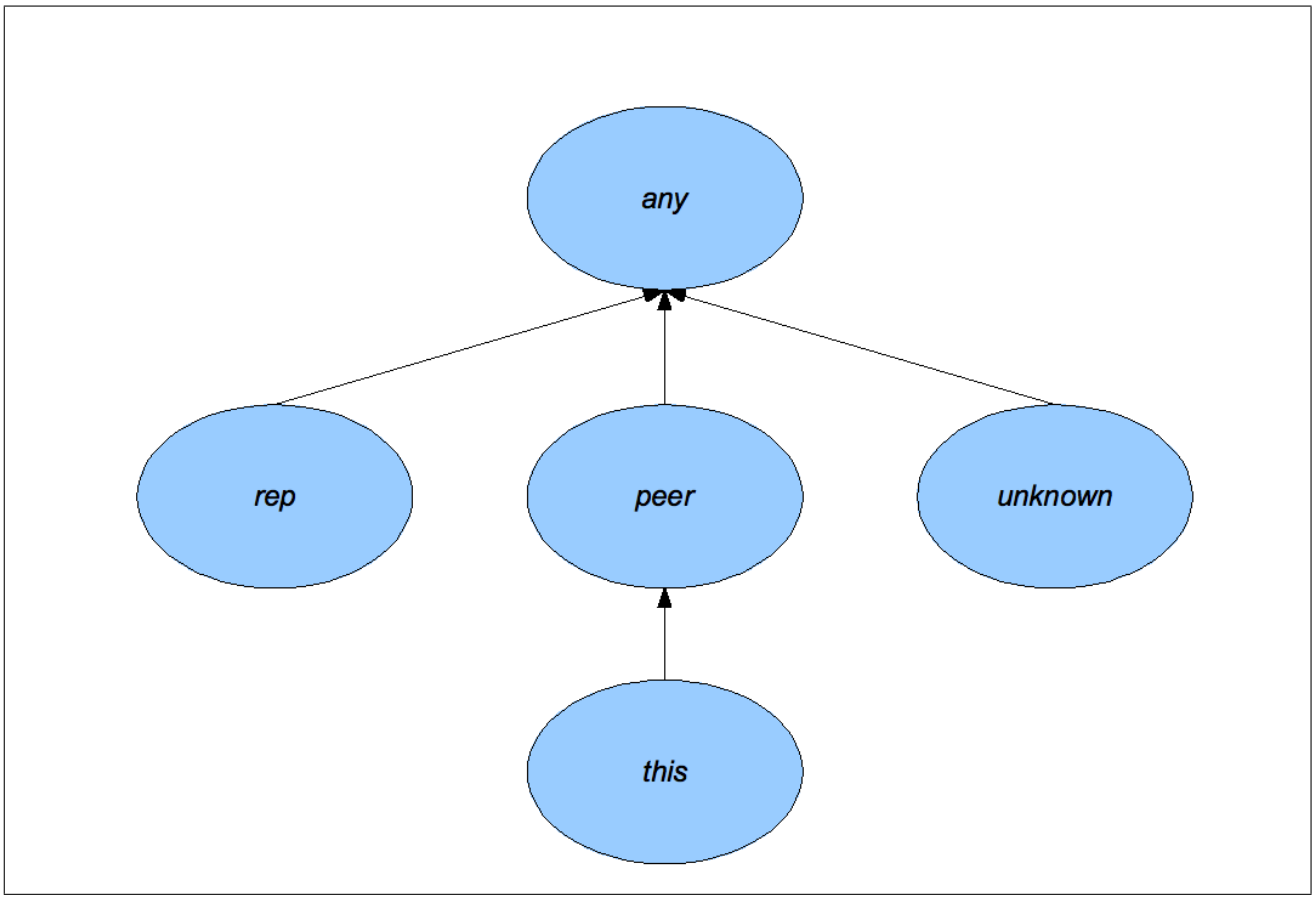


Figure 2.6: Subtyping of the universe modifer

Subtyping

As subclassing and the order of the ownership modifiers is specified, subtyping can be defined with those two declarations. It is a combination of subclassing and the order of the ownership modifiers and thus a type is a subtype of another type if and only if the class parts of the types are in a subclassing relation and the ownership modifier parts are in a sub-relation (Figure 2.7).
Subtyping will be used in the expression rules, to allow certain expressions that are more specific to be well typed. An example can be seen in the following source-code:

```
//Subtyping example, what is allowed and what is not allowed
Class C extends D {... }
Class D extends E {...}
 ...
D v;
v=new D() // OK types are equal
v=new C() // OK C is a subtype of D
v=new E() // ERROR, E is a supertype of D
```

As subclassing and the order of ownership modifier is reflexive and transitive, subtyping is reflexive and transitive too.

$$\frac{CT \vdash\ C[: D \qquad u \leq u'}{CT \vdash\ u\ C <: u'\ D}\ \text{st-1-subtype}$$

Figure 2.7: Subtyping

2.1.4 Type rules

For a type system it is required that an expression represents a type. For every characteristic of an expression there must exist a rule (compare figure 2.8) to get the type of the expression.

- The Null expression does not tell anything about a type, thus it could be any type.
- An environment variable has the type of the variable in the environment.
- A field read has the type of the field adapted to the current viewpoint using the type of the receiver object.
- A field update has the type of the left handed part, which can be handled like a field read. The right handed part has to evaluate to a subtype of this type. This subtype is the type of the field update. To guarantee type safety, there has to be an additional constraint, namely to forbid the ownership part of the type to be *unknown*.
- A method call has the type of the return value adapted to the current viewpoint. To be able to call a method all arguments have to be well typed. For this, the type of each expression for the arguments has to be a subtype of the corresponding argument type and the ownership part is not allowed to be *unknown*. To do this subtype check the type has to be adapted to the viewpoint of the method.

- An object creation has the type of the new object.
- A type cast results in the type an expression is cast to.

Those rules together with a subsumption rule complete the typing rules.
For every well typed expression the typing rules can match to a type. With the aid of the subsumption rule the expression would also match to every more general type. In contrast to Jinja for example, where in every expression that should allow subtyping it is a direct assumption, using the subsumption rule the subtyping has to be defined only once. The main reason for the decision for the subsumption rule was to keep in line with the formalization of generic universe types.

$$\frac{CT;\Gamma^s \vdash e : T' \qquad CT \vdash\ T' <: T}{CT;\Gamma^s \vdash e : T}\ \text{t-subs}$$

$$\frac{}{CT;\Gamma^s \vdash null\colon T}\ \text{t-null}$$

$$\frac{}{CT;\Gamma^s \vdash v : \Gamma^s(v)}\ \text{t-var}$$

$$\frac{CT;\Gamma^s \vdash e : u\ C}{CT;\Gamma^s \vdash e.f : u{\triangleright}^t\ fType(CT,C,f)}\ \text{t-read}$$

$$\frac{\begin{array}{rr} CT;\Gamma^s \vdash e_0 : u_0\ C_0 & u_0{\triangleright}^t\ fType(CT,C_0,f) = u\ C \\ u \neq\ unknown & CT;\Gamma^s \vdash e_1 : u\ C \end{array}}{CT;\Gamma^s \vdash e_0.f = e_1 : u\ C}\ \text{t-update}$$

$$\frac{\begin{array}{rr} CT;\Gamma^s \vdash e_0 : u_0\ C_0 & mType(CT,C_0,m) = (w,\ \overline{T_s},\ T) \\ u{\triangleright}^t\ \overline{T_s} = \overline{T_{su}} & u_0{\triangleright}^t\ T = T_u \\ \forall u_p\ C_p \in \overline{T_{su}}.u_p \neq\ unknown & CT;\Gamma^s \vdash \overline{e_s} : \overline{T_{su}} \end{array}}{CT;\Gamma^s \vdash e_0.m(\overline{e_s}) : T_u}\ \text{t-invk}$$

$$\frac{u\ \in\ \{peer, rep, any_u\}}{CT;\Gamma^s \vdash new\ u\ C : u\ C}\ \text{t-new}$$

$$\frac{CT;\Gamma^s \vdash e : T'}{CT;\Gamma^s \vdash (T)e : T}\ \text{t-cast}$$

Figure 2.8: Typing rules

2.1.5 Well-formedness rules

To guarantee that an expression could be executed in an environment some formal preconditions have to be specified. An environment with that formal preconditions is called well formed.

It can be defined by splitting up the declaration into parts and express a wellformdness of parts of the system (See Figure 2.9).
A type T is well formed [t-typing] if the class of the type is in the classtable.
As the type of fields [f-typing] should be preserved during subclassing; if the type of a field is defined in the superclass it should be the same in the subclass.
The term "$CT, C \vdash_m\ mDef$" expresses that $mDef$ is well formed in the class C. To achieve that

$$\frac{C \in\ dom(CT)}{CT \vdash_t\ uC}\ \text{t-typing}$$

$$\frac{CT \vdash_t\ T \qquad fType(CT, super(CT, C), f) = S \implies S = T}{CT, C \vdash_f\ Tf}\ \text{f-typing}$$

$$\frac{\begin{array}{c}\forall T_0 v \in \overline{vDef}.CT \vdash_t\ T_0 \qquad CT \vdash_t\ T \\ \Gamma^s = \overline{vDef}; this \longmapsto (this_u\ C) \qquad CT; \Gamma^s \vdash e : T \\ mType(CT, super(CT, C), m) = (w',\ \overline{vDef'},\ T')\ \} \implies \left\{\begin{array}{c} w \in\ \{Pure, w'\} \\ CT \vdash\ \overline{vDef'} <: \overline{vDef} \\ CT \vdash\ T <: T') \end{array}\right.\end{array}}{CT, C \vdash_m\ w\ T\ m\ (\overline{vDef})\ \{return\ e\}}\ \text{m-typing}$$

$$\frac{\begin{array}{c}\forall vDef_i \in \overline{vDef}.CT, C \vdash_f\ vDef_i \\ \forall mDef_i \in \overline{mDef}.CT, C \vdash_m\ mDef_i \\ D \in dom(CT)\end{array}}{CT \vdash_c\ Class\ C\ extends\ D\ (\overline{vDef},\ \overline{mDef})}\ \text{c-typing}$$

$$\frac{}{CT \vdash_c\ Class\ Object\ extends\ Object\ ([],\ [])}\ \text{c-typing-object}$$

$$\frac{\forall C \in dom(CT).CT \vdash_c\ CT(C)}{\vdash\ CT}\ \text{classes-typing}$$

Figure 2.9: Wellformedness rules

every argument has to be well formed , and if the method was already defined in the supertype it has to have the same signature. Besides this the expression in the method body must be a subtype of the type of the return value[m-typing].
"$CT \vdash_c cDef$" means that $cDef$ is well formed in the classtable CT. To be well formed every field and method has to be well formed, and the superclass has to be part of the classtable.
Besides this limitation the definition of *Object* [c-typing-object] has to have an empty field table and an empty method table.
A wellformed class table [classes-typing] consists of wellformed classes.

2.2 Runtime model

This section specifies the runtime model for the semantics.
The runtime model describes how the language behaves in the runtime environment containing a heap and a mapping for the local variables. To do that, the heap will be modeled. There will be specified what a welltyped runtime environment is and a semantics for executing an expression will be declared. Together with all the information given the type preservation can be shown.

2.2.1 Heap definition

The heap is an abstract definition of the memory of a computer where the program can allocate space. To be able to do that, the heap has a mapping from addresses to objects. In the abstract formalization the heap will be represented by a partial function mapping addresses to objects. Those

objects consist of the runtime type and the field-table of an object, which is a partial function mapping variable names to addresses.
Besides the definition of a heap the runtime environment has to be specified. It is a partial function mapping variable names to addresses located in the heap.
There are three special addresses in the Universe Type System.
One of them is $null_a$. It is used in the creation of new objects to map the fields to a valid address. It is the only address that can be assigned to fields or variables which is not in the heap.
In addition to that in the Universe Type System every object has an owner, which is represented by an address. A semi-formal definition can be seen in figure 2.10
The second one is $anyowner_a$, which is used during the creation of an any_u object, where it represents the owner address of that object.
Finally to distinguish between the $anyowner_a$ and an any_u type in a runtime type check, the any_a owner address represents the any_u type in runtime checks.

ι	$\in$	$Addr$	$::=$	$Set\ of\ addresses \mid null_a$
ι_o	$\in$	$OwnerAddr$	$::=$	$Addr \mid anyowner_a \mid any_a$
T^r	$\in$	$Type^r$	$::=$	$OwnerAddr\ Class$
V^r	$\in$	Var^r	$::=$	$VarName \rightarrow Addr$
Obj	$\in$	$Object^r$	$::=$	$Type^r\ Var^r$
h	$\in$	$Heap$	$::=$	$Addr \rightarrow Object$
Γ^r	$\in$	Env^r	$::=$	Var^r

Figure 2.10: Heap definition

2.2.2 Auxiliary functions for the runtime environment

To define welltypedness and the bigstep semantics in the runtime environment, and to make the inference rules simpler to read, the following functions are required.

- $owner :: Heap \times Addr \rightarrow OwnerAddr$
  The owner function is used to extract the owner from the heap. It is done by fetching the object from the heap and reading the owner from the runtime type definition of the object.
- $sowner :: Heap \times Addr \times OwnerAddr \rightarrow Bool$
  The set of owners function sowner represents a relation from addresses to owners. It can be described as a set of the direct owners extended with transitivity. It is required to show the owner-as-modifier property.
- $dynType :: Heap \times Addr \times Type^s \rightarrow Type^r$
  Using dynType a static type can be transformed into a dynamic type. It replaces $peer$ and $this_u$ with the owner of the current viewpoint, rep with the address of the current viewpoint, and any_u or $unknown$ with any_a. The current viewpoint is the address given to the function as second parameter (See Figure 2.11).
- $aClass :: Heap \times Addr \rightarrow Class$
  In some definitions the class of an object is required. To reduce the writing work, the aClass functions gets the class to a corresponding address from the heap.
- $\_ \vdash \_ <:_r \_ :: ClassTable \times Type^r \times Type^r \rightarrow Bool$
  Besides the static subtyping rules there also exists a corresponding counterpart for the runtime

types. To be runtime subtypes, the class part has to be in a subclassing relation. For the owner-part, on both sides there is the same owner, or in the case of an any type, on the right handed side the owner is any_a.

- $\_,\_,\_ \vdash \_ : \_ :: ClassTable \times\ Heap \times\ Addr \times\ Addr \times\ Type^s \rightarrow Bool$
  For the type preservation lemma it is useful to have a definition of as what type an address (the current viewpoint) sees another address. If you read the definition "$CT, h, \iota_{me} \vdash \iota : T$" you could say "In the heap h, ι_{me} sees ι as static type T".

- $\_,\_ \vdash \_ : \_ :: ClassTable \times\ Heap \times\ Env^r \times\ Env^s \rightarrow Bool$
  One of the last things to be done is to describe when a runtime environment fits to a static environment. It is the case when all variables in the static environment have a counterpart in the runtime environment: in the case it is not a null-pointer, the type of the object at the result address should match to the static type. The address for the current-viewpoint is given with the runtime environment, being the address *this* is mapping to.

- $declareFields :: Class \rightarrow Var^r$
  At last there is a function to simplify the step of the object creation. It takes the classname and as result will return the runtime field definition, where all fields are mapping to the $null_a$.

$$\frac{}{dynType(h, \iota_{me}, Object_t\ rep\ C) = Object_r\ \iota_{me}\ C}\ \text{dynType-object-rep}$$

$$\frac{u \in\ \{peer, this_u\}}{dynType(h, \iota_{me}, Object_t\ u\ C) = Object_r\ owner(h, \iota_{me})\ C}\ \text{dynType-object-peer}$$

$$\frac{u \in\ \{unknown, any_u\}}{dynType(h, \iota_{me}, Object_t\ u\ C) = Object_r\ any_a\ C}\ \text{dynType-object-any}$$

Figure 2.11: The dynType function

2.2.3 Well typedness of the runtime environment

The next step is to define what a well formed runtime environment is (compare with figure 2.12). It is required to guarantee some preconditions that are needed in the further proofs. The elements of the runtime environment are:

- The addresses
- The objects
- The heap
- The mapping for local variables

A wellformed address or object [wt-a] consists of valid fields and a valid owner.
A field of an object has a static type T in the class the object has. A field is valid if and only if the address the field maps to is well formed, and if that address is a subtype of the static type T.

In a well formed heap [wt-h] all addresses are well formed, and the address $null_a$ is not in the

domain of the heap.

A well formed environment Γ^r [wt-env] needs a well formed heap and all elements in Γ^r have to be well formed, too.

$$\frac{h(\iota) = T^r\ V^r \quad owner(h,\iota) = \iota_o \quad \iota_o \in dom(h) \vee\ \iota_o \in \{null_a, anyowner_a\} \quad aClass(h,\iota) = C \quad \forall(f\ \iota_f\ \in\ V^r.CT, h, \iota\ \vdash \iota_f : fType(CT, C, f)}{CT, h \vdash_a\ \iota}\ \text{wt-a}$$

$$\frac{\forall \iota\ \in dom(h).CT, h \vdash_a \iota \quad null_a \notin dom(h)}{CT \vdash_h\ h}\ \text{wt-h}$$

$$\frac{CT_h \vdash\ h \quad \forall(v) \in dom(\Gamma^r).CT, h \vdash_a\ \Gamma^r(v)\ \vee \Gamma^r(v) = null_a}{CT, h \vdash_w\ \Gamma^r}\ \text{wt-env}$$

Figure 2.12: Wellformed model

With those conditions, a valid runtime-environment can be described. They are enough to prove the adaptation from a viewpoint and the adaptation to a viewpoint, which will be shown later on.

2.2.4 Bigstep semantics

Once a valid state for a program is specified, the execution of a program has to be described. There are two possibilities. One of them is to define a small-step semantics, where only the state transitions from one state to the next state are written down. It is called small-step because everyone of those steps evaluates only a very small part of a complete expression. After a small-step the result will be a new environment and (if not completed) a new expression that has to be evaluated.
Besides that, there exists the bigstep semantics, where the whole expression is executed as one single big step. Thus the result is only a new heap and a resulting address.
As proving properties is normally easier with a bigstep semantics and there is no need to show progress, a bigstep semantics can be used to define how the program execution behaves. For this we have to define one bigstep for every possible expression type(compare 2.13).
With the evaluation of a bigstep, a derivation tree will be built. That derivation tree can be used to run a proof on it.
In the case of the specified expressions there exist the following big-steps:

- The null expression where nothing is done, only the null-pointer will be returned
- An environment variable, where the address is read out of the current environment. This is done by looking up the name in the environment.
- A field read $e.f$, where the expression is evaluated and afterwards there is done a lookup in the field-table of the object for f.
- A field update $e1.f = e2$, where the expressions are evaluated, afterwards the field-table of the left-handed object is updated.

- A method call $e.m(es)$, where the expressions are evaluated, afterwards a new environment is created, the body expression of the method m is fetched and then the bigstep is done over the expression of the method
- An object creation $NewT$, to create a new object. Using the type T, a new object is created in the heap.
- A type cast $(T)e$, where the expression is evaluated and, after that, the semantics tests if the resulting object has a subtype of the dynamization of T.

$$\frac{}{CT \vdash h, \Gamma^r, \text{null} \rightsquigarrow h, null_a} \text{ os-null}$$

$$\frac{}{CT \vdash h, \Gamma^r, v \rightsquigarrow h, \Gamma^r(v)} \text{ os-var}$$

$$\frac{\begin{array}{c}\iota \notin dom(h) \\ u \neq any \implies h' = h[\iota \mapsto (dynType(h, \Gamma^r(\text{this}), (u\ C))\ declareFields(C))] \\ u = any \implies h' = h[\iota \mapsto (anyowner_a\ C)\ declareFields(C))]\end{array}}{CT \vdash h, \Gamma^r, new\ u\ C \rightsquigarrow h', addr} \text{ os-new}$$

$$\frac{CT \vdash h, \Gamma^r, e \rightsquigarrow h', \iota \qquad CT, h, \Gamma^r(\text{this}) \vdash\ \iota : T}{CT \vdash h, \Gamma^r, (T)e \rightsquigarrow h', \iota} \text{ os-cast}$$

$$\frac{CT \vdash h, \Gamma^r, e \rightsquigarrow h', \iota_e \qquad h'(\iota_e) = (T^r\ V^r) \qquad V^r(f) = \iota}{CT \vdash h, \Gamma^r, e.f \rightsquigarrow h', \iota} \text{ os-read}$$

$$\frac{\begin{array}{c}CT \vdash h, \Gamma^r, e_0 \rightsquigarrow h', \iota_0 \qquad CT \vdash h', \Gamma^r, e_1 \rightsquigarrow h'', \iota_1 \\ h''(\iota_0) = T^r\ V^r \\ h''' = h''[\iota_0 \mapsto (T^r\ V^r[f \mapsto \iota_1])]\end{array}}{CT \vdash h, \Gamma^r, e_0.f = e_1 \rightsquigarrow h''', \iota_1} \text{ os-update}$$

$$\frac{\begin{array}{c}CT \vdash h, \Gamma^r, e_0 \rightsquigarrow h', \iota_0 \qquad CT \vdash h', \Gamma^r, \overline{e_s} \rightsquigarrow h'', \overline{\iota_s} \\ mBody(CT, aClass(h, \iota_0), m) = (\overline{v_s\ T_s}, e_1) \\ \Gamma^r_{new} = [\overline{v_s}[\mapsto]\overline{\iota_s}][\text{this}\mapsto \iota_p] \\ CT \vdash h'', \Gamma^r_{new}, e_1 \rightsquigarrow h''', \iota_1\end{array}}{CT \vdash h, \Gamma^r, e_0.m(\overline{e_s}) \rightsquigarrow h''', \iota_1} \text{ os-invoke}$$

Figure 2.13: Bigstep semantics

As already told in the beginning, there is no need for the proofs to initialize a new runtime environment consisting of a classname, classtable and an expression. So it will not be formalized. To show it in a nutshell: to create an initial state out of the classname and an expression, a new object has to be created in the heap using that classname which will be owned by the root owner. Besides that the environment exists only of the mapping from $this_u$ to the newly created object. With those information in mind a big-step can be done.

2.3 Topological proofs

In the following section a few theorems will be introduced, and a description how to prove them. First some lemmas are introduced there that will be used in the later proof. Afterwards the type-preservation will be specified.

2.3.1 Correct viewpoint adaptation

Ownership information is expressed with respect to a viewpoint. When the viewpoint changes, the ownership information has to be adapted.
On the one side the adaptation from some viewpoint to *this* is needed. It is done directly by the $\triangleright^t$ function for two objects at the addresses ι_1 and ι_2. There is also the environment Γ^r where *this* maps to the current viewpoint denoted as ι_{this}. For the definitions it will be assumed that the addresses are not $null_a$. The lemma can be seen in figure 2.14.

On the other side the adaptation to a viewpoint is sometimes required, for example in the

$$\left.\begin{array}{c} CT, h, \iota_{this} \vdash\ \iota_1 : (u\ C) \\ CT, h, \iota_1\ \vdash\ \iota_2 : T_2 \end{array}\right\} \Longrightarrow \{\ CT, h, \iota_{this}\ \vdash\ \iota_2 : u \triangleright^t T_2$$

Figure 2.14: Adaptation from a viewpoint

case of field updates. For those cases there has to be guaranteed that the universe modifier of the receiver object does not have an *unknown* universe modifier in the current viewpoint(Look at figure 2.15).

$$\left.\begin{array}{c} CT, h, \iota_{this}\ \vdash\ \iota_1 : (u\ C) \\ CT, h, \iota_{this}\ \vdash\ \iota_2 : u \triangleright^t (u'\ D) \\ u' \neq\ unknown \end{array}\right\} \Longrightarrow \{\ CT, h, \iota_1\ \vdash\ \iota_2 : (u'\ D)$$

Figure 2.15: Adaptation to a viewpoint

To proove those lemmas, there has to be done a case distinction by the ownership modifiers of t_1 and t_2. This will result in 25 cases, some of them take some writing work to prove, but it is not really too complicated.

2.3.2 Type preservation

Because the semantics is only partially defined and there exist no exceptions, progress can not be shown. That should not be a problem. The Universe Type System should not modify the property of a language to guarantee progress for a welltyped program. There already exist many formalizations of Java that show using properly defined exceptions it is possible to guarantee progress with that semantics. Thus it could be assumed that the progress property could be easily integrated into the type system.
Only a weaker definition of type safety will be shown, where in a well typed expression, which is evaluated with a big step to a state and a result address, the new state is well typed too, and the address is in the new well formed heap.
This prohibits errors like assigning a character to an integer variable.

It will be expressed by the theorem 2.16:

Using a welltyped classtable and runtime environment fitting to a static environment, and having and expressions that results in that static environment in a specific type T than if a bigstep is

done with that expression the resulting environment is welltyped and the result address will be a subtype of the static type T. The type-preservation can be proven by rule induction over the shape

$$\left.\begin{array}{c} \vdash CT \\ CT \vdash h, \Gamma^r, e \rightsquigarrow h', \iota \\ CT, h \vdash_w \Gamma^r \\ CT; \Gamma^s \vdash e : T \\ CT, h \vdash \Gamma^r \ : \Gamma^s \end{array}\right\} \Longrightarrow \left\{\begin{array}{c} CT, h' \vdash_w \Gamma^r \\ CT, h', \Gamma^r(\text{this}) \vdash \iota \ : T \end{array}\right.$$

Figure 2.16: Type safety

of the derivation tree of the bigstep semantics. For the cases fieldread and method invocation the adaptation from a viewpoint is required, the case fieldupdate and method invocation need the lemma for the adaptation to a viewpoint.
The complete prove is done using Isabelle, and can be either looked up in the Isabelle sources or in parts later in the work.

2.4 Encapsulation system

As the type system supports type preservation, the encapsulation of the type system should be now enforced. To achieve that goal the expression and the classtable have to have some additional limitations.
One of them is the purity of methods.
A method is pure, if it does not modify existing objects in the heap and does only call pure methods. For inheritance it has to be guaranteed that if a superclass is pure, all the subclasses have to be pure. As purity checks need to analyze the expression a new writing will be introduced: "pattern" $\in$ e means that there is a subexpression of the form of the pattern in the expression e.
To give an example: Exp.Name=Exp $\in$ e matches all field updates so the expression e_r.c(e_z.y=e_y) would match because the pattern "Exp.Name=Exp" matches with the sub-expression "e_z.y=e_y".
To complete the encapsulation of a class, all methods have to be encapsulated being either a pure method with a pure expression or a normal method with an encapsulated expression. For an encapsulated classtable that means that all classes have to follow that encapsulation limitations (see figure 2.17).

Encapsulation

With the purity and encapsulation information the owner-as-modifier property of the type system can be described (compare with figure 2.18):

The owner-as-modifier property expresses that in an encapsulated expression, where *obj* is an object at address ι and the fields of *obj* in h' differ from the fields of *obj* in h, the object *obj* is modified. Because of encapsulation this means that the owner of the current viewpoint has to be a direct or indirect owner of the object *obj*. It expresses the same as the theorem 5.4 in the GUT paper (owner-as modifier) property where for the preconditions we can show that for all objects in the heap either the object is unmodified or the owner of $\Gamma^r(this)$ is a transitive owner of the modified object.

$$\frac{mType(C,m) = (pure,\ \overline{t_s},\ t) \qquad \forall (u\ C) \in \overline{t_s}.u = any}{pure(C,m)}\ \text{pure Function}$$

$$\frac{\begin{array}{c} e_1.x = e_2 \notin e \\ \forall e_1.m(\overline{e_s}) \in e.(CT; \Gamma^s \vdash e_1 : u\ C \wedge pure(C,m)) \end{array}}{CT, \Gamma^s \vdash_p\ e}\ \text{pure Expression}$$

$$\frac{\begin{array}{c} \forall e_1.f = e_2 \in e.CT; \Gamma^s \vdash e_1 : u\ C \implies u = peer\ \vee\ u = rep \\ \forall e_1.m(\overline{e_s}) \in e.\ CT; \Gamma^s \vdash e_1 : u\ C \implies u = peer\ \vee\ u = rep\ \vee pure(C,m) \end{array}}{CT, \Gamma^s \vdash_e\ e}\ \text{encapsulated Expression}$$

$$\frac{\begin{array}{c} \Gamma^s = \overline{vDef}; this \longmapsto (this_u\ C) \\ w = Pure \implies CT, \Gamma^s \vdash_p\ e \\ w \neq\ Pure \implies CT, \Gamma^s \vdash_e\ e \\ w \neq\ Pure \vee \forall (v(uC)) \in \overline{vDef}.\ u = any \end{array}}{CT, C \vdash_{em} w\ T\ m\ (\overline{vDef})\ \{return\ e\}}\ \text{encapsulated m-typing}$$

$$\frac{\forall\ mDef_i \in \overline{mDef}.CT, C \vdash_{em}\ mDef_i}{CT \vdash_e\ Class\ C\ extends\ D\ (\overline{vDef},\ \overline{mDef})}\ \text{encapsulated-c-typing}$$

$$\frac{\forall\ cDef \in dom(CT).CT \vdash_e\ cDef}{\vdash_e\ CT}\ \text{classes-typing}$$

Figure 2.17: Encapsulation rules

$$\left.\begin{array}{c} \vdash CT \\ \vdash_e CT \\ CT, \Gamma^s \vdash_e e \\ CT \vdash h, \Gamma^r, e \rightsquigarrow h', addr \\ CT, h_w \vdash \Gamma^r \\ CT; \Gamma^s \vdash e : T \\ CT, h \vdash \Gamma^r\ : \Gamma^s \end{array}\right\} \implies \left\{\begin{array}{c} \forall \iota\ \in dom(h). \\ fields(h,\iota) = fields(h',\iota) \vee \\ sowner(h,\iota,\Gamma^r(this)) \end{array}\right.$$

Figure 2.18: Owner-as-Modifier property

3 Formalization of arrays

After the complete formalization of the basic type system it is extended by arrays.
This will be done in the following chapter. At first in section 3.1 an array definition is introduced. In section 3.2 it will be described how to extend the static checking. The same will be done for the runtime model in section 3.3.

3.1 New definition of arrays

Arrays are a core element in software programs and a universe implementation of arrays is already integrated into the Java Modeling Language (JML) [4].
To formalize them, there has to be specified how the universe modifiers in the type definitions of arrays are interpreted.
One way as done in the JML tools is to use two universe modifiers. This former specification of universe types for arrays will be called old system. For the formalization another interpretation of the universe types for arrays will be used, which will be called the new system. To understand how they work i want to give an example:

```
class Example {
        rep peer array[] f;
}
```

In the old style system this means the following: the first modifier, the *rep* tells you that f is owned by the *Example* object, the object that is created during runtime out of the *Example* class. Besides this, all elements in the array f have a *peer* owner in relation to the array. Because the array is *rep* all elements would also be owned by the *Example* object.

In concurrence to the old system both ownership modifiers describe in the new system the relation to the class they are defined in (see figure 3.1 for a better understanding). For the array itself this does not change the meaning, because it is still owned by the *Example* object.
But in contrast to the old system, the elements of the array are not owned by the *Example* object, but by the owner of the *Example* object.
To summarize this:
In the old system the modifier for the elements describe the relation to the array, in the new system to the current viewpoint.
Figure 3.2 will show the relation of the old and the new system. The only case, where the new system results in loss of information is the case with an any_u array, where all elements of the array have the same owner as the array. This loss should not be very problematic.

For the formalization it is also allowed to use different universe modifiers in the type definition of the array elements. An example for that would be the type "$rep\,array\ <peer\,array\ <rep\,Class\,C>>$".
To transform the simple schema into the more complex schema a transformation rule can be used. It is explained in figure 3.3.

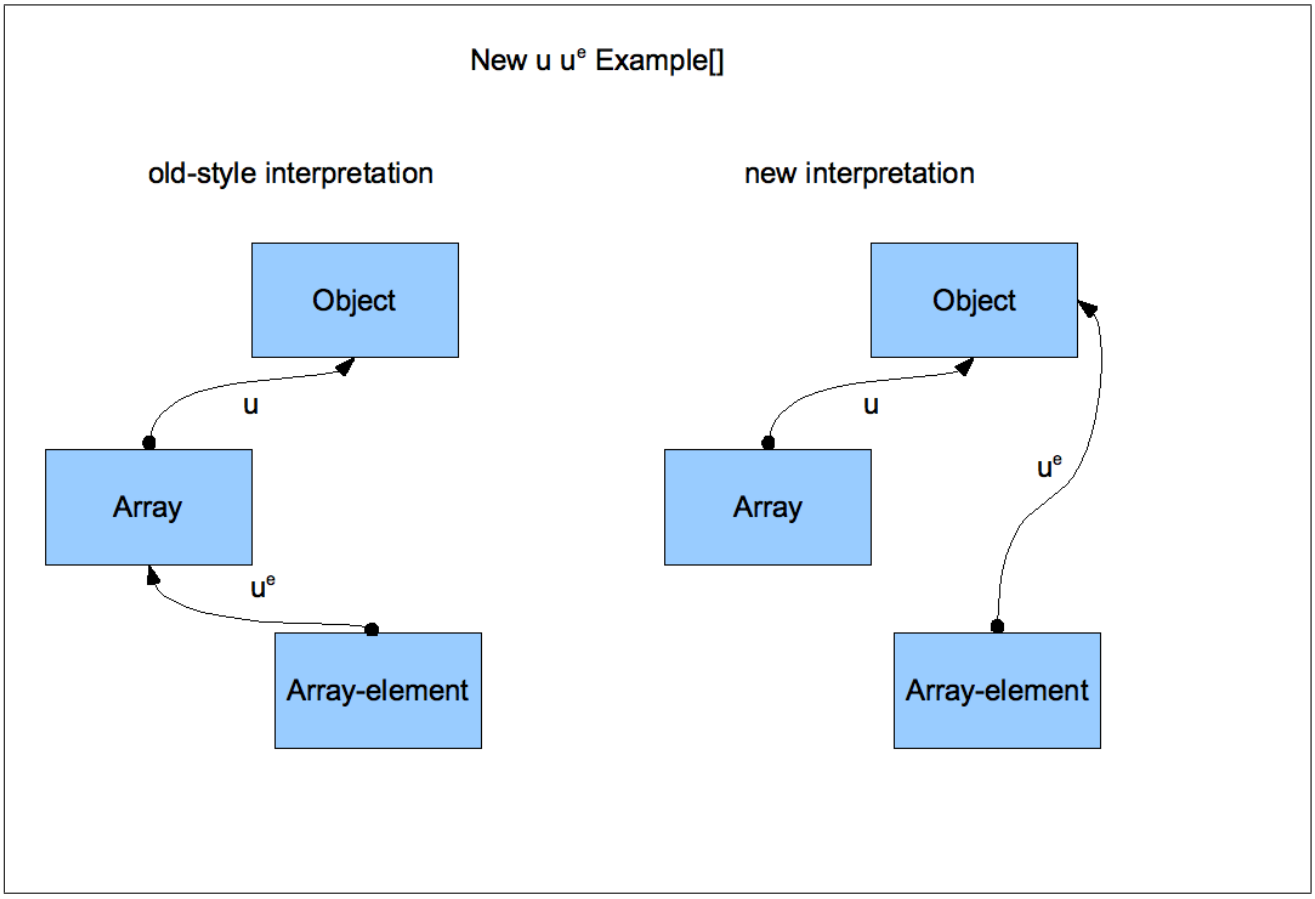


Figure 3.1: Old and new definition of arrays

There are three new expression types required to be formalized:

- Creation of a new array
- Read from an array
- Update of an array

.

3.2 Static checking

3.2.1 Syntax

As arrays should be supported by the language, the syntax has to be extended. The changes have to be done in the definition of a type, which has to be an inductive definition, an index number for the arrays, and the extension of the expressions with the three cases (see extended syntax in 3.7). Besides extending the syntax there is also a new function required which tests, if any of the ownership modifiers in a type definition T has the value *unknown*.

Old system	New system
any any [] *f*	*any any* [] *f*
not defined	*any peer* [] *f*
not defined	*any rep* [] *f*
peer any [] *f*	*peer any* [] *f*
peer peer [] *f*	*peer peer* [] *f*
not defined	*peer rep* [] *f*
rep any [] *f*	*rep any* [] *f*
not defined	*rep peer* [] *f*
rep peer [] *f*	*rep rep* [] *f*
any peer [] *f*	*any any* [] *f*

Figure 3.2: Mapping old to new array system

$$
\begin{aligned}
&u\ u'\ []\ Class\ C \implies u\ array\ <u'\ Class\ C> \\
&u\ u'\ []\ []\ Class\ C \implies u\ array\ <u'\ array\ <u'\ Class\ C>> \\
&u\ u'\ []\ []\ []\ ClassC \implies u\ array\ <u'\ array\ <u'\ array\ <u'\ Class\ C>>> \\
&...
\end{aligned}
$$

Figure 3.3: Transformation rules for array definition

$notUnknown :: Type^s \rightarrow bool$
If it does not have any *unknown* in its definition, true will be returned, else false will be returned. It is a recursive function and applies to all $Type^s$s. In the prior formalization the type was not defined inductive and thus the universe modifier can be looked up directly with an equal or unequal.

3.2.2 Viewpoint adaptation

To handle the new definition of $Type^s$s, the function for the adaptation of a viewpoint has to be changed (see rule 3.5).

As seen in the paper about generic universe types, there are some cases where some ownership modifier in the bounding definitions of the type variables are not allowed during viewpoint adaptation.
There are some equivalent limitations in the specification of arrays. For example the adaptation *peer* $\triangleright^t$ Array$_t$ *peer* <Object$_t$ *rep* C> should result in a type that can not be expressed in the current viewpoint, and because of that, it should be forbidden in a method call by the use of the *notUnknown* function.
Using the new adaptation function the type would result in
Array$_t$ (*peer* $\triangleright$ *peer*) < *peer* $\triangleright^t$ (Object$_t$ *rep* C> or at last
Array$_t$ (*peer*) <Object$_t$ *unknown* C>.
Using notUnknown(Array$_t$ (*peer*) <Object$_t$ *unknown* C>) would result in false.

3.2.3 Subtyping

The subtyping rules for arrays in Java do not guarantee type preservation by the use of static checks. To make life more easy for the programmer, the subtyping rules for arrays are not too strict and are handled in a covariant subtyping of the element types. Because of that subtyping behavior, a runtime check for subtyping has to be done during runtime. That means for normal arrays that the

$$
\begin{array}{llll}
n & \in & Index & ::= \ Natural\ numbers\ with\ zero \\
T & \in & Type^s & ::= \ OM\ Class \mid \\
 & & & \quad Array_t\ OM\ <Type^s> \\
vDef & \in & VarDef & ::= \ Type^s\ VarName \\
e & \in & Expr & ::= \ null \mid \\
 & & & \quad VarName \mid \\
 & & & \quad Expr.VarName \mid \\
 & & & \quad Expr.VarName = Expr \mid \\
 & & & \quad Expr.MethodName(\overline{Expr}) \mid \\
 & & & \quad new\ OM\ Class \mid \\
 & & & \quad (Type^s)Expr \mid \\
 & & & \quad new\ Array_t\ OM\ Type^s\ Index \mid \\
 & & & \quad Expr[Index] \mid \mid \\
 & & & \quad Expr[Index] = Expr
\end{array}
$$

Figure 3.4: Syntax extended with arrays

$$
\begin{array}{l}
\_ \triangleright^t \ \_ :: OM \times \ Type^s \rightarrow Type^s \\
u \triangleright^t u'\ C = (u \triangleright u')\ C \\
u \triangleright^t \text{Array}_t\ u'\ T = \text{Array}_t\ (u \triangleright u')\ (u \triangleright^t T)
\end{array}
$$

Figure 3.5: Viewpoint adaptation with arrays

class-type of arrays has a covariant subtyping.

```
\\ Example that can not be handled by a static typecheck:
class D extends C {
  ...
}
class Example{
   void mytest(C[] myarray){
        myarray[0]=new D; //OK
        myarray[1]=new C; // runtime error
   }
   void runexample(){
       D[] newarray= new D[5];
       mytest(newarray);
   }
}
```

Transferring that information to the universe types result in two possibilities. Either it restricts the programmer and limits the subtyping rules in a very strict way to guarantee that the semantics of array updates does not have to be extended in the Java Virtual Machine (JVM), or (on the other side) make the subtyping rules for the universe modifier as simple as possible, allowing a covariant subtyping for the whole type, even for the universe modifiers.
To find out how strict the typing rules would have to be limited, let's look at the different cases. Those cases either require a covariant or a contravariant subtyping of the type part of arrays:

- An array should be assigned to a field ($o.f = ArrayX$);

it should be possible to assign arrays to more ”general” types.
$\Longrightarrow$ Covariant subtyping behavior for array elements

- A method call $o.m(ArrayX)$;
  the same as before should apply to that
  $\Longrightarrow$ Covariant subtyping behavior for array elements.

- An array update should not destroy the original type of the array;
  in java this is guaranteed for the subclassing during runtime.
  $\Longrightarrow$ Would require contra-variant subtyping behavior for array elements.

There are the concurrent goals for field updates and method calls on the one side and array updates on the other side.
Because that would limit the developer and it is not intuitive to write a program with those limitations in mind, covariant subtyping should be used. Problems with the array updates are handled in the bigstep semantics.
This knowledge in mind results in additional subtyping rules, which can be seen in figure 3.6.

$$\frac{CT \vdash\ C[: D \qquad u \leq u'}{CT \vdash\ Object_t\ u\ C <: Object_t\ u'\ D}\ \text{st-1-subtype-class}$$

$$\frac{u_a \leq u'_a \qquad CT \vdash\ T <: T'}{CT \vdash\ Array_t\ u_a\ T <: Array_t\ u'_a\ T}\ \text{st-1-subtype-array}$$

Figure 3.6: Array subtyping rules

3.2.4 Type system

With the specification of arrays and how the subtyping is done, the new typing rules can be declared (See figure 3.7).

In most parts they are the same as the previously defined typing rules, except for the following parts: the comparison that a universe modifier is not *unknown* was replaced by the function notUnknown to handle arrays.
The type constructors have been added and additional expression rules where introduced: for the case of a new array, the result will be the type of that new array. For the case of an array read, the resulting type will be the type of the elements of the array. At last for the case of an array update, the type of the elements of the array has to be a supertype to the type of the element that is assigned to that array.

3.3 Runtime model

3.3.1 Heap definition and well typedness of the environment

Because arrays have a different structure than normal objects and they have to store more typing information, the heap definition has to be extended. The new definition can be seen in figure 3.8.

There have to be type constructors for the two types object and array, and it has to be differentiated between arrays and objects in the heap store.

To be able to handle the new runtime subtypes, the definition of the dynType (Figure 3.9) function and the runtime subtyping rules have to be extended.
The function
$getRType :: Heap \times Addr \rightarrow Type^r$

returns the runtime type of an object, for arrays as well as for normal objects.

Next the well typedness of addresses could be defined. With the extension for arrays it has to be distinguished between normal objects and arrays for the well typing of an address. While well typing for addresses that map to a normal object stay the same in most parts, if an adress maps to an array it has to guarantee, that it has a valid owner (compare the case for normal objects) and all elements of the arrays have to have a more specific type then the type-definition for the elements in the array. See figure 3.10 for more details.

The definition for the wellformedness of the heap and the environment can stay the same.

3.3.2 Bigstep semantics

For the part of the bigstep semantics there have changed some things (See Figure 3.11). First of all, the rule for object creation [os-new-class] has to be limited to class-types only. In the rules for the field read and update [os-read,os-update] and the method call [os-invoke] the receiver object has to be a normal object. Besides that, the new rules to handle arrays have to be introduced:

- To read an element from the array [os-array-read] the bigstep semantics does a lookup in the array-field at the value n. It will result in an address.
- In the case of an array-update [os-array-update] it has to be guaranteed that the array-field is defined at position n, and that the runtime type of the address is a subtype of the runtime type of the array elements. If those conditions hold, the array-field could be updated at the position of n.
- The last possible expression is the creation of a new array [os-new-array]. To create a new array object the type has to be dynamized. All numbers from zero up to n have to map to the null-pointer in the array-field. All other numbers are not allowed to be in the domain of the array-field.

3.4 Proofs

The resulting lemmas will stay the same as in the type system without arrays. The only difference is the proof itself, because the new definition and semantics have to be used.

$$\frac{CT;\Gamma^s \vdash e : T' \quad CT \vdash\ T' <: T}{CT;\Gamma^s \vdash e : T}\ \text{t-subs}$$

$$\frac{}{CT;\Gamma^s \vdash null\colon T}\ \text{t-null}$$

$$\frac{}{CT;\Gamma^s \vdash v : \Gamma^s(v)}\ \text{t-var}$$

$$\frac{CT;\Gamma^s \vdash e : Object_t\ u\ C}{CT;\Gamma^s \vdash e.f : u \rhd^t\ \ fType(CT,C,f)}\ \text{t-read}$$

$$\frac{\begin{array}{cc} CT;\Gamma^s \vdash e_0 : Object_t\ u_0\ C_0 & u_0 \rhd^t\ \ fType(CT,C_0,f) = T \\ notUnknown(T) & CT;\Gamma^s \vdash e_1 : T \end{array}}{CT;\Gamma^s \vdash e_0.f = e_1 : T}\ \text{t-update}$$

$$\frac{\begin{array}{cc} CT;\Gamma^s \vdash e_0 : Object_t\ u_0\ C_0 & mType(CT,C_0,m) = (w,\ \overline{T_s},\ T) \\ u \rhd^t\ \overline{T_s} = \overline{T_{su}} & u_0 \rhd^t\ \ T = T_u \\ \forall T \in \overline{T_{su}}.notUnknown(T) & CT;\Gamma^s \vdash \overline{e_s} : \overline{T_{su}} \end{array}}{CT;\Gamma^s \vdash e_0.m(\overline{e_s}) : T_u}\ \text{t-invk}$$

$$\frac{u \in\ \{peer, rep, any_u\}}{CT;\Gamma^s \vdash new\ u\ C : Object_t\ u\ C}\ \text{t-new-class}$$

$$\frac{CT;\Gamma^s \vdash e : T'}{CT;\Gamma^s \vdash (T)e : T}\ \text{t-cast}$$

$$\frac{CT;\Gamma^s \vdash e : Array_t\ u_0\ T}{CT;\Gamma^s \vdash e[n] : T}\ \text{t-read-array}$$

$$\frac{\begin{array}{c} CT;\Gamma^s \vdash e_0 : Array_t\ u_0\ T \\ CT;\Gamma^s \vdash e_1 : T \end{array}}{CT;\Gamma^s \vdash e_0[n] = e_1 : T}\ \text{t-update-array}$$

$$\frac{u_0\ \in\ \{peer, rep, any_u\}}{CT;\Gamma^s \vdash new\ Array_t\ u_0\ T : Array_t\ u_0\ T\ n}\ \text{t-new-array}$$

Figure 3.7: Syntax rules with arrays

ι	$\in$	$Addr$	$::=$	$Integer \mid null_a$
ι_o	$\in$	$OwnerAddr$	$::=$	$Addr \mid any_a \mid anyowner_a$
T^r	$\in$	$Type^r$	$::=$	$OwnerAddr\ Class \mid$
				$Array_r\ OwnerAddr\ Type^r$
V^r	$\in$	Var^r	$::=$	$VarName \rightarrow\ Addr$
I^r	$\in$	$Index^r$	$::=$	$Index \rightarrow\ Addr$
Obj	$\in$	$Object^r$	$::=$	$Type^r\ Var^r \mid$
				$Array_o\ Type^r\ Index^r$
h	$\in$	$Heap$	$::=$	$Addr \rightarrow\ Object$
Γ^r	$\in$	Env^r	$::=$	Var^r

Figure 3.8: Heap with arrays

$$\frac{dynType(h, \iota_{me}, T) = T_r}{dynType(h, \iota_{me}, Array_t\ rep\ T_r) = Array_r\ \iota_{me}\ T_r}\ \text{dynType-array-rep}$$

$$\frac{dynType(h, \iota_{me}, T) = T_r \qquad u\ \in\ \{peer, this\}}{dynType(h, \iota_{me}, Array_t\ u\ T) = Array_r\ owner(h, \iota_{me})\ T_r}\ \text{dynType-array-peer}$$

$$\frac{dynType(h, \iota_{me}, T) = T_r \qquad \text{u=}(\ unknown|\ any_u)}{dynType(h, \iota_{me}, Array_t\ u\ T) = Array_r\ any_a\ T_r}\ \text{dynType-array-any}$$

Figure 3.9: The dynType function for arrays

$$\frac{\begin{array}{c} h(\iota) = T^r\ V^r \qquad owner(\iota) = \iota_o \qquad \iota_o \in dom(h) \vee\ \iota_o \in \{null_a, anyowner_a\} \\ isObject(Obj) \qquad\qquad aClass(h, \iota) = C \\ \forall(f\ \in\ dom(V^r).CT, h, \iota\ \vdash\ V^r(f) : fType(CT, C, f) \end{array}}{CT, h \vdash_a\ \iota}\ \text{wt-a}$$

$$\frac{\begin{array}{c} h(\iota) = Array_o\ (Array_r\ \iota_o\ T)\ I^r \qquad owner(\iota) = \iota_o \qquad \iota_o \in dom(h) \vee\ \iota_o \in \{null_a, anyowner_a\} \\ \forall(n \in dom(I^r).CT, h, \iota\ \vdash I^r(n) : T \end{array}}{CT, h \vdash_a\ \iota}\ \text{wt-a}$$

Figure 3.10: Welltyped address

$$\frac{}{CT \vdash h, \Gamma^r, \text{null} \rightsquigarrow h, null_a} \text{ os-null}$$

$$\frac{\Gamma^r(v) = \iota}{CT \vdash h, \Gamma^r, v \rightsquigarrow h, \iota} \text{ os-var}$$

$$\frac{\begin{array}{c}\iota \notin dom(h) \\ u \neq any \implies h' = h[\iota \mapsto Object_o(dynType(h, \Gamma^r(\text{this}), (Object_t u C))declareFields(C))] \\ u = any \implies h' = h[\iota \mapsto (anyowner_a\ C)declareFields(C))]\end{array}}{CT \vdash h, \Gamma^r, new\ u\ C \rightsquigarrow h', addr} \text{ os-new-class}$$

$$\frac{CT \vdash h, \Gamma^r, e \rightsquigarrow h', \iota \qquad CT, h, \Gamma^r(\text{this}) \vdash\ \iota : T}{CT \vdash h, \Gamma^r, (T)e \rightsquigarrow h', \iota} \text{ os-cast}$$

$$\frac{CT \vdash h, \Gamma^r, e \rightsquigarrow h', \iota_e \qquad h'(\iota_e) = Object_o(T^r\ V^r) \qquad V^r(f) = \iota}{CT \vdash h, \Gamma^r, e.f \rightsquigarrow h', \iota} \text{ os-read}$$

$$\frac{\begin{array}{c}CT \vdash h, \Gamma^r, e_0 \rightsquigarrow h', \iota_0 \qquad CT \vdash h', \Gamma^r, e_1 \rightsquigarrow h'', \iota_1 \\ h''(\iota_0) = Object_o(T^r\ V^r) \\ h''' = h''[\iota_0 \mapsto Object_o(T^r\ V^r[f \mapsto \iota_1])]\end{array}}{CT \vdash h, \Gamma^r, e_0.f = e_1 \rightsquigarrow h''', \iota_1} \text{ os-update}$$

$$\frac{\begin{array}{c}CT \vdash h, \Gamma^r, e_0 \rightsquigarrow h', \iota_0 \qquad CT \vdash h', \Gamma^r, \overline{e_s} \rightsquigarrow h'', \overline{\iota_s} \\ h''(\iota_0 = Object_o\ (\iota_o\ C)\ V^r \\ mBody(CT, aClass(h, \iota_0), m) = (\overline{v_s\ T_s}, e_1) \\ \Gamma^r_{new} = [\overline{v_s}[\mapsto]\overline{\iota_s}][\text{this}\mapsto \iota_p] \\ CT \vdash h'', \Gamma^r_{new}, e_1 \rightsquigarrow h''', \iota_1\end{array}}{CT \vdash h, \Gamma^r, e_0.m(\overline{e_s}) \rightsquigarrow h''', \iota_1} \text{ os-invoke}$$

$$\frac{\begin{array}{c}CT \vdash h, \Gamma^r, e \rightsquigarrow h', \iota_e \\ Array_o\ T\ rI = h'(\iota_e) \qquad rI(n) = \iota\end{array}}{CT \vdash h, \Gamma^r, e[n] \rightsquigarrow h', \iota} \text{ os-read-array}$$

$$\frac{\begin{array}{c}CT \vdash h, \Gamma^r, e_0 \rightsquigarrow h', \iota_0 \qquad CT \vdash h', \Gamma^r, e_1 \rightsquigarrow h'', \iota_1 \\ h''(\iota_0) = Array_o\ (Array_r \iota_o T_r)rI \qquad n \in dom(rI) \\ CT \vdash getRType(h'', \iota_1) <: T_r \\ h''' = h''[\iota_0 \mapsto (Array_o T_r rI[n \mapsto \iota_1])]\end{array}}{CT \vdash h, \Gamma^r, e0[n] = e1 \rightsquigarrow h''', \iota_1} \text{ os-update-array}$$

$$\frac{\begin{array}{c}\iota \notin dom(h) \\ \forall i < n.rI(i) = NullP \qquad \forall i \geq n.i \notin dom(rI) \\ u \neq any \implies h' = h[\iota \mapsto Array_o(dynType(Array_t\ u\ T)\ rI)] \\ u = any \implies h' = h[\iota \mapsto Array_o\ (Array_r\ anyowner_a\ dynType(T)\ rI)]\end{array}}{CT \vdash h, \Gamma^r, new\ Array_t\ u\ T\ n \rightsquigarrow h', \iota} \text{ os-new-array}$$

Figure 3.11: Bigstep semantics with arrays

4 Isabelle Formalization

The following chapter contains the Isabelle/HOL formalization. Section 5.2 is generated using the isabelle tools.

4.1 Overview over theory files

The formalization of the typesystem is split up into different theory files:

syntaxdef.thy The abstract syntax tree is specified, derived from the Featherweight Java formalization.

vpadaptation.thy The viewpoint adaptation is defined.

typingrelation.thy The typingrelation consists of subclassing, relation of the universe modifier and subtyping.

statichelper.thy Some auxiliary functions for the static part are declared.

exptyping Typing rules for expressions are defined as well as welltypedness of the classtable.

static.thy Different lemmas about the static parts of the typesystem are shown.

heap.thy The heap is modeled. The theory is adapted from the Jinja heap.

runtimehelper.thy Auxiliary functions for the runtime part have to be defined.

weltyped.thy The wellformedness of the runtime environment is defined in a theory.

bigstep.thy An inductive definition of the bigstep semantics has to be written down.

topologicalproofs.thy The proofs about the adapatation of the viewpoint have to be done.

typesafety.thy The type system will be shown type safe.

enctyping.thy For the encapsulation type system some encapsulation rules are specified.

ownerasmodifier.thy The owner-as-modifier property is shown.

The symbols and variable names in the formalization of Isabelle will be in most cases the same as in the description, only where another notation is required for the proofs they have been changed.

4.2 Formalization

theory *syntaxdef* **imports** *Main*

begin

4.2.1 Syntax definition

In the very first part of the formalization, the abstract syntax has to be defined using Isabelle types and definitions.
As there already exists a formalization of Featherweight Java, it follows that definition in many cases.

To represent the names of variables, fields and methods, different types are used. It is simplified by using natural numbers to represent the names. If one wants to use names instead of natural numbers, he can introduce a lookup-table mapping names to natural numbers and vice versa.

The

types *varName* = *nat*

represent the names for variables and fields.

For methods

types *methodName* = *nat*

will be used.

At last for classes

types *className* = *nat*

will be used.

The next element is the universe modifier, being a datatype with the different modifiers as possibilities:

datatype *OM* =
Peer
|*Rep*
|*Any*
|*This*
|*Unknown*

The

datatype *Purity* = *Pure* | *NonPure*

is required to define functions that are not allowed to modify the heap. They can be called at every point in the program, and should not break the owner-as-modifier-property. It will be proven later on.

As normal objects as well as arrays are supported by the typesystem, an inductive type is declared, being either an object or an array:

datatype *uType* = *sObject OM className*
| *sArray OM uType*

To make the rules and lemmas more readable, there are additional access functions specified.

This will not only be done for the uType but, also for other Isabelle types.

consts *utOM* :: *uType* ⇒ *OM*
primrec
utOM (*sObject u C*) = *u*
utOM (*sArray u T*) = *u*

consts *utClass* :: *uType* ⇒ *className*

primrec
utClass (*sObject u C*) = *C*

lemma [*iff*]:*utOM* (*sObject u C*)=*u* **by** (*simp*)
lemma [*iff*]:*utClass* (*sObject u C*)=*C* **by** (*simp*)

The classes in the system have fields. To identify those fields, a name together with the type of the field has to be stored in the classtable. This is done using a list of

types *varDef* = *varName* × *uType*

pairs in the class definition.

At last there exist functions to access the parts of the pairs, because using fst and snd could be sometimes quite confusing:

constdefs *vdName* :: *varDef* ⇒ *varName*
vdName T ≡ *fst T*
constdefs *vdType* :: *varDef* ⇒ *uType*
vdType T ≡ *snd T*

When possible a simplification can be done for those elements using the following lemmas:

lemma [*iff*]:*vdName* (*a*, *b*)=*a* **by** (*simp add*:*vdName-def*)
lemma [*iff*]:*vdType* (*a*, *b*)=*b* **by** (*simp add*:*vdType-def*)

As the expressions allow to read variables from an environment, there has to be an appropriate static definition

types *staticEnv* = *varName* ⇀ *uType*

for the environment.

Constants

In the static environment there exist two constants:

consts
Object :: *className*
this :: *varName*

Object is the top level class in the class hierarchy. It has no methods and no fields. Every other class and even the arrays are derived from that class. Every type in the system is a subclass of Object.
As there exists a current viewpoint in the typesystem representing the object the program execution currently is in, the this variable should point to that object. In the static environment it represents the type of the current viewpoint.

Expressions

As already stated in the beginning, the syntax uses a very small subset of Java, with only a few more expressions than Featherweight Java. The expression defined here, will be used later on to define the semantics on top of it. Every methods expression will be a composition of all those single elements.

datatype *exp* =
Null
| *Var varName*
| *FieldProj exp varName*
| *FieldUpdate exp varName exp*
| *MethodInvk exp methodName exp list*

| *New OM className*
| *Cast uType exp*
| *NewArray OM uType nat*
| *ArrayProj exp nat*
| *ArrayUpdate exp nat exp*

One can see that most of the elements have a subexpression as part of their expression. Only a few of them are terminating expressions.

Methods,Classes and the ClassTable

To define a method there is a whole bunch of information required. To arrange those elements a record definition is used:

record *methodDef* =
mPure :: *Purity*
mReturn :: *uType*
mName :: *methodName*
mParams :: *varDef list*
mBody :: *exp*

Purity is required only for the encapsulation typesystem. As there should not be kept two separate method definition tables, one of them only handling the purity of methods, it is already specified in the beginning. All other parts should not need any further description.

Because classes have fields and methods, they have to define lists of those elements:

record *classDef* =
cName :: *className*
cSuper :: *className*
cFields :: *varDef list*
cMethods :: *methodDef list*

An alternative would be using a partially declared function mapping names to the corresponding records, but it seemed to be more simple to handle those lists.

Last but not least, the classtable is of importance. It is a mapping of classnames to their specifications.

types *classTable* = *className* $\rightharpoonup$ *classDef*

If compared with a Java program, that abstract definition stands for the parsed, but not verified source code. Parsed because the definition fulfills some formal limitations, for example there is no expression called "Throw" defined.
It is not verified, because the expressions do not have to fulfill some required limitations. The rules would still allow to specify a class Example which extends Example, what should definitely be forbidden.

end

theory *vpadaption* **imports** *syntaxdef*

begin

4.2.2 Viewpoint adaptation

The viewpoint adaptation is used to adapt the viewpoint of a type to another viewpoint. It is a functional representation for the prior defined lookup table.

constdefs *adaptoo* :: $OM \Rightarrow OM \Rightarrow OM$ (- $\triangleright$ - [51,51] 50)
$x \triangleright y \equiv$ *case x of This* $\Rightarrow y$
| *Peer* $\Rightarrow$ (*case y of Peer* $\Rightarrow$ *Peer* | *Any* $\Rightarrow$ *Any* |- $\Rightarrow$ *Unknown*)
| *Rep* $\Rightarrow$ (*case y of Peer* $\Rightarrow$ *Rep* | *Any* $\Rightarrow$ *Any* |- $\Rightarrow$ *Unknown*)
| - $\Rightarrow$ (*case y of Any* $\Rightarrow$ *Any* |- $\Rightarrow$ *Unknown*)

As in most of the cases there will be an adaptation of an universe modifier with a type, there is a separate infix function for this job:

consts *adaptot* :: $OM \Rightarrow uType \Rightarrow uType$ (- $\triangleright^t$ - [51,51] 50)
primrec
$(x::OM) \triangleright^t$ *sObject u C* = *sObject* $(x \triangleright u)$ *C*
$(x::OM) \triangleright^t$ *sArray u T* = *sArray* $(x \triangleright u)$ $(x \triangleright^t T)$

There are some special cases, where the adaptation step can be done, although not all of the parameter information exist:
One of those cases is the Any modifier on the right hand of the adaptation function.

lemma *adaptoo-x-ro*[*iff*]: $x \triangleright Any = Any$
lemma *adaptot-x-ro*[*iff*]: $x \triangleright^t$ *sObject Any y* = *sObject Any y*

To help the simplifier work with the adaptation function, and still do not have to unfold the definition all the time, there exist some more simplification lemmas:

lemma [*iff*]:*This* $\triangleright$ *y* = *y* **by** (*simp add*:*adaptoo-def*)
lemma [*iff*]:*Peer* $\triangleright$ *Peer* = *Peer* **by** (*simp add*:*adaptoo-def*)
lemma [*iff*]: *x* $\triangleright$ *Unknown* = *Unknown***lemma** [*iff*]:*Peer* $\triangleright$ *This* = *Unknown* **by** (*simp add*:*adaptoo-def*)
lemma [*iff*]:*Peer* $\triangleright$ *Rep* = *Unknown* **by** (*simp add*:*adaptoo-def*)
lemma [*iff*]:*Rep* $\triangleright$ *Peer* = *Rep* **by** (*simp add*:*adaptoo-def*)
lemma [*iff*]:*Rep* $\triangleright$ *This* = *Unknown* **by** (*simp add*:*adaptoo-def*)
lemma [*iff*]:*Rep* $\triangleright$ *Rep* = *Unknown* **by** (*simp add*:*adaptoo-def*)

At last the notUnknown function is specified, testing an Universe Type for any evidence of an Unknown modifier. It is required for the typing rules.

consts *notUnknown*:: *uType* $\Rightarrow$ *bool*
primrec
notUnknown (*sObject u C*) = (*case u of Unknown* $\Rightarrow$ *False* | - $\Rightarrow$ *True*)
notUnknown (*sArray u T*) = ($u \neq Unknown \wedge$ *notUnknown*(*T*))

end

theory *typingrelation* **imports** *syntaxdef*

begin

4.2.3 Typing relation

The internal definition of reflexivity and transitivity in Isabelle is used to describe the reflexive and transitive closure of subclassing and the order of the universe modifier. Because the typing depends on the classtable CT, the set is defined with the classtable as parameter.

Subclassing

consts *subclassing* :: *classTable* $\Rightarrow$ (*className* $*$ *className*) *set*
subclassings :: *classTable* $\Rightarrow$ (*className list* $*$ *className list*) *set*

There is a subclassing relation defined for the direct subclassing of two classes. Besides this, there is a subclassing relation extended with the reflexive and transitive closure.

syntax
-subclassing :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - [:$_s$ - [*80*,*80*,*80*] *80*)
-subclassingclos :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - [: - [*80*,*80*,*80*] *80*)
-subclassings :: [*classTable*, *className list*, *className list*] ⇒ *bool* (- ⊢$^+$ - [: - [*80*,*80*,*80*] *80*)
-neg-subclassing :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - ¬[: - [*80*,*80*,*80*] *80*)
translations
CT ⊢ *C* [:$_s$ *D* ⇌ (*C*, *D*) ∈ (*subclassing CT*)
CT ⊢ *C* [: *D* ⇌ (*C*, *D*) ∈ (*subclassing CT*)*
CT ⊢$^+$ *Cs* [: *Ds* ⇌ (*Cs*, *Ds*) ∈ (*subclassings CT*)
CT ⊢ *C* ¬[: *D* ⇌ (*C*, *D*) ∉ (*subclassing CT*)*

Subclassing is defined inductive with only the one relation for the direct subclassing.

inductive *subclassing CT*
intros
sc-1-super : ⟦ *CT*(*C*) = *Some*(*CDef*); *cSuper CDef* = *D*; *C* ≠ *Object*⟧ ⟹ *CT* ⊢ *C* [:$_s$ *D*

In some cases lists of classes have to be handled. There exists a subclassing relation for lists.

inductive *subclassings CT*
intros
— The empty list fits to an emtpy list
ss-nil : *CT* ⊢$^+$ [] [: []
— For the case of a concatenation of an element and a list, a clear induction rule is defined:
ss-cons : ⟦ *CT* ⊢ *C0* [: *D0*; *CT* ⊢$^+$ *Cs* [: *Ds* ⟧ ⟹ *CT* ⊢$^+$ (*C0*#*Cs*) [: (*D0*#*Ds*)

Order of the ownership modifier

Early in the text there was an order of the ownership modifier defined:

consts *omorder* :: (*OM* ∗ *OM*) *set*
syntax
-omorder :: [*OM*, *OM*] ⇒ *bool* (- <:$_s$ - [*80*,*80*] *80*)
-omorderclos :: [*OM*, *OM*] ⇒ *bool* (- <: - [*80*,*80*] *80*)
-omordern :: [*OM*, *OM*] ⇒ *bool* (- ¬<: - [*80*,*80*] *80*)
translations
u1 <:$_s$ *u2* ⇌ (*u1*, *u2*) ∈ *omorder*
u1 <: *u2* ⇌ (*u1*, *u2*) ∈ *omorder*^∗
u1 ¬<: *u2* ⇌ (*u1*, *u2*) ∉ *omorder*^∗

Only the direct dependencies have to be declared in the inductive definition, the reflexivity and transitivity will be added using the internal definition of Isabelle again:

inductive *omorder*
intros
om-order-l : *Rep* <:$_s$ *Any*
om-order-2 : *Peer* <:$_s$ *Any*
om-order-3 : *This* <:$_s$ *Peer*
om-order-4 : *Unknown* <:$_s$ *Any*

4.2.4 Subtyping relation

The subtyping relation is a combination of subclassing and the order of the ownership modifier. Besides that, there are special rules to handle arrays:

consts *subtyping* :: *classTable* ⇒ (*uType* ∗ *uType*) *set*
subtypings :: *classTable* ⇒ (*uType list* ∗ *uType list*) *set*

syntax
-subtyping :: [*classTable*, *uType*, *uType*] ⇒ *bool* (- ⊢ - <: - [80,80,80] 80)
-subtypingsingle :: [*classTable*, *uType*, *uType*] ⇒ *bool* (- ⊢ - <:$_s$ - [80,80,80] 80)
-subtypings :: [*classTable*, *uType list*, *uType list*] ⇒ *bool* (- ⊢$^+$ - <: - [80,80,80] 80)
-neg-subtyping :: [*classTable*, *uType*, *uType*] ⇒ *bool* (- ⊢ - ¬<: - [80,80,80] 80)
translations
CT ⊢ *S* <:$_s$ *T* ⇌ (*S*, *T*) ∈ (*subtyping CT*)
CT ⊢ *S* <: *T* ⇌ (*S*, *T*) ∈ (*subtyping CT*)*
CT ⊢$^+$ *Ss* <: *Ts* ⇌ (*Ss*, *Ts*) ∈ (*subtypings CT*)
CT ⊢ *S* ¬<: *T* ⇌ (*S*, *T*) ∉ (*subtyping CT*)
inductive *subtyping CT*
intros
— Subtyping of objects is defined straight forward by the classpart and the ownership part:
st-1-subtype : ⟦*CT* ⊢ *C* [: *D*; *u* <: *ux*⟧ ⟹ *CT* ⊢ (*sObject u C*) <:$_s$ (*sObject ux D*)
— For the case of arrays there exists a special rule handling the subtyping of the type part for one step:
st-2-arraytype : ⟦*CT* ⊢ *S* <:$_s$ *T*⟧ ⟹ *CT* ⊢ (*sArray u S*) <:$_s$ (*sArray u T*)
— The same is done for the universe part:
st-3-arrayom : ⟦*u* <:$_s$ *ux*⟧ ⟹ *CT* ⊢ (*sArray u S*) <:$_s$ (*sArray ux S*)
— At last per definition every array type is a subtype of the Object class:
st-4-arrayObj : *CT* ⊢ *sArray u S* <:$_s$ *sObject u Object*
inductive *subtypings CT*
intros
ss-nil : *CT* ⊢$^+$ [] <: []
ss-cons : ⟦ *CT* ⊢ *S0* <: *T0*; *CT* ⊢$^+$ *Ss* <: *Ts*⟧ ⟹ *CT* ⊢$^+$ (*S0*#*Ss*) <: (*T0*#*Ts*)

Proofs of subtyping of types

Object is the weakest subclass. This can be shown:

lemma [*iff*]: ¬ *CT* ⊢ *Object* [:$_s$ *C*
by *fastsimp*

The same rule with other words says that if object is on the left side of the subclassing relation, on the right side there has to be Object, too:

lemma [*iff*]: (*CT* ⊢ *Object* [: *C*) = (*C* = *Object*)

Sometimes the subclassing definition should be unfolded. This is done by the following lemma:

lemma *sc-open*:*CT* ⊢ *C*[:$_s$ *D* ⟹ ∃ *CDef*. *CT*(*C*) = *Some*(*CDef*) ∧ *cSuper CDef* = *D* ∧ *C* ≠ *Object*

As there exist only a few universe modifiers and the right handed side of the subtyping relation for the universe modifiers are known, the variable on the left handed side can be limited to one or two possibilities:

lemma *this-def*[*iff*]:(*x* <: *This*) = (*x*=*This*)
lemma *peer-def*[*iff*]:(*x* <: *Peer*) = (*x*=*This* ∨ *x*=*Peer*)
lemma *unknown-def*[*iff*]:(*x* <: *Unknown*) = (*x*=*Unknown*)
lemma *rep-def*[*iff*]:(*x* <: *Rep*) = (*x*=*Rep*)
end

theory *statichelper* **imports** *syntaxdef*

begin

4.2.5 Static helper functions

The lookup function

The lookup function is used to handle lists of elements. There already exists a lemma in Isabelle to do a lookup on pairs of elements. The newly defined lookup function is required to handle the records defined in the syntax.

consts *lookup* :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *'a option*

primrec
lookup [] *P* = *None*
lookup (*h*#*t*) *P* = (*if P h then Some h else lookup t P*)

Variable definition accessors

The following section contains two helper functions for reading of the names and types of variable definitions (e.g., in field and method parameter declarations). They split varDefs into the two parts names and types.

Both constdefs are just nicer names for the map functions to be used in the semantics to create more readable code:

constdefs *varDefs-names* :: *varDef list* ⇒ *varName list*
varDefs-names ≡ *map vdName*

constdefs *varDefs-types* :: *varDef list* ⇒ *uType list*
varDefs-types ≡ *map vdType*

Field relation

The field relation describes the relation and the type of fields.

The fields function is defined as an inductive set, and "returns" all fields defined in a class:

consts *fields* :: (*classTable* ∗ *className* ∗ *varDef list*) *set*
syntax
-fields :: [*classTable*, *className*, *varDef list*] ⇒ *bool* (*fields'*(-,-') = - [*80*,*80*,*80*] *80*)
translations
fields(*CT*, *C*) = *Cf* ⇌ (*CT*, *C*, *Cf*) ∈ *fields*
inductive *fields*
intros
— For the case of the Object class there exist no fields. To specify the definition inductively this case has to be declared separately:

f-obj:
fields(*CT*, *Object*) = []

— In all the other cases the fields of the superclasses are merged with the fields of the class:
f-class:
⟦ *CT*(*C*) = *Some*(*CDef*);
cSuper CDef = *D*;
cFields CDef = *Cf*;
fields(*CT*, *D*) = *Dg*;
DgCf = *Dg* @ *Cf*;
C≠*Object*⟧
⟹ *fields*(*CT*, *C*) = *DgCf*

Next, the ftype function gets the universe type of a field. To do that, it uses the fields functions. It either returns the type of the field or None, if the field is not denoted.

consts *ftype* :: (*classTable* ∗ *varName* ∗ *className* ∗ *uType option*) *set*
syntax
-ftype :: [*classTable*, *className*, *varName*, *uType option*] ⇒ *bool* (*ftype′*(-,-,-′) = - [*80*,*80*,*80*,*80*] *80*)
translations
ftype(*CT*, *C*, *f*) = *T* ⇌ (*CT*, *f*, *C*, *T*) ∈ *ftype*
inductive *ftype*
intros
ft-definition:
⟦ *fields*(*CT*, *C*)=*myfields*;
map-of myfields f=*Some*(*T*)⟧
⟹ *ftype*(*CT*, *C*, *f*) = *Some*(*T*)

ft-definition-none:
⟦ *fields*(*CT*, *C*)=*myfields*;
map-of myfields f=*None*⟧
⟹ *ftype*(*CT*, *C*, *f*) = *None*

Method functions

In contrast to the informal definition, there is only one mdata function used instead of a mbody and a mtype function.

consts *mdata* :: (*classTable* ∗ *methodName* ∗ *className* ∗ *methodDef option*) *set*
syntax
-mdata :: [*classTable*, *className*, *methodName*, *methodDef option*] ⇒ *bool* (*mdata′*(-, -, -′) = - [*80*,*80*,*80*,*80*] *80*)
translations
mdata(*CT*, *C*, *m*) = *mDef* ⇌ (*CT*, *m*, *C*, *mDef*) ∈ *mdata*

inductive *mdata*
intros
— In the normal case it will return the methods record declared in the class
mt-class:
⟦ *CT*(*C*) = *Some*(*CDef*);
C≠*Object*;
lookup (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *Some*(*mDefo*)⟧
⟹ *mdata*(*CT*, *C*, *m*) = *Some*(*mDefo*)
— Only if it is not defined there, it will return the mdata result of the superclass
mt-super:
⟦ *CT*(*C*) = *Some* (*CDef*);
C≠*Object*;
lookup (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *None*;
cSuper CDef = *D*;
mdata(*CT*, *D*, *m*) = *mDef* ⟧
⟹ *mdata*(*CT*, *C*, *m*) = *mDef*
— At last if no other class can be inspected, the rule for the Object class has to be used. As Object does not declare any methods it will return None as result.

mt-object:
mdata(*CT*, *Object*, *m*) = *None*

Lemmas adopted from the Featherweight Java formalization

To be able to use the lookup function in the later cases, some properties have to be shown about the behavior of the lookup function.

Lookup

At the beginning the lookup function was introduced. As one can easily see, the lookup function is functional:

lemma *lookup-functional*:
  assumes *lookup l f = o1*
  and *lookup l f = o2*
  shows *o1 = o2*
using *prems* **by**(*induct l*, *auto*)

Another helpful information we know from the lookup function is, that for the resulting element the predicate holds.

lemma *lookup-true*:
  lookup l P = Some r ⟹ P r

In contrast to the previous lemma, from the information that lookup will not return a result we know, that all elements of the array a do not match to the predicate P.

lemma *lookup-none:lookup a P = None ⟹ ∀ i < length a. (¬ P (a!i))*
theory *exptyping* **imports** *vpadaption typingrelation statichelper*

begin

4.2.6 Expression typing

The following rules describe the typesystem and the typing rules. They limit the expression to fit to the topological typesystem.

Static definition

The typing and typings relation specify the typing rules for the expressions:

consts
  typing :: (*classTable* ∗ *staticEnv* ∗ *exp* ∗ *uType*) *set*
  typings :: (*classTable* ∗ *staticEnv* ∗ *exp list* ∗ *uType list*) *set*
syntax
  -typing :: [*classTable*, *staticEnv*, *exp*, *uType*] ⇒ *bool* (-, - ⊢ - : - [*80*,*80*,*80*,*80*] *80*)
  -typings :: [*classTable*, *staticEnv*, *exp list*, *uType*] ⇒ *bool* (-, - ⊢⁺ - : - [*80*,*80*,*80*,*80*] *80*)
translations
  CT, Γ ⊢ *e* : *C* ⇌ (*CT*, Γ, *e*, *C*) ∈ *typing*
  CT, Γ ⊢⁺ *es* : *Cs* ⇌ (*CT*, Γ, *es*, *Cs*) ∈ *typings*

All Isabelle typing rules are derived from the previous rules, which were defined informally. Only the rules to handle lists of expression are added to the declaration.

inductive *typings typing*
intros
ts-nil : *CT*, Γ ⊢⁺ [] : []
ts-cons :
⟦ *CT*, Γ ⊢ *e0* : *T0*; *CT*, Γ ⊢⁺ *es* : *Ts* ⟧
  ⟹ *CT*, Γ ⊢⁺ (*e0* # *es*) : (*T0* # *Ts*)

t-var :
$[\![\ \Gamma(x) = Some\ T\ ]\!] \Longrightarrow CT, \Gamma \vdash Var\ x : T$
t-null :
$CT, \Gamma \vdash Null : T$

t-read :
$[\![\ CT, \Gamma \vdash e0 : sObject\ u\ C;$
$ftype(CT, C, f) = Some(Ti);$
$u \rhd^t Ti = Ti\text{-}a]\!]$
$\Longrightarrow CT, \Gamma \vdash FieldProj\ e0\ f : Ti\text{-}a$

t-update :
$[\![\ CT, \Gamma \vdash e0 : sObject\ u\ C0;$
$ftype(CT, C0, f) = Some(Ti);$
$u \rhd^t Ti = Ti\text{-}a;$
$notUnknown(Ti\text{-}a);$
$CT, \Gamma \vdash e1 : Ti\text{-}a]\!]$
$\Longrightarrow CT, \Gamma \vdash FieldUpdate\ e0\ f\ e1 : Ti\text{-}a$

t-invk :
$[\![\ CT, \Gamma \vdash e0 : sObject\ u\ C;$
$mdata(CT, C, m)=Some(mDef);$
$w=mPure\ mDef;$
$Tsu= varDefs\text{-}types(mParams\ mDef);$
$Tu=mReturn\ mDef;$
$CT, \Gamma \vdash^+ es : Ts;$
$u \rhd^t Tu = T;$
$map\ (\lambda tsd.\ u \rhd^t tsd)\ Tsu = Ts;$
$\forall\ t \in (set\ Ts).\ notUnknown(t)]\!]$
$\Longrightarrow CT, \Gamma \vdash MethodInvk\ e0\ m\ es : T$

t-new :
$[\![\ u=Peer \vee u=Rep \vee u=Any;$
$CT(C)=Some(CDef)]\!]$
$\Longrightarrow CT, \Gamma \vdash New\ u\ C\ : sObject\ u\ C$

t-cast :
$[\![\ CT, \Gamma \vdash e : S]\!]$
$\Longrightarrow CT, \Gamma \vdash Cast\ T\ e : T$
t-subs:
$[\![CT, \Gamma \vdash e : T'; CT \vdash T' <: T\ ]\!]$
$\Longrightarrow CT, \Gamma \vdash e : T$

t-newarray:
$[\![u=Peer \vee u=Rep \vee u=Any;$
$CT(C)=Some(CDef)]\!]$
$\Longrightarrow CT, \Gamma \vdash NewArray\ u\ T\ n\ : sArray\ u\ T$
t-arrayread:
$[\![CT, \Gamma \vdash e0 : sArray\ u\ T;$
$utOM\ T \neq This]\!]$
$\Longrightarrow CT, \Gamma \vdash ArrayProj\ e0\ n : T$
t-arrayupdate :
$[\![CT, \Gamma \vdash e0 : sArray\ u\ Ti;$
$CT, \Gamma \vdash e1 : Ti]\!]$
$\Longrightarrow CT, \Gamma \vdash ArrayUpdate\ e0\ n\ e1 : Ti$

declare *typings-typing.intros*[*intro*!]

lemmas *typings-typing-induct* = *typings-typing.induct* [*split-format* (*complete*)]

Because of the subsumption rule the inductive semantics can not be applied directly in most proofs, an unfolding rule or generation lemma is required. To prove the lemma the defined limits for the unification of variables need to be raised, else the prove would fail.

ML⟪
Unify.search-bound := *40*;
Unify.trace-bound := *40*;
⟫

lemma shows
typings-unfolding-rules:

$CT,\ sEnv \vdash^{+} es : rTs \implies True$
and *typing-unfolding-rules*:
$CT,\ sEnv \vdash e : T \implies$

$(\forall\ u\ C.\ ((e = New\ u\ C) \longrightarrow (
(u{=}Peer \vee u = Rep \vee u{=}Any) \wedge
C \in dom\ CT) \wedge CT \vdash (sObject\ u\ C) <: T)) \wedge$

$(\forall\ u\ Tx\ n.\ ((e = NewArray\ u\ Tx\ n) \longrightarrow (
(u{=}Peer \vee u{=}Rep \vee u{=}Any)) \wedge (CT \vdash (sArray\ u\ Tx) <: T)
)) \wedge$

$((e = Null) \longrightarrow (True)) \wedge$

$(\forall\ x.\ e{=}Var\ x \longrightarrow (\exists\ Ta.\ (sEnv(x){=}Some(Ta) \wedge CT \vdash Ta <: T))) \wedge$

$(\forall\ e1\ f.\ e{=}FieldProj\ e1\ f \longrightarrow (\exists\ u\ C\ Ti\ Ti\text{-}a\ .(CT,sEnv \vdash e1 : (sObject\ u\ C) \wedge ftype(CT,C,f){=}Some(Ti)
\wedge\ u \rhd^{t} Ti = Ti\text{-}a \wedge CT \vdash Ti\text{-}a <: T))) \wedge$

$(\forall\ e0\ f\ e1.\ e{=}FieldUpdate\ e0\ f\ e1 \longrightarrow (\exists\ u\ C\ Ti\ Ti\text{-}a\ .(CT,sEnv \vdash e0 : (sObject\ u\ C) \wedge ftype(CT,C,f){=}Some(Ti)
\wedge\ u \rhd^{t} Ti = Ti\text{-}a \wedge notUnknown(Ti\text{-}a) \wedge CT,sEnv \vdash e1 : Ti\text{-}a \wedge CT \vdash Ti\text{-}a <: T\ ))) \wedge$

$(\forall\ e0\ m\ es.\ e{=}MethodInvk\ e0\ m\ es \longrightarrow (\exists\ u\ C\ Ts\ mDef\ Ti.(CT,sEnv \vdash e0 : (sObject\ u\ C) \wedge mdata(CT,C,m){=}Some(m
\wedge$
$CT,sEnv \vdash^{+} es : Ts \wedge$
$u \rhd^{t} (mReturn\ mDef) = Ti \wedge$
$map\ (\lambda tsd.\ u \rhd^{t} tsd)\ (varDefs\text{-}types(mParams\ mDef)) = Ts \wedge$
$(\forall\ t \in (set\ Ts).\ notUnknown\ t) \wedge CT \vdash Ti <: T))) \wedge$

$(\forall\ Ti\ e0.\ e{=}Cast\ Ti\ e0 \longrightarrow (\exists\ S.\ CT,sEnv \vdash e0 : S \wedge CT \vdash Ti <: T)) \wedge$

$(\forall\ e1\ n.\ e{=}ArrayProj\ e1\ n \longrightarrow (\exists\ u\ Ti\ .(CT,sEnv \vdash e1 : (sArray\ u\ Ti) \wedge utOM\ Ti{\neq}\ This \wedge CT \vdash Ti <:
T))) \wedge$

$(\forall\ e0\ e1\ n.\ e{=}ArrayUpdate\ e0\ n\ e1 \longrightarrow (\exists\ u\ Ti\ .(CT,sEnv \vdash e0 : (sArray\ u\ Ti) \wedge CT,sEnv \vdash e1 : Ti \wedge
CT \vdash Ti <: T)))$

Static well typing rules

Every universe type used in the classtable has the following two limitations. One of them is that the class has to be in the classtable. The second one limits the universe modifier to be either Peer, Rep or Any. This and Unknown are only internal modifiers.

consts *t-typing* :: $(classTable * uType)\ set$
t-typings :: $(classTable * uType\ list\ )\ set$
syntax
-t-typing :: $[classTable, uType] \Rightarrow bool\ (-\ {}_t\vdash\ -\ [80,80]\ 80)$
-t-typings :: $[classTable, uType\ list] \Rightarrow bool\ (-\ {}_t\vdash^+\ -\ [80,80]\ 80)$
translations
$CT\ {}_t\vdash\ T \rightleftharpoons (CT,\ T) \in t\text{-}typing$
$CT\ {}_t\vdash^+\ T \rightleftharpoons (CT,\ T) \in t\text{-}typings$

inductive *t-typing*
intros
t-type:
$[\![C \in dom(CT);$
$u{=}Any \vee u{=}Peer \vee u{=}Rep]\!]$
$\Longrightarrow CT\ {}_t\vdash\ sObject\ u\ C$
t-type:
$[\![CT\ {}_t\vdash\ T;$
$u{=}Any \vee u{=}Peer \vee u{=}Rep]\!]$
$\Longrightarrow CT\ {}_t\vdash\ sArray\ u\ T$

inductive *t-typings*
intros
ts-nil :
$CT\ {}_t\vdash^+\ []$

ts-cons :
$[\![\ CT\ {}_t\vdash\ T;$
$CT\ {}_t\vdash^+\ Ts]\!]$
$\Longrightarrow CT\ {}_t\vdash^+\ (T\ \#\ Ts)$

Every class is defined by its fields and methods. Rules to describe the wellformedness of those parts, would make the work more simple.
For fields the limitations would be that a supertype of a class has to have the same type for a field or the field should no be allowed to be defined in the superclass.

consts *f-typing* :: $(classTable * className * varDef)\ set$
f-typings :: $(classTable * className * varDef\ list\ )\ set$
syntax
-f-typing :: $[classTable, className, varDef] \Rightarrow bool\ (-,\ -\ {}_f\vdash\ -\ [80,80,80]\ 80)$
-f-typings :: $[classTable, className, varDef\ list] \Rightarrow bool\ (-,\ -\ {}_f\vdash^+\ -\ [80,80,80]\ 80)$
translations
$CT,\ C\ {}_f\vdash\ v \rightleftharpoons (CT,\ C,\ v) \in f\text{-}typing$
$CT,\ C\ {}_f\vdash^+\ vs \rightleftharpoons (CT,\ C,\ vs) \in f\text{-}typings$

inductive *f-typing*
intros
f-type:
$[\![CT\ {}_t\vdash\ (vdType\ vs);$
$CT(C){=}Some(CDef);$
$ftype(CT,\ cSuper\ CDef,\ vdName\ vs){=}supert;$
$supert{=}Some(vdType\ vs) \vee supert{=}None\ ]\!]$
$\Longrightarrow CT,\ C\ {}_f\vdash\ vs$

inductive *f-typings*
intros
ts-nil :
$CT,\ C\ {}_f\vdash^+ []$

ts-cons :
$[\![\ CT,\ C\ {}_f\vdash v;$
$CT,\ C\ {}_f\vdash^+ vs]\!]$
$\Longrightarrow CT,\ C\ {}_f\vdash^+ (v\ \#\ vs)$

In contrast to fields, methods have a much more complicated welltyping definition, as it has to express the covariance of the return value and the contravariance of parameters.
Even the expression in the method has to follow some formal limitations to match the return value.

consts *method-typing* :: $(classTable * methodDef * className)\ set$
method-typings :: $(classTable * methodDef\ list * className)\ set$
syntax
-method-typing :: $[classTable,\ className,\ methodDef] \Rightarrow bool$ $(-,\ -\ {}_m\vdash\ -\ [80,80,80]\ 80)$
-method-typings :: $[classTable,\ className,\ methodDef\ list] \Rightarrow bool$ $(-,\ -\ {}_m\vdash^+\ -\ [80,80,80]\ 80)$
translations
$CT,\ C\ {}_m\vdash\ md \rightleftharpoons (CT,\ md,\ C) \in method\text{-}typing$
$CT,\ C\ {}_m\vdash^+\ mds \rightleftharpoons (CT,\ mds,\ C) \in method\text{-}typings$

inductive *method-typing*
intros
m-typing:
$[\![\ CT(C) = Some(CDef);$
$cName\ CDef = C;$
$cSuper\ CDef = D;$
$mName\ mDef = m;$
$lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = Some(mDef);$
$mBody\ mDef = e;$
$mPure\ mDef = w;$
$mParams\ mDef = vDefs;$
$mReturn\ mDef = T;$
$varDefs\text{-}types\ vDefs = vDefsT;$
$varDefs\text{-}names\ vDefs = vDefsN; distinct\ (this \# vDefsN);$
$CT\ {}_t\vdash^+ vDefsT;$
$CT\ {}_t\vdash\ T;$
$\forall\ mDefD.\ (mdata(CT,\ D,\ m) = Some(mDefD) \longrightarrow$
$(\ ((mPure\ mDefD) = w \vee w = Pure) \wedge$
$((varDefs\text{-}names\ (mParams\ mDefD)) = vDefsN) \wedge$
$(CT \vdash^+ (varDefs\text{-}types\ (mParams\ mDefD)) <: vDefsT) \wedge$
$CT \vdash T <: (mReturn\ mDefD)));$
$\Gamma = [vDefsN\ [\mapsto]\ vDefsT](this \mapsto (sObject\ This\ C));$
$CT, \Gamma \vdash e : T]\!]$
$\Longrightarrow CT,\ C\ {}_m\vdash mDef$

inductive *method-typings*
intros
ms-nil :
$CT,\ C\ {}_m\vdash^+ []$

ms-cons :
$[\![\ CT,\ C\ {}_m\vdash m;$
$CT,\ C\ {}_m\vdash^+ ms]\!]$

$\Longrightarrow CT,\ C\ {}_m\vdash^+ (m\ \#\ ms)$

Class Typing Relation

At last with all those parts the welltyping of a class and the classtable can be defined.

consts *class-typing* :: $(classTable * classDef)\ set$
syntax
-class-typing :: $[classTable,\ classDef] \Rightarrow bool$ $(-\ {}_c\vdash\ -\ [80,80]\ 80)$
translations
$CT\ {}_c\vdash\ CDef \rightleftharpoons (CT,\ CDef) \in class\text{-}typing$

In all definitions natural numbers are used to represent the name of the class. To simplify recursive definitions, it is specified that a subclass has to have a higher number than its superclass. That will allow to use the order of the natural numbers to aid in defining an relation between classes.
In a class the fields and methods have to follow the limitations named before.

inductive *class-typing*
intros
t-class: $[\![\ cName\ cDef\ =\ C;\ C\neq\ Object;$
$cSuper\ cDef\ =\ D;$
$D<C;$
$cMethods\ cDef\ =mDefs;$
$fields(CT,\ C)=fDefs;$
$CT(D)=\ Some(DDef);$
$\forall\ m\ mDefC.\ (mdata(CT,\ C,\ m)=Some(mDefC)\longrightarrow\ CT,C\ {}_m\vdash\ mDefC);$
$\forall\ f\ \in (set\ fDefs).\ CT,C\ {}_f\vdash\ f]\!]$
$\Longrightarrow CT\ {}_c\vdash\ cDef$
t-object: $[\![cMethods\ cDef=[];$
$cFields\ cDef=[];$
$cSuper\ cDef=Object;$
$cName\ cDef=Object]\!]$
$\Longrightarrow CT\ {}_c\vdash\ cDef$

In the classtable every class definition has to follow those limitation rules.

consts *ct-typing* :: *classTable set*
syntax
-ct-typing :: *classTable* $\Rightarrow$ *bool* $(\vdash\ -\ 80)$
translations
$\vdash\ CT \rightleftharpoons\ CT \in ct\text{-}typing$
inductive *ct-typing*
intros
ct-all-ok:
$[\![\ \forall\ C\ CDef.\ CT(C)\ =\ Some(CDef)\ \longrightarrow\ (CT\ {}_c\vdash\ CDef\ \wedge\ cName\ CDef=C)]\!]$
$\Longrightarrow\ \vdash\ CT$

end

theory *static* **imports** *exptyping*

begin

4.2.7 Static lemmas

The fields function will be used later on, and it has to be shown that it is functional.

lemma *fields-functional*:$\bigwedge x\ y.[\![\vdash CT;\ fields(CT,\ C){=}x;\ fields(CT,\ C){=}y]\!] \Longrightarrow x{=}y$

Because the fields function is functional, and the ftype function directly relies on that function, it has to be functional, too:

lemma *ftype-functional*:$[\![\vdash CT;\ ftype(CT,\ C,\ f){=}T;\ ftype(CT,\ C,\ f){=}S]\!] \Longrightarrow T{=}S$

The same that applies to the fields, can also be applied to method definitions. They are functional, too.

lemma *mdata-functional*:$[\![\vdash CT;\ mdata(CT,\ C,\ m){=}Some(x);\ mdata(CT,\ C,\ m){=}Some(y)]\!] \Longrightarrow x{=}y$

Besides unfolding expressions, the method's definition also has to be unfolded for the case of method calls. The following lemma will simplify that step.

lemma *mdata-unfold*:$[\![\ \vdash CT;\ mdata(CT,\ C,\ m){=}Some\ mDef]\!]$
$\Longrightarrow CT,\ [varDefs\text{-}names\ (mParams\ mDef)\ [\mapsto]\ varDefs\text{-}types\ (mParams\ mDef),\ this \mapsto (sObject\ This\ C)]$
$\vdash mBody\ mDef\ :\ mReturn\ mDef$
$\wedge\ distinct(this\#varDefs\text{-}names(mParams\ mDef))$

Methods will get executed during the bigstep semantics. To do the executing in the sense of the typing rules, the following lemma describes the typing rule generated from a method.

lemma *mdata-exec*:$[\![\ \vdash CT;\ mdata(CT,\ C,\ m){=}Some\ mDef]\!]$
$\Longrightarrow CT,\ [varDefs\text{-}names\ (mParams\ mDef)\ [\mapsto]\ varDefs\text{-}types\ (mParams\ mDef),\ this \mapsto (sObject\ This\ C)]$
$\vdash mBody\ mDef\ :\ mReturn\ mDef$

As types of parameters in methods are contravariant, the resulting limitations have to be integrated into the proofs.

lemma *mdata-subclass*:$[\![\ CT \vdash C\ [:\ D;\ \vdash CT;\ mdata(CT,\ C,\ m){=}Some\ mDef;\ mdata(CT,\ D,\ m){=}Some\ mDefa]\!]$
$\Longrightarrow\ CT \vdash mReturn\ mDef\ <:\ mReturn\ mDefa$
$\wedge\ varDefs\text{-}names\ (mParams\ mDefa){=}varDefs\text{-}names\ (mParams\ mDef)$
$\wedge\ CT \vdash^{+} varDefs\text{-}types(mParams\ mDefa)\ <:\ varDefs\text{-}types(mParams\ mDef)$

Those have been some rules to handle method data.
At last the handling of lists of expressions should be simplified. Because in the typing rules there is no state that changes during the rule application, the order of evaluation of the expressions does not matter. With that information, the typing rules of expression lists can be written down in a more simple way.
That was done to keep the typing rules in line with the other inductive definitions, and to possibly be able to integrate some state into the typing rules later on.

lemma *typing-SomeD*:$\bigwedge elist.\ CT,\ sEnv \vdash^{+} elist\ :\ Ts \Longrightarrow \forall\ e \in set\ elist.\ \exists\ T.\ (CT,\ sEnv \vdash e\ :\ T)$
end

theory *heap* **imports** *syntaxdef statichelper vpadaption* **begin**

4.2.8 Heap model

To reason about a program with heap updates, a heap has to be defined. The formalization of the heap is derived from the formalization of Jinja. As the Universe Type System integrates an owner, the heap model has to extend the type definition.

types *Addr* = *nat*
types *OwnerAddr* = *Addr*

There exist three special addresses.

consts

NullP :: *Addr*
AnyAd :: *Addr*
AnyOwner ::*Addr*

The nullpointer NullP, which maps nowhere, is used to define an empty address. The anyaddress AnyAd is used in the typing rules with the transformation from the any universe type to an address. Besides those two there exists an AnyOwner, the owner for objects that are created with the Any modifier. The distinction between AnyAd and AnyOwner is required to allow the subtyping rule for runtime types to be as expressive as the subtyping rule for static types.

Elements in the heap can either be normal objects or arrays. Like the representation in the static types, there has to be a differentiation in the runtime types:

datatype *rType* = *rObject OwnerAddr className*
| *rArray OwnerAddr rType*

To reach the parts of the types more easy in the later rules, the functions rtOwner, rtClass and rtType are defined.

consts *rtOwner* :: *rType* ⇒ *OwnerAddr*
primrec
rtOwner (*rObject addr C*) = *addr*
rtOwner (*rArray addr T*) = *addr*
consts *rtClass* :: *rType* ⇒ *className*
primrec
rtClass (*rObject addr C*) = *C*
consts *rtType* :: *rType* ⇒ *rType*
primrec
rtType (*rArray addr T*)=*T*

Again, the simplifier should know how to handle those rules:

lemma [*iff*]:*rtOwner* (*rObject addr C*)=*addr* **by** (*simp*)
lemma [*iff*]:*rtClass* (*rObject addr C*)=*C* **by** (*simp*)
lemma [*iff*]:*rtOwner* (*rArray addr T*)=*addr* **by** (*simp*)
lemma [*iff*]:*rtType* (*rArray addr T*)=*T* **by** (*simp*)

Object definition in the heap

The definition of a heap object is a main goal of the heap formalization. It is separated into the two cases normal objects and arrays.

types
fields = *varName* ⇀ *Addr* — field name, addr
arrayL = *nat* ⇀ *Addr*
rEnv = *varName* ⇀ *Addr* — name of variable in runtime environment, addr

datatype *obj* = *oObject rType fields* | — class instance with class name and fields
oArray rType arrayL

The function oType is used to access the type definition part of an object in the heap:

consts *oType* :: *obj* ⇒ *rType*
primrec
oType (*oObject rT fs*) = *rT*
oType (*oArray rT aL*) = *rT*
consts *oFields* :: *obj* ⇒ *fields*
primrec
oFields (*oObject rT fs*)= *fs*

To create a new object in the heap, the fieldmap has to be initialized. To do this, the field definition list will be taken, and as a result the fieldmap will be generated.

constdefs
init-fields :: (*varName*) *list* ⇒ *fields*
init-fields ≡ *map-of* ∘ *map* (λ(*F*). (*F*,*NullP*))
init-varDef :: (*varDef*) *list* ⇒ *fields*
init-varDef x ≡ *init-fields* (*varDefs-names x*)

Heap

The heap itself is represented as a partial function mapping addresses to objects.

types *heap* = *Addr* ⇀ *obj*

To create a new object on that heap, a free address is required. It will be fetched using the function new-Addr.

constdefs
new-Addr :: *heap* ⇒ *Addr option*
new-Addr h ≡ *if* ∃ *a*. *h a* = *None then Some*(*SOME a*. *h a* = *None*) *else None*

As a result we are now able to update the heap and do a heap extension hext. This heap extension will limit the sort of updates in a way that the type of an object in the heap is not allowed to be modified.

constdefs
hext :: *heap* ⇒ *heap* ⇒ *bool* (- ⊴ - [*51*,*51*] *50*)
h ⊴ *h'* ≡ (∀ *a C fs*. *h a* = *Some*(*oObject C fs*) ⟶ (∃ *fs'*. *h' a* = *Some*(*oObject C fs'*)))
∧ (∀ *a C fs*. *h a* = *Some*(*oArray C fs*) ⟶ (∃ *fs'*. *h' a* = *Some*(*oArray C fs'*)))

At last the function aClass is defined, as it is required more often for the later used rules.

constdefs *aClass* ::*heap* ⇒ *Addr* ⇒ *className*
aClass h addr ≡ *rtClass* (*oType* (*the* (*h addr*)))

Heap extension (⊴) lemmas

With the definition of a heap, and how it could be extended, some lemmas can now be proven using those information.

The very first one is a direct result from the allocation of a new address. As it is not allowed to override an existing object in the heap, the old heap will map to no object in the new address.

lemma *new-Addr-SomeD*:
new-Addr h = *Some a* ⟹ *h a* = *None*

To prove the heap extension for the bigstep semantics, an introduction rule will be required.

lemma *hextI*:
⟦∀ *a C fs*. *h a* = *Some*(*oObject C fs*) ⟶ (∃ *fs'*. *h' a* = *Some*(*oObject C fs'*));
∀ *a C fs*. *h a* = *Some*(*oArray C fs*) ⟶ (∃ *fs'*. *h' a* = *Some*(*oArray C fs'*)) ⟧
⟹ *h* ⊴ *h'*

The next lemmas handle the direct results known from a heap extension. If there is an object with a specific type T on the heap, the type will stay the same in the extended heap.

lemma *hext-objD*: ⟦ *h* ⊴ *h'*; *h a* = *Some*(*oObject T fs*) ⟧ ⟹ ∃ *fs'*. *h' a* = *Some*(*oObject T fs'*)
lemma *hext-arrayD*: ⟦ *h* ⊴ *h'*; *h a* = *Some*(*oArray T fs*) ⟧ ⟹ ∃ *fs'*. *h' a* = *Some*(*oArray T fs'*)

With the definition of the heap, it can be shown that the heap extension is reflexive and transitive.

lemma *hext-refl* [*iff*]: $h \trianglelefteq h$
lemma *hext-trans*: $[\![h \trianglelefteq h'; h' \trianglelefteq h'']\!] \Longrightarrow h \trianglelefteq h''$

The following two lemmas describe how the heap is extended exactly, to create a new object on the heap and to update an object on the heap.

lemma *hext-new* [*simp*]: $h\ a = None \Longrightarrow h \trianglelefteq h(a \mapsto x)$
lemma *hext-upd-obj*: $h\ a = Some\ (oObject\ C\ fs) \Longrightarrow h \trianglelefteq h(a \mapsto (oObject\ C\ fs'))$

end

theory *runtimehelper* **imports** *vpadaption typingrelation statichelper heap*

begin

4.2.9 Runtime model

Owner relation

To get the owner of an object from the heap, the owner function takes the heap and an address as parameter, and the result will be the address of the owner.

constdefs *owner* ::*heap* ⇒ *Addr* ⇒ *OwnerAddr*
owner h addr ≡ *rtOwner* (*oType* (*the* (*h addr*)))

For the owner as modifier property, not only the direct owner is required, but all the owners of the owner, too. To define this relation an inductive set of owners is used, which will be extended by transitivity.

consts *sowner* :: *heap* ⇒ (*Addr* ∗ *OwnerAddr*) *set*
inductive *sowner h*
intros
owner-nullp:⟦*h addr*=*Some x*⟧ ⟹ (*addr*, *owner h addr*) ∈ (*sowner h*)

To reduce the writing work in the lemmas for the owner-as-modifier property, the owner_is_this relation is used for simplicity.

constdefs *owner-is-this* ::*heap* ⇒ *fields* ⇒ *Addr* ⇒ *bool*
owner-is-this h rEnv addr ≡
if ((*addr*,*owner h* (*the* (*rEnv this*))) ∈ (*sowner h*)^+) *then True*
else False

Type dynamization and runtime subtyping

To define a runtime subtyping static types have to be dynamized. For the universe modifiers the corresponding address will be looked up using the dynOM function.

constdefs *dynOM* :: *heap* ⇒ *Addr* ⇒ *OM* ⇒ *Addr*
dynOM h addr-this u ≡
if (*u*=*Peer* ∨ *u*=*This*) *then* (*owner h addr-this*)
else if (*u*=*Rep*) *then addr-this*
else AnyAd

Together with the dynOM function, the dynType function can dynamize the type recursively.

consts *dynType* :: *heap* ⇒ *Addr* ⇒ *uType* ⇒ *rType*
primrec
dynType h addr-this (*sObject u C*) = *rObject* (*dynOM h addr-this u*) *C*
dynType h addr-this (*sArray u T*) = *rArray* (*dynOM h addr-this u*) (*dynType h addr-this T*)

As dynamization should be done by the simplifier, the following lemmas help to unfold the functions where possible, and still keep a nice structure in the theory files.

lemma [*iff*]:*dynType h addr* (*sObject Rep C*)=(*rObject addr C*) **by**(*simp add*:*dynOM-def*)
lemma [*iff*]:*dynType h addr* (*sObject Peer C*)=(*rObject* (*owner h addr*) *C*) **by**(*simp add*:*dynOM-def*)
lemma [*iff*]:*dynType h addr* (*sObject This C*)=(*rObject* (*owner h addr*) *C*) **by**(*simp add*:*dynOM-def*)
lemma [*iff*]:*dynType h addr* (*sObject Any C*)=(*rObject AnyAd C*) **by**(*simp add*:*dynOM-def*)
lemma [*iff*]:*dynType h addr* (*sObject Unknown C*)=(*rObject AnyAd C*) **by**(*simp add*:*dynOM-def*)

The dynamization is not directly required for the runtime subtyping, but as types have to be compared during runtime, too, they are both needed later on.

consts *rtsubs* :: *classTable* ⇒ (*rType* ∗ *rType*) *set*
syntax
-rtsubt :: [*classTable*, *rType*, *rType*] ⇒ *bool* (- ⊢ - <:$_r$ - [*80*,*80*,*80*] *80*)
translations
CT ⊢ *rS* <:$_r$ *rT* ⇌ (*rS*, *rT*) ∈ (*rtsubs CT*)

In comparison to the static subtyping, runtime subtyping is really simple. The class parts of the types have to be in a subclassing relation. Only the owner-part has to have a separate handling, where on both sides either the owner is equal, or on the right side the owner is the AnyAd. Although static and runtime subtyping are comparable in many cases, there exists no bijective function to transform one representation into the other one.

inductive *rtsubs CT*
intros
rtsubsarray: ⟦*CT* ⊢ *rS* <:$_r$ *rT*; *a*=*b* ∨ *b*=*AnyAd*⟧ ⟹ *CT* ⊢ *rArray a rS* <:$_r$ *rArray b rT*
rtsubsob: ⟦*a*=*b* ∨ *b*=*AnyAd*; *CT* ⊢ *rC* [: *rD*⟧ ⟹ *CT* ⊢ *rObject a rC* <:$_r$ *rObject b rD*
rtsubsmix: ⟦*a*=*b* ∨ *b*=*AnyAd*⟧ ⟹ *CT* ⊢ *rArray a rS* <:$_r$ *rObject b Object*

The runtime subtyping can now be used, to express the static type of an address with respect to a viewpoint.

consts *ftypeOK* :: (*classTable* ∗ *heap* ∗ *Addr* ∗ *Addr* ∗ *uType*) *set*
syntax
-ftypeOK :: [*classTable*, *heap*, *Addr*, *Addr*, *uType*] ⇒ *bool* (-, -, - ⊢ - : - [*80*,*80*,*80*,*80*] *80*)
translations
CT, *h*, *me* ⊢ *addr* : *T* ⇌ (*CT*, *h*, *me*, *addr*, *T*) ∈ *ftypeOK*

In words you can say to that relation: On the heap ”h”, ”me” sees ”addr” as type ”T”.

inductive *ftypeOK*
intros
Tcomp:
⟦*h addr*=*Some*(*Obj*);
CT ⊢ (*oType Obj*) <:$_r$ *dynType h me T*;
utOM T=*This* ⟶ *me*=*addr*⟧
⟹ *CT*, *h*, *me* ⊢ *addr* : *T*
Tcompnull: *CT*, *h* ,*me* ⊢ *NullP* : *T*

consts *ftypesOK* :: (*classTable* ∗ *heap* ∗ *Addr* ∗ *Addr list* ∗ *uType list*) *set*
syntax
-ftypesOK :: [*classTable*, *heap*, *Addr*, *Addr list*, *uType list*] ⇒ *bool* (-, -, - ⊢$^+$ - : - [*80*,*80*,*80*,*80*] *80*)
translations
CT, *h*, *me* ⊢$^+$ *addr* : *T* ⇌ (*CT*, *h*, *me*, *addr*, *T*) ∈ *ftypesOK*
inductive *ftypesOK*
intros
Tnil:
CT, *h*, *me* ⊢$^+$ [] : []
Tcons: ⟦ *CT*, *h*, *me* ⊢ *addr* : *t*; *CT*, *h*, *me* ⊢$^+$ *addrs* : *ts*⟧ ⟹ *CT*, *h*, *me* ⊢$^+$ (*addr*#*addrs*) : (*t*#*ts*)

Another issue, which can be handled by type dynamization is the matching of a runtime environment to a static environment.

consts *rEnvtosEnv* :: (*classTable* ∗ *heap* ∗ *rEnv* ∗ *staticEnv*) *set*
syntax
-rEnvtosEnv :: [*classTable*, *heap*, *rEnv*, *staticEnv*] ⇒ *bool* (-, - ⊢ - : - [*80*,*80*,*80*,*80*] *80*)
translations
CT, *h* ⊢ *mrEnv* : *msEnv* ⇌ (*CT*, *h*, *mrEnv*, *msEnv*) ∈ *rEnvtosEnv*
inductive *rEnvtosEnv*
intros
rtosrEnvtosEnv:
⟦*mrEnv this*= *Some*(*adthis*);
∀ *name addr*. (*mrEnv*(*name*)=*Some*(*addr*) ⟶
(*addr* = *NullP* ⟶ (∃ *T*.(*msEnv name*=*Some*(*T*)))) ∧
(*addr* ≠ *NullP* ⟶ (∃ *T*.(*msEnv name*=*Some*(*T*) ∧
CT, *h*, *adthis* ⊢ *addr* : *T*))))⟧
⟹ *CT*, *h* ⊢ *mrEnv* : *msEnv*

Resulting lemmas

As subtyping does not modify the heap, it can be written down in an alternative way, much better usable for proving other lemmas.

lemma *allsubtyping*:⋀ *addrs*. *CT*, *h2*, *me* ⊢⁺ *addrs* : *Ts-s*⟹∀ *i*< *length Ts-s*. *CT*, *h2*, *me* ⊢ *addrs*!*i* : *Ts-s*!*i*

At last the information from the heap extensions can be combined with the information from the set of owners to show that all owner sets in the old heap are also sets of the extended heap.

lemma *hextkeepowner*:⟦ *h* ◁ *hf*; *h a*=*Some*(*oObject x y*)⟧ ⟹ (*a*, *b*) ∈ (*sowner hf*)=((*a*, *b*) ∈ (*sowner h*))
end

theory *welltyped* **imports** *static runtimehelper*

begin

4.2.10 Welltypedness definition

The definition of a well typed runtime environment is a complex issue. To simplify the definition it is split up into the parts of the environment.

The first part is the welltypedness of an address in the heap.
An address could either map to an object, an array or be the nullpointer. In the case of an object, the fields of that object have to be subtypes of the type defined by the class. The field-T relation will guarantee that:

constdefs *field-T* ::*classTable* ⇒ *heap* ⇒ *Addr* ⇒ *fields* ⇒*className* ⇒ *varName* ⇒ *uType* ⇒ *bool*
field-T CT h addr myFields C fname T≡
(∃ *faddr*.
(*myFields fname*=*Some*(*faddr*) ∧
(∃ *ob*. *h faddr* = *Some*(*ob*) ∨ *faddr*=*NullP*) ∧
(*CT*, *h*, *addr* ⊢ *faddr* : *T*)))

A well typed address is not defined recursive for its fields to simplify the definition of the function. It can be done, because there is no need to directly declare that the addresses stored in the fields of

an object are well typed.
It will result from the information that they are either a nullpointer or they are in the domain of the heap. All other information can be found in the definition of a well formed heap. For the runtime type of the object there are a few limitations, too. First of all, the owner of the objects has to follow some rules. Besides that, there are some more limitations, depending if it is an array or a normal object:

constdefs *wta* :: *classTable* ⇒ *heap* ⇒ *Addr* ⇒ *bool* (-,- $_a$⊢ - [*80*,*80*,*80*] *80*)
(*CT*, *h* $_a$⊢ *addr*) ≡
(*addr*≠*NullP* ⟶ ((
∃ *oT owneraddr myFields C CDef* .(
Some(*oObject oT myFields*)=*h addr* ∧
rtClass oT=*C* ∧ *is-objectT*(*oT*) ∧
owneraddr≠ *addr* ∧
(*owner h addr* = *owneraddr*) ∧
((∃ *ownerObj*. (*h owneraddr*=*Some*(*ownerObj*) ∧ *is-object*(*ownerObj*))) ∨ (*owneraddr*=*NullP*) ∨ (*owneraddr*=*AnyOwner*)) ∧
CT(*C*)=*Some*(*CDef*) ∧
(∀ *fname T*. *ftype*(*CT*, *C*, *fname*)=*Some*(*T*) ⟶ (*field-T CT h addr myFields C fname T*)))) ∨ (
∃ *u T owneraddr myFields* .(
Some(*oArray* (*rArray u T*) *myFields*)=*h*(*addr*) ∧
(∀ *n addr*. *myFields n*=*Some*(*addr*) ⟶ (*addr*=*NullP* ∨ (∃ *oB* . (*h addr*=*Some*(*oB*) ∧
CT ⊢ (*oType oB*) <:$_r$ *T*)))) ∧
owneraddr≠ *addr* ∧
(*owner h addr* = *owneraddr*) ∧
((∃ *ownerObj*. (*h owneraddr*=*Some*(*ownerObj*) ∧ *is-object*(*ownerObj*))) ∨ (*owneraddr*=*NullP*) ∨ (*owneraddr*=*AnyOwner*))))))

consts *well-typed-addrs* :: (*classTable* ∗ *Addr list* ∗ *heap*) *set*
syntax
-well-typed-addrs :: [*classTable*, *heap*, *Addr list*] ⇒ *bool* (-, - $_a$⊢$^+$ - [*80*,*80*,*80*] *80*)
translations
CT, *h* $_a$⊢$^+$ *addrs* ⇌ (*CT*, *addrs*, *h*) ∈ *well-typed-addrs*
inductive *well-typed-addrs*
intros
wta-nil: *CT*, *h* $_a$⊢$^+$ []
wta-cons: ⟦ *CT*, *h* $_a$⊢$^+$ *addrs*; *CT*, *h* $_a$⊢ *addr*⟧ ⟹ *CT*, *h* $_a$⊢$^+$ (*addr*#*addrs*)

The definition of a well typed address is required for the definition of the wellformedness of the heap. All addresses that are in the domain of the heap have to be well typed. A second limitation is that the addresses AnyOwner,NullP and AnyAd are not allowed to be in the domain of the heap.

consts *well-typed-heap* :: (*classTable* ∗ *heap*) *set*
syntax
-well-typed-heap :: [*classTable*, *heap*] ⇒ *bool* (- $_h$⊢ - [*80*,*80*] *80*)
translations
CT $_h$⊢ *h* ⇌ (*CT*, *h*) ∈ *well-typed-heap*
inductive *well-typed-heap*
intros
wth:⟦∀ *addr obj*. (*h addr*=*Some*(*obj*)) ⟶ (*CT*, *h* $_a$⊢ *addr*);
h AnyOwner=*None*;
h NullP=*None*;
h AnyAd =*None*⟧
⟹ *CT* $_h$⊢ *h*

At last a welltyped environment guarantees a welltyped heap, a welltyped classtable, and that the runtime environment maps only to valid addresses.

consts *well-typed-env* :: (*classTable* ∗ *heap* ∗ *rEnv*) *set*
syntax
-well-typed-env :: [*classTable*, *heap*, *rEnv*] ⇒ *bool* (-, - $_w$⊢ - [*80*,*80*] *80*)
translations
CT, *h* $_w$⊢ *Env* ⇌ (*CT*, *h*, *Env*) ∈ *well-typed-env*
inductive *well-typed-env*
intros
wte:⟦⊢ *CT*;
CT $_h$⊢ *h*;
∀ *name addr*. *rEnv name*=*Some*(*addr*) ⟶ (*CT*, *h* $_a$⊢ *addr*);
rEnv this≠*Some*(*NullP*);
h (*the* (*rEnv this*))=*Some*(*oObject rT rf*);
rEnv this ≠*None*⟧
⟹ *CT*, *h* $_w$⊢ *rEnv*

Extending a wellformed environment

The definition of the heap extension together with the definition of welltypedness result in some new lemmas which can be derived from them.

The first one is the extension of a dynType definition. It does not require the whole information of a welltyped environment, but as that information will be given later on, the whole definition will be taken as assumption.

lemma *hext-dyntype*:⋀ *Tr*. ⟦*h*⊴*hf*; *CT*, *h* $_w$⊢ *Env*; *dynType h y Ts*=*Tr*; *h y*=*Some* (*Ob*)⟧ ⟹*dynType hf y Ts*=*Tr*

The next information that can be derived from those assumptions is that a runtime environment fitting to a static environment will also fit to each other after the heap is extended

lemma *hextkeepEnv*:⟦ *CT*, *h* $_w$⊢ *rEnv*; *h* ⊴ *h′*; *CT*, *h* ⊢ *rEnv* : *sEnv*⟧ ⟹ *CT*, *h′* ⊢ *rEnv* : *sEnv*

From that information, and the information about a welltyped heap after an extension, you can prove that the new environment is well typed, too.

lemma *wtehext*:⟦ *CT*, *h* $_w$⊢ *rEnv*; *h*⊴*hf*; *CT* $_h$⊢ *hf*⟧ ⟹ *CT*, *hf* $_w$⊢ *rEnv*

It is also known, that the current viewpoint is defined in the heap:

lemma *wte-this*:*CT*, *h* $_w$⊢ *Env* ⟹ ∃ *ob*. *h*(*the* (*Env this*))=*Some*(*ob*)

end

theory *bigstep* **imports** *welltyped* **begin**

4.2.11 Bigstep formalization

The bigstep formalization will be done using an inductively defined set. Both the definition for executing a single expression as well as executing multiple expressions have to be declared together. Most of the rules just follow the informal definition of the bigstep semantics. Only in a few cases they are written down in a more complicated way to allow Isabelle to use the rules more easily.

consts *bigstep* :: (*classTable* ∗ *heap* ∗ *rEnv* ∗ *exp* ∗ *heap* ∗ *Addr*) *set*
syntax
-bigstep :: [*classTable*, *heap*, *rEnv*, *exp*, *heap*, *Addr*] ⇒ *bool* (- ⊢ -, -, - ⇝ -, - [*80*,*80*,*80*] *80*)

translations
$CT \vdash h,\ Env,\ e \rightsquigarrow h',\ addr \rightleftharpoons (CT,\ h,\ Env,\ e,\ h',\ addr) \in bigstep$

consts $bigsteps :: (classTable * heap * rEnv * exp\ list * heap * Addr\ list)\ set$
syntax
$-bigsteps :: [classTable, heap, rEnv, exp\ list, heap, Addr\ list] \Rightarrow bool\ (- \vdash -, -, - \rightsquigarrow+ -, - [80,80,80]\ 80)$

translations
$CT \vdash h,\ Env,\ e \rightsquigarrow+ h',\ addr \rightleftharpoons (CT,\ h,\ Env,\ e,\ h',\ addr) \in bigsteps$

inductive *bigstep bigsteps*
intros

os-nil: $CT \vdash h,\ Env,\ [] \rightsquigarrow+ h,\ []$

os-cons:
$[\![\ CT \vdash h,\ Env,\ e0 \rightsquigarrow h1,\ addr;$
$CT \vdash h1,\ Env,\ es \rightsquigarrow+ hf,\ addrs\ ]\!]$
$\Longrightarrow CT \vdash h,\ Env,\ e0\#es \rightsquigarrow+ hf,\ addr\#addrs$

os-null:
$CT \vdash h,\ Env,\ Null \rightsquigarrow h,\ NullP$

os-var:
$[\![\ Env\ vn{=}Some(addr)]\!]$
$\Longrightarrow CT \vdash h,\ Env,\ Var\ vn \rightsquigarrow h,\ addr$

os-read:
$[\![\ CT \vdash h,\ Env,\ e \rightsquigarrow hf,\ addr\text{-}e;$
$hf\ addr\text{-}e = Some(oObject\ T\ eField);$
$addr\text{-}e \neq NullP;$
$eField\ f{=}Some(addr)]\!]$
$\Longrightarrow CT \vdash h,\ Env,\ FieldProj\ e\ f \rightsquigarrow hf,\ addr$

os-new:
$[\![\ new\text{-}Addr\ h{=}Some(addr);$
$addr \neq NullP;$
$addr \neq AnyAd;$
$addr \neq AnyOwner;$
$Env\ this{=}Some(myself);$
$h\ myself{=}Some(myselfo);$
$(u{=}Peer \vee u{=}Rep) \longrightarrow dynType\ h\ myself\ (sObject\ u\ C) = mType;$
$u{=}Any \longrightarrow rObject\ AnyOwner\ C{=}mType;$
$is\text{-}objectT(mType);$
$CT(C) = Some(CDef);$
$fields(CT,\ C){=}myFields;$
$init\text{-}varDef\ myFields{=}obFields;$
$h(addr{\mapsto}(oObject\ mType\ obFields)){=}hf$
$]\!] \Longrightarrow CT \vdash h,\ Env,\ New\ u\ C \rightsquigarrow hf,\ addr$

os-cast:

$[\![\ Env\ this{=}Some(myself);$
$hf\ myself{=}Some(oObject\ msType\ msFields);$
$CT \vdash h,\ Env,\ e \rightsquigarrow hf,\ addr;$
$CT,\ hf,\ the\ (Env\ this) \vdash\ addr : T]\!]$
$\Longrightarrow\ CT \vdash h,\ Env,\ Cast\ T\ e \rightsquigarrow hf,\ addr$

os-upd:
$[\![\ CT \vdash h,\ Env,\ e1 \rightsquigarrow h1,\ obaddr;$
$CT \vdash h1,\ Env,\ e2 \rightsquigarrow h2,\ addr;$
$obaddr \neq NullP;$
$h2\ obaddr{=}Some(oObject\ obType\ Fields);$
$h2(obaddr{\mapsto}(oObject\ obType\ (Fields(f{\mapsto}addr)))){=}hf\ ]\!]$
$\Longrightarrow CT \vdash h,\ Env,\ FieldUpdate\ e1\ f\ e2 \rightsquigarrow hf,\ addr$

os-invk:
$[\![\ CT \vdash h,\ Env,\ e \rightsquigarrow h1,\ paddr;$
$CT \vdash h1,\ Env,\ elist \rightsquigarrow{+}\ h2,\ addrs;$
$paddr{\neq}\ NullP;$
$h2\ paddr{=}Some(oObject\ rt\ fs);$
$mdata(CT,\ aClass\ h2\ paddr,\ m){=}Some(mDef);$
$mBody\ mDef{=}e1;$
$mParams\ mDef{=}vDefs;$
$varDefs\text{-}names\ vDefs = xs;$
$[xs[{\mapsto}]addrs](this{\mapsto}paddr){=}newEnv;$
$CT \vdash h2,\ newEnv\ ,\ e1 \rightsquigarrow hf,\ addr]\!]$
$\Longrightarrow CT \vdash h,\ Env,\ MethodInvk\ e\ m\ elist{\rightsquigarrow}\ hf,\ addr$

os-newarray:
$[\![\ new\text{-}Addr\ h\ {=}Some(addr)\ ;$
$addr{\neq}NullP;$
$addr{\neq}AnyAd;$
$addr{\neq}AnyOwner;$
$Env\ this{=}Some(myself);$
$h\ myself{=}Some(myselfo);$
$(u{=}Peer \vee u{=}Rep) \longrightarrow dynType\ h\ myself\ (sArray\ u\ T) = mType;$
$u{=}Any \longrightarrow rArray\ AnyOwner\ (dynType\ h\ myself\ T){=}mType;$
$is\text{-}arrayT(mType);$
$\forall\ i{<}\ n\ .\ (myArray\ i{=}Some(NullP));$
$\forall\ i \geq n\ .(\ myArray\ i{=}None);$
$h(addr{\mapsto}(oArray\ mType\ myArray)){=}hf$
$]\!]\ \Longrightarrow CT \vdash h,\ Env,\ NewArray\ \ u\ T\ n \rightsquigarrow hf,\ addr$

os-arrayread:
$[\![CT \vdash h,\ Env,\ e \rightsquigarrow hf\ ,\ addr\text{-}e;$
$hf\ addr\text{-}e{=}Some(oArray\ T\ eField);$
$addr\text{-}e{\neq}NullP;$
$eField\ n{=}Some(addr)]\!]$
$\Longrightarrow CT \vdash h,\ Env,\ ArrayProj\ e\ n \rightsquigarrow hf\ ,\ addr$

os-arrayupd:
$[\![\ CT \vdash h,\ Env,\ e1 \rightsquigarrow h1,\ obaddr;$
$CT \vdash h1,\ Env,\ e2 \rightsquigarrow h2,\ addr;$
$obaddr \neq NullP;$
$h2\ obaddr{=}Some(oArray\ (rArray\ own\ rT)\ Fields);$
$n \in dom(Fields);$

$\forall\ ob\ .\ (h2\ addr{=}Some(ob) \longrightarrow CT \vdash (oType\ ob) <:_r rT);$
$h2(obaddr{\mapsto}(oArray\ (rArray\ own\ rT)\ (Fields(n{\mapsto}addr))){=}hf\ ]\!]$
$\Longrightarrow CT \vdash h, Env, ArrayUpdate\ e1\ n\ e2 \rightsquigarrow hf, addr$

Some first results can be extracted from the bigstep semantics. One of them says that a bigstep will result in a new heap that extends the old heap.

lemma *bigstephext*:$CT \vdash h, Env, e1 \rightsquigarrow h1, obaddr \Longrightarrow h \trianglelefteq h1$

The same also could be said for the bigstep of multiple expressions.

lemma *bigstepshext*:$CT \vdash h, Env, e1 \rightsquigarrow+ h1, obaddr \Longrightarrow h \trianglelefteq h1$

Another interesting result is the welltypedness of a new environment, created with the result from a bigstep over a list of expressions. This will be required to prove the case of method invocation.

lemma *wte-new-env*:$[\![$
$CT,h2\ {}_w\vdash Env;$
$[xs\ [{\mapsto}]\ addrs, this \mapsto paddr] = newEnv\ ;$
$paddr \neq NullP\ ;$
$h2\ paddr{=}Some\ (oObject\ rT\ rf);$
$CT \vdash h1, Env, elist \rightsquigarrow+ h2, addrs;$
$CT, h2, the\ (Env\ this) \vdash^+ addrs : Ts\text{-}s;$
$CT, h1, the\ (Env\ this) \vdash paddr : T\text{-}e]\!]$
$\Longrightarrow CT, h2\ {}_w\vdash newEnv$
end

theory *topologicalproofs* **imports** *welltyped*

begin

4.2.12 Topological typesystem proofs

The proofs for the viewpoint adaption in the topological typesystem will show, that the static definition of a viewpoint adaption does fit to the runtime definition of a viewpoint adaption.

To prove the whole viewpoint adaption, there are different sub-lemmas to prove.

The first one, simple and, if one knows the definition for the runtime typing rule, it is clear to show that an address that has the static universe modifier "This" has to have the same address as the current viewpoint.

lemma *thistypethis*:$[\![CT, hf, addre \vdash addr : (sObject\ This\ C);\ addr{\neq}NullP]\!] \Longrightarrow addre{=}addr$

In the next step the relation of two type dynamization for an address addr and a viewpoint paddr will be shown.

lemma *adapt-from-dynType*: $\bigwedge\ addr\ .$
$[\![CT, hf, paddr \vdash addr : sObject\ u\text{-}Te\ C\text{-}te;$
$addr{\neq}NullP]\!]$
$\Longrightarrow CT \vdash dynType\ hf\ addr\ T\text{-}u <:_r dynType\ hf\ paddr\ (u\text{-}Te \rhd^t T\text{-}u)$

Another lemma required to prove the viewpoint adaption sound is the transitivity of runtime types.

lemma *dynType-trans*: $\bigwedge\ a\ b.\ [\![$
$CT \vdash a <:_r b;$
$CT \vdash b <:_r c]\!]$
$\Longrightarrow CT \vdash a <:_r c$
— Again an induction

apply (*induct c*)
— will be used.

Ultimately, it can be shown that the viewpoint adaption is sound using the defined rules and the heap definition

lemma *adapt-from-vpT*: $\bigwedge$ *addr* . ⟦
CT, *hf*, *addr-e* ⊢ *addr* : *T-u*;
CT, *hf*, *the* (*Env this*) ⊢ *addr-e* : *sObject u-Te C-te*;
addr-e ≠ *NullP*⟧
⟹ *CT*, *hf*, *the* (*Env this*) ⊢ *addr* : (*u-Te* $\triangleright^t$ *T-u*)
— The proof can be done most of the time straightforward with the use of case distinctions like
apply (*case-tac u-Te*,*simp-all*)
— and
apply (*case-tac OM*,*simp-all*)
— .

The next step will be the integration of subtyping and its relation from static to runtime subtyping. If there is a type T and a supertype T-e, and if there exists a runtime-subtyping relation of some runtime type to a dynamization of the static type T-e, the runtime subtyping-relation from that runtime type to a dynamization of the static type T is also true.

lemma *st-dynType*:$\bigwedge$ *T-e x*. ⟦*addr-e* ≠ *NullP*; *CT* ⊢ *T-e* <: *T*; *CT* ⊢ *x* <:$_r$ *dynType hf addr-e T-e*
⟧
⟹ *CT* ⊢ *x* <:$_r$ *dynType hf addr-e T*
— To prove that property, an induction is done over
apply (*induct T*)
— resulting in a case for arrays, and a case for normal objects.

This can then be used in another way using the ftype relation for subtypes, saying, that if an address addr-e sees another address addr as type T-e, it can see the address also as any supertype of T.

lemma *adapt-subtype*:⟦
addr-e≠*NullP*;
CT, *hf*, *addr-e* ⊢ *addr* : *T-e*;
CT ⊢ *T-e* <: *T*⟧
⟹ *CT*, *hf*, *addr-e* ⊢ *addr* : *T*

Now the final lemma for the adaption from a viewpoint can be shown using the adapt-from-vpT and the adapt-subtype lemma:

lemma *adapt-from-vp*: **assumes**
this-notnull:*the* (*Env this*)≠ *NullP* **and**
addre-addr-T-u:*CT*, *hf*, *addr-e* ⊢ *addr* : *T-u* **and**
this-addre-uC:*CT*, *hf*, *the* (*Env this*) ⊢ *addr-e* : *sObject u C* **and**
addre-notnull:*addr-e* ≠ *NullP* **and**
T-u-oo-Tu:*CT* ⊢ (*u* $\triangleright^t$ *T-u*) <: *T* **shows**
CT, *hf*, *the* (*Env this*) ⊢ *addr* : *T*
proof −
from *this-notnull addre-addr-T-u this-addre-uC addre-notnull*
have *adapt-f-vpT*:*CT*,*hf*,*the* (*Env this*) ⊢ *addr* : (*u* $\triangleright^t$ *T-u*)
by(*erule-tac u-Te*=*u* **and** *C-te*=*C* **in** *adapt-from-vpT*,*simp-all*)
from *this-notnull adapt-f-vpT T-u-oo-Tu* **show** *?thesis*
by(*simp-all add*:*adapt-subtype*)
qed

The next part will be the proof to adapt a type to a viewpoint.
To prove that the following lemma is required as a runtime-subtyping relation:

lemma *adapt-to-dynType*: $\bigwedge$ *addr* . ⟦
owner hf paddr≠AnyAd;*paddr≠AnyAd*;
CT,*hf*,*paddr* ⊢ *addr* : *sObject u-Te C-te*;*addr≠NullP*;
notUnknown (*u-Te* $\triangleright^t$ *T-u*)
⟧
⟹ *CT* ⊢ *dynType hf paddr* (*u-Te* $\triangleright^t$ *T-u*) $<:_r$ *dynType hf addr T-u*
— The proof does an induction over the type T-u
proof (*induct T-u*)
— The rest can be done by an induction over the universe modifiers together with the limitations from the notUnknown function.
qed

Using the previous information all together, the adaption to a viewpoint can be proven very simply. As a result the following lemma can be used further on:

lemma *adapt-to-vp*:⟦
CT, *hf*, *paddr* ⊢ *obaddr* : *sObject u-Te C-Te*;
CT, *hf*, *paddr* ⊢ *addr* :(*u-Te* $\triangleright^t$ *Ti*);
owner hf paddr≠ AnyAd;
paddr≠AnyAd;
obaddr≠ NullP;
T-e=sObject u-Te C-T;
obaddr≠AnyAd;
notUnknown (*u-Te* $\triangleright^t$ *Ti*)⟧
⟹ *CT*, *hf*, *obaddr* ⊢ *addr* : *Ti*

At last the same lemma will be provided for lists of addresses, which is required in the case of method invocation for the adaption of the parameters.

lemma *adapt-to-vps*:$\bigwedge$ *Ts-s Tsu*. ⟦
obaddr≠AnyAd;
owner hf paddr≠AnyAd;
paddr≠AnyAd;
T-e1=sObject u-Te C-Te;
CT, *hf*, *paddr* ⊢ *obaddr* : *T-e1*;
CT, *hf*, *paddr* $\vdash^+$ *addrs* : *Ts-s*;
∀ *i* <*length Tsu*. (*utOM T-e1* $\triangleright^t$ *Tsu*!*i*)=*Ts-s*!*i* ;
obaddr ≠ *NullP*;
length addrs=*length Tsu*;
∀ *x* ∈ *set Ts-s* . *notUnknown*(*x*)⟧
⟹ *CT*, *hf*, *obaddr* $\vdash^+$ *addrs* : *Tsu*

end
theory *typesafety* **imports** *bigstep topologicalproofs* **begin**

ML⟪
Unify.search-bound := *40*;
Unify.trace-bound := *40*;
⟫

In the previous theory files all the information required to prove the typesafety is specified. Together with the proofs about the topology now the type preservation can be proven: To apply the induction rule derived from the bigstep semantics, the type preservation has to have a special form. The bigstep itself will be used as the only direct assumption. From this assumption it can be concluded that with the declared preconditions for the welltyping the type preservation can be shown:

As the bigstep semantics has to handle single expressions as well as lists of expressions, both cases have to be proven together.

lemma shows *bigstep-well-typingX*:
((*CT* ⊢ *h*, *rEnv*, *e* ⇝ *h3*, *addr*)
⟹ (⋀ *T sEnv*. ⟦(*CT*, *h* $_w$⊢ *rEnv*);
(*CT*, *sEnv* ⊢ *e* : *T*) ;
(*CT*, *h* ⊢ *rEnv* : *sEnv*)⟧
⟹ (*CT*, *h3* $_w$⊢ *rEnv*) ∧
CT, *h3*, *the* (*rEnv this*) ⊢ *addr* : *T*))
and *bigsteps-well-typingX*:
(*CT* ⊢ *h*, *rEnv*, *es* ⇝+ *h2*, *addrs*
⟹ (⋀ *Ts sEnv*. ⟦ (*CT*, *h* $_w$⊢ *rEnv*);
(*CT*, *sEnv* ⊢$^+$ *es* : *Ts*) ;
(*CT*, *h* ⊢ *rEnv* : *sEnv*)⟧
⟹ ((*CT*, *h2* $_w$⊢ *rEnv*) ∧
CT, *h2*, *the* (*rEnv this*) ⊢$^+$ *addrs* : *Ts*)))
— The proof will run with an induction over the bigstep semantic:
proof (*induct rule*:*bigstep-bigsteps-induct*)
— The
case (*os-nil CT Env h Ts sEnv*)
— is simple to show, as it does not change the heap and just result in an empty list of addresses. **next**
case (*os-cons CT Env addr addrs e0 es h h1 hf Ts sEnv*) **thus** *?case* **proof** −
— To prove the type preservation for a list of expressions sound is not too complicated either, as the required information will be generated in most parts by the induction step . **qed**
next
— As Null is a very simple expression, there has not much to be done to prove it typesafe.
case (*os-null CT Env h T sEnv*) **thus** *?case*
by (*simp add*:*Tcompnull*)
next
— The last really simple case is the expression to read a variable from the environment.
case (*os-var CT Env addr h vn Ts sEnv*) **thus** *?case* **done**
next
— It is not too complicated to prove the cast expression to preserve the type, as there is a typecheck done during the bigstep.
case (*os-cast CT Env Tc addr e h hf ThisFields ThisType ThisAddr T sEnv*)

— The next cases are much more difficult to prove.
The first one shows, that a new object on the heap keeps all the information.
case (*os-new C CDef CT Env addr h hf mType myFields thisAddr ThisOb EmptyField u NewType sEnv*) **thus** *?case* **proof** −
— The first step is to show that the created object will match to the statically defined type.
from *x15 NWisnC x12 x11 hext x6 x15 addrnot0 env-this hf-this addrnotthis NewPeer NewRep x13 def-addr* **have** *typeswell*:*CT*, *hf*, *the* (*Env this*) ⊢ *addr* : *NewType*
— Afterwards it has to be shown that the new address is welltyped in the resulting heap.
from *x15 x2 h-this x6 x6a x1 def-addr x8 x9 x10 x12 addrnot0 addrnotany2 OMisPeerorRep NewRep NewPeer NewAny addrnotthis addrnotany hf-addr hf-this h-this t-resolv* **have** *wta-addr* : *CT*,*hf* $_a$⊢ *addr* **apply** −
— With all those information the type preservation in the case of the new expression can be shown.
qed

next
— In the next step the case of fieldreads will be proven sound.
case (*os-read CT Env To addr addr-e e eField f h hf T sEnv*) **thus** *?case* **proof** −
— Using the induction lemma will result in the following assumptions for the proofs:
have *x1*:*CT* ⊢ *h*, *Env*, *e* ⇝ *hf*, *addr-e*
and *x2*:⋀*sEnv T*.
⟦*CT*, *h* $_w$⊢ *Env*; *CT*,*sEnv* ⊢ *e* : *T*; *CT*, *h* ⊢ *Env* : *sEnv*⟧
⟹ *CT*, *hf* $_w$⊢ *Env* ∧ *CT*, *hf*, *the* (*Env this*) ⊢ *addr-e* : *T*
and *x10*:*hf addr-e* = *Some*(*oObject To eField*)

and *x5*:*eField f = Some addr*
and *x6*:*CT*, *h* $_w\vdash$ *Env*
and *x7*:*CT*, *sEnv* $\vdash$ *FieldProj e f* : *T*
and *x8*:*CT*, *h* $\vdash$ *Env* : *sEnv* **.**
— The assumption x7 has to be unfolded.
from *T-xtemp2* **have** *T-x*:*CT*, *sEnv* $\vdash$ *e* : *sObject u C* **and** *T-u-def*:*ftype*(*CT*, *C*, *f*) = *Some T-u* **and** *T-OM-def*:(*CT* $\vdash$ ((*u*) $\triangleright^t$ *T-u*) <: *T*)
by *auto*

— It can by seen easily from the assumptions that the resulting heap is welltyped.
from *T-x x6 x8 x2* **have** *wte-hf*:*CT*, *hf* $_w\vdash$ *Env* **and** *addr-e-T*:*CT*, *hf*, *the* (*Env this*) $\vdash$ *addr-e* : *sObject u C*
by (*simp*, *blast*)
— The next step is to prove the type of the addr sound:
from *x10 wte-hf T-u-def addr-e-T addr-e-typed x3 x4 x5* **have** *addr-e-addr-T-u*:*CT*, *hf*, *addr-e* $\vdash$ *addr* : *T-u*
— Using the viewpoint adaption together with the unfolded typing rule, the type of the result addr from the viewpoint of this can be shown.
from *addr-e-addr-T-u T-u-def addr-e-T x3 T-OM-def x6* **have** *addr-T*:*CT*, *hf*, *the* (*Env this*) $\vdash$ *addr* : *T*
— The welltypedness of the resulting heap together with the type of the result address will show the case.
from *wte-hf addr-T* **show** *?case* **by** *simp*
qed
next
— In some parts the proof for the fieldupdate follows the same way as the fieldread, but not always. Where in the fieldread the complicated part was to proof the result address being a subtype of the expected type, the difficulty in the case of the fieldupdate is to show that the heap is well typed.
case (*os-upd CT Env Fields addr e1 e2 f h h1 h2 hf obType obaddr T sEnv*) **thus** *?case* **proof** −
— Again a lot of assumptions are generated from the induction, which can be used for the proof later on:
have *x1*:*CT* $\vdash$ *h*, *Env*, *e1* $\rightsquigarrow$ *h1*, *obaddr*
and *x2*:$\bigwedge T$. ⟦*CT*, *h* $_w\vdash$ *Env*; *CT*, *sEnv* $\vdash$ *e1* : *T*; *CT*, *h* $\vdash$ *Env* : *sEnv*⟧
$\Longrightarrow$ *CT*, *h1* $_w\vdash$ *Env* $\wedge$ *CT*, *h1*, *the* (*Env this*) $\vdash$ *obaddr* : *T*
and *x3*:*CT* $\vdash$ *h1*, *Env*, *e2* $\rightsquigarrow$ *h2*, *addr*
and *x4*:$\bigwedge T$. ⟦*CT*,*h1* $_w\vdash$ *Env* ; *CT*, *sEnv* $\vdash$ *e2* : *T*; *CT*, *h1* $\vdash$ *Env* : *sEnv*⟧
$\Longrightarrow$ *CT*, *h2* $_w\vdash$ *Env* $\wedge$ *CT*, *h2*, *the* (*Env this*) $\vdash$ *addr* : *T*
and *x6*:*h2 obaddr = Some* (*oObject obType Fields*)
and *x8*:*h2*(*obaddr* $\mapsto$ (*oObject obType* (*Fields*(*f* $\mapsto$ *addr*)))) = *hf*
and *x9*:*CT*, *h* $_w\vdash$ *Env*
and *x10*:*CT*, *sEnv* $\vdash$ *FieldUpdate e1 f e2* : *T*
and *x11*:*CT*, *h* $\vdash$ *Env* : *sEnv* **.**
— After the assumptions are applied to the two bigsteps to use the resulting information, the type of obaddr can be shown in the resulting heap hf.
from *obaddr-T1 wte-h1 wte-h2 x1 x3 x6 x8 x5* **have** *hf-obaddr-T1*:*CT*, *hf*, *the* (*Env this*) $\vdash$ *obaddr* : *sObject u-Te C-Te*
— Now that information can be used to easily show the type of addr,
from *x6 wte-h2 x8 h2-addr-T hext-hf* **have** *hf-addr-T-x*:*CT*, *hf*, *the* (*Env this*) $\vdash$ *addr* : *u-Te* $\triangleright^t$ *Ti*
— which will result in the type preservation
from *T-unfold hf-addr-T-x wte-h2 x8 x10 hf-addr-T* **have** *hf-addr-T2*:*CT*, *hf*, *the* (*Env this*) $\vdash$ *addr* : *T*
— by the use of some transformations.

— As the object at address obaddr is updated, the last step to do is to show that it is a welltyped address
from *hext-hf T-unfold ob-not-any hf-ob-addr-Ti hf-obaddr-T1 wta-h2-obaddr wte-h2 x8 x6* **have** *wta-hf-obaddr*:*CT*, *hf* $_a\vdash$ *obaddr*
— as that information is required to show the resulting environment welltyped:
from *wta-hf-obaddr wte-h2 x8 x5 hext ob-not-any* **have** *wte-hf*:*CT*, *hf* $_w\vdash$ *Env* **next**
— Besides the expressions for arrays, the last one and to prove the most difficult one is the method invocation.

case (*os-invk CT Env addr addrs e e1 elist fs h h1 h2 hf m mDef newEnv paddr rt vDefs xs T sEnv*) **thus** *?case* **proof** −
— Using the induction rule will result in a lot of new assumptions
have *x1*:$CT \vdash h, Env, e \rightsquigarrow h1, paddr$
and *x2*:$\bigwedge sEnv\ T.\ [\![CT, h\ {}_w\!\vdash Env; CT, sEnv \vdash e : T; CT, h \vdash Env : sEnv]\!] \Longrightarrow CT, h1\ {}_w\!\vdash Env \wedge CT, h1, the\ (Env\ this) \vdash paddr : T$
and *x3*:$CT \vdash h1, Env, elist \rightsquigarrow^+ h2, addrs$
and *x4*:$\bigwedge sEnv\ Ts.\ [\![CT, h1\ {}_w\!\vdash Env; length\ elist = length\ Ts; length\ Ts = length\ addrs; CT, sEnv \vdash^+ elist : Ts; CT, h1 \vdash Env : sEnv]\!] \Longrightarrow CT, h2\ {}_w\!\vdash Env \wedge CT, h2, the\ (Env\ this) \vdash^+ addrs : Ts$
and *x5*:$paddr \neq NullP$
and *x6*:$mdata(CT, aClass\ h2\ paddr, m) = Some(mDef)$
and *x6a*:$mBody\ mDef = e1$
and *x6b*:$mParams\ mDef = vDefs$
and *x6c*:$varDefs\text{-}names\ vDefs = xs$
and *x7*:$[xs\ [\mapsto]\ addrs, this \mapsto paddr] = newEnv$
and *x8*:$CT \vdash h2, newEnv, e1 \rightsquigarrow hf, addr$
and *x9*:$\bigwedge sEnv\ T.\ [\![CT, h2\ {}_w\!\vdash newEnv; CT, sEnv \vdash e1 : T; CT, h2 \vdash newEnv : sEnv]\!] \Longrightarrow CT, hf\ {}_w\!\vdash newEnv \wedge CT, hf, the\ (newEnv\ this) \vdash addr : T$
and *x10*:$CT, h\ {}_w\!\vdash Env$
and *x11*:$CT, sEnv \vdash MethodInvk\ e\ m\ elist : T$
and *x12*:$CT, h \vdash Env : sEnv$
and *x13*:$h2\ paddr = Some\ (oObject\ rt\ fs)$ **.** — which have to be unfolded partially:
from *Temp-unfold* **obtain** *mDefa u C w Tsu Tu Ts-s* **where** *T-unfoldu*:$(CT, sEnv \vdash e : sObject\ u\ C) \wedge mdata(CT, C, m) = Some(mDefa) \wedge w = mPure\ mDefa \wedge Tsu = varDefs\text{-}types\ (mParams\ mDefa) \wedge Tu = mReturn\ mDefa \wedge (CT, sEnv \vdash^+ elist : Ts\text{-}s) \wedge (CT \vdash (u \rhd^t Tu) <: T) \wedge (map\ (\lambda tsd.\ u \rhd^t tsd)\ Tsu = Ts\text{-}s) \wedge (\forall\ t \in (set\ Ts\text{-}s).notUnknown(t))$
— Besides some trivial proofs, the first information that has to be shown is the welltyping of the created environment,
from *mapSomeD ftypesOK-SomeD wte-h2 x7 x5 x3 x13 h2-addrs-Ts h1-paddr-T-e* **have** *wte-h2-new*:$CT, h2\ {}_w\!\vdash newEnv$ — and the type of the address paddr:
from *x13 wte-h1 h1-paddr-T-e x3* **have** *h2-paddr-T-e*:$CT, h2, the\ (Env\ this) \vdash paddr : sObject\ u\ C$
— Using the unfolding and the adaption lemmas it is a non trivial step to prove that the created runtime environment fits to the environment required for the method call:
from *x13 distinct-xs newsEnv-def3 wte-h2-new wte-h2 h2-addrs-Ts x7 h2-paddr-T-e T-unfold x5 Tsu-Tss length-addrs-Ts length-elist-addrs length-xs-addrs* **have** *newsEnv-def*:$CT, h2 \vdash newEnv : newsEnv$
— As a result the static type of the methods expression can be shown
from *distinct-xs newsEnv-def x6 x6a x6b x6c T-unfold h2-paddr-T-e wte-h2 x13 Tsu-Tss newsEnv-def3 x5* **have** *e1-Tu*:$CT, newsEnv \vdash e1 : Tu$
— which will be used to show the type of the resulting address:
from *hf-paddr-T-e wte-h1 hf-addr-T-new x5 x7 T-unfold x3 x8* **have** *hf-addr-T*:$CT, hf, the\ (Env\ this) \vdash addr : T$
— At last with all those information it is not too difficult to prove the resulting environment welltyped,
from *wte-hf-new wte-h2 x8* **have** *wte-hf*:$CT, hf\ {}_w\!\vdash Env$ — as that could be used to show the whole case
from *wte-hf hf-addr-T* **show** *?case* **by** *simp*
qed
next
— Now, as all the expression types for normal objects have been shown, the remaining three cases introduced for arrays have to be proven typesafe.

To prove a newly created array typesafe can be done in the same way as a new normal object:
case (*os-newarray CT Env Tx addr h hf mType myArray myself myselfo n u Tarray sEnv*) **thus** *?case* **proof**

– — Using the assumptions generated by the induction it is not too complicated to show
from *wte-hf res2* **show** *CT*, *hf* $_w\vdash$ *Env* $\wedge$ *CT*, *hf*, *the* (*Env this*) $\vdash$ *addr* : *Tarray* **by** *simp*
qed
next
— As the new alternative definition of arrays does not require a viewpoint adaption in contrast to the old declaration, there is no adaption step required, which makes the proofs much simpler.
case (*os-arrayread CT Env Ta addr addr-e e eField h hf n T sEnv*) **thus** *?case* **proof** –
— Using the assumptions resulting from the induction it is not too tricky to show the case typesafe:
from *wte-hf addr-Ti x7-TiTa x7-Tithis* **show** *CT*, *hf* $_w\vdash$ *Env* $\wedge$ *CT*, *hf*, *the* (*Env this*) $\vdash$ *addr* : *T*
next
— At last the update of an array has to be shown. Because a runtime typecheck is done in the bigstep semantics, it is not too complicated to prove that again:
case (*os-arrayupd CT Env Fields addr e1 e2 h h1 h2 hf n obaddr own rT T sEnv*) **thus** *?case* **proof** –
— Using the assumptions it takes only a few steps to show the intermediate heap is welltyped and the type of the address is typesafe.
from *h1-env wte-h1 e2-T x4* **have** *wte-h2*:*CT*, *h2* $_w\vdash$ *Env* **and** *addr-Ti*:*CT*, *h2*, *the* (*Env this*) $\vdash$ *addr* : *Ti* **by** *auto*

— Using that information it is easy to show the resulting heap welltyped,
from *hf-wta-obaddr x6 x8 wte-h2* **have** *wte-hf*:*CT*, *hf* $_w\vdash$ *Env*
— and the resulting address preserving the type:
from *hf-addr-Ti Ti-T wte-hf* **have** *hf-addr-T*: *CT*, *hf*, *the* (*Env this*) $\vdash$ *addr* : *T*
qed

Now, having the type preservation proven with this strange looking lemma, it can be transformed into a nicer representation:

lemma shows *bigstep-well-typing*:
⟦*CT* $\vdash$ *h*, *rEnv*, *e* $\rightsquigarrow$ *hf*, *addr*;
CT, *h* $_w\vdash$ *rEnv*;
CT, *sEnv* $\vdash$ *e* : *T*;
CT, *h* $\vdash$ *rEnv* : *sEnv*⟧
$\Longrightarrow$ *CT*, *hf* $_w\vdash$ *rEnv* $\wedge$
CT, *hf*, *the* (*rEnv this*) $\vdash$ *addr* : *T*
by (*simp add*: *bigstep-well-typingX*)

lemma shows *bigsteps-well-typing*:
⟦*CT* $\vdash$ *h*, *rEnv*, *es* $\rightsquigarrow+$ *h2*, *addrs*;
CT, *h* $_w\vdash$ *rEnv*;
CT, *sEnv* $\vdash^+$ *es* : *Ts*;
CT, *h* $\vdash$ *rEnv* : *sEnv*⟧
$\Longrightarrow$ *CT*, *h2* $_w\vdash$ *rEnv* $\wedge$
CT, *h2*, *the* (*rEnv this*) $\vdash^+$ *addrs* : *Ts*
by (*simp add*:*bigsteps-well-typingX*)

end

theory *enctyping* **imports** *statichelper exptyping*

begin

4.2.13 Encapsulating typing rules

Two sorts of typing rules are required to encapsulate expressions: one sort for pure expressions and one for encapsulated expressions will be defined.

The goal of the pure typing rules is to limit the expressions to such ones that do not modify the heap.

This can be done by forbidding field and array updates and only by allowing to call pure methods. nonpure methods are not allowed to call.

consts
puretyping :: $(classTable * staticEnv * exp)\ set$
puretypings :: $(classTable * staticEnv * exp\ list)\ set$
syntax
-ptyping :: $[classTable, staticEnv, exp\ list] \Rightarrow bool\ (-, -\ {}_p\vdash\ -\ [80,80,80]\ 80)$
-ptypings :: $[classTable, staticEnv, exp\ list] \Rightarrow bool\ (-, -\ {}_p\vdash^+\ -\ [80,80,80]\ 80)$
translations
$CT, \Gamma\ {}_p\vdash\ e \rightleftharpoons (CT, \Gamma, e) \in puretyping$
$CT, \Gamma\ {}_p\vdash^+\ es \rightleftharpoons (CT, \Gamma, es) \in puretypings$

inductive *puretypings puretyping*
intros
ts-nil : $CT,\Gamma\ {}_p\vdash^+\ []$

ts-cons :
$[\![\ CT, \Gamma\ {}_p\vdash\ e0;\ CT, \Gamma\ {}_p\vdash^+\ es\ ]\!]$
$\Longrightarrow CT, \Gamma\ {}_p\vdash^+\ (e0\ \#\ es)$

t-var :
$CT, \Gamma\ {}_p\vdash\ Var\ x$
t-null :
$CT, \Gamma\ {}_p\vdash\ Null$

t-fieldProj :
$[\![\ CT, \Gamma\ {}_p\vdash\ e0]\!]$
$\Longrightarrow CT, \Gamma\ {}_p\vdash\ FieldProj\ e0\ f$

t-invk :
$[\![\ CT, \Gamma \vdash e0 : (sObject\ u\ C);$
$mdata(CT, C, m)=Some(mDef);$
$mPure\ mDef=Pure;$
$CT, \Gamma\ {}_p\vdash\ e0;$
$CT, \Gamma\ {}_p\vdash^+\ es]\!]$
$\Longrightarrow CT, \Gamma\ {}_p\vdash\ MethodInvk\ e0\ m\ es$

t-new :
$CT, \Gamma\ {}_p\vdash\ New\ u\ C$
t-newarray:$CT, \Gamma\ {}_p\vdash\ NewArray\ u\ T\ n$
t-cast :
$[\![\ CT, \Gamma\ {}_p\vdash\ e]\!]$
$\Longrightarrow CT, \Gamma\ {}_p\vdash\ Cast\ T\ e$

t-arrayread:
$[\![\ CT, \Gamma\ {}_p\vdash\ e0]\!]$
$\Longrightarrow CT, \Gamma\ {}_p\vdash\ ArrayProj\ e0\ n$

In comparison to the pure definition, an encapsulated expression allows to update fields and arrays. This update is only allowed if the universe type of the receiver object is either Peer or Rep.

consts
enctyping :: (*classTable* ∗ *staticEnv* ∗ *exp*) *set*
enctypings :: (*classTable* ∗ *staticEnv* ∗ *exp list*) *set*
syntax
-etyping :: [*classTable*, *staticEnv*, *exp list*] ⇒ *bool* (-, - $_e$⊢ - [*80*,*80*,*80*] *80*)
-etypings :: [*classTable*, *staticEnv*, *exp list*] ⇒ *bool* (-, - $_e$⊢$^+$ - [*80*,*80*,*80*] *80*)
translations
CT, Γ $_e$⊢ *e* ⇌ (*CT*, Γ, *e*) ∈ *enctyping*
CT, Γ $_e$⊢$^+$ *es* ⇌ (*CT*, Γ, *es*) ∈ *enctypings*

inductive *enctypings enctyping*
intros
e-ts-nil : *CT*, Γ $_e$⊢$^+$ []

e-ts-cons :
⟦ *CT*, Γ $_e$⊢ *e0*; *CT*, Γ $_e$⊢$^+$ *es* ⟧
⟹ *CT*, Γ $_e$⊢$^+$ (*e0* # *es*)

e-t-var :
CT, Γ $_e$⊢ *Var x*
e-t-null :
CT, Γ $_e$⊢ *Null*

e-t-fieldProj :
⟦ *CT*, Γ $_e$⊢ *e0*⟧
⟹ *CT*, Γ $_e$⊢ *FieldProj e0 f*

e-t-fieldUpdate :
⟦ *CT*, Γ $_e$⊢ *e0*;
CT, Γ ⊢ *e0* : *sObject u C*;
u=*Peer* ∨ *u*=*Rep*;
CT, Γ $_e$⊢ *e1*⟧
⟹ *CT*, Γ $_e$⊢ *FieldUpdate e0 f e1*

e-t-invk :
⟦ *CT*, Γ ⊢ *e0* : (*sObject u C*);
mdata(*CT*, *C*, *m*)=*Some*(*mDef*);
w=*mPure mDef*;
w=*Pure* ∨ *u*=*Peer* ∨ *u*=*Rep*;
CT, Γ $_e$⊢ *e0*;
CT, Γ $_e$⊢$^+$ *es*⟧
⟹ *CT*, Γ $_e$⊢ *MethodInvk e0 m es*

e-t-new :
CT, Γ $_e$⊢ *New u C*
e-t-newarray:
CT, Γ $_e$⊢ *NewArray u T n*
e-t-cast :
⟦ *CT*, Γ $_e$⊢ *e*⟧
⟹ *CT*, Γ $_e$⊢ *Cast T e*
e-t-arrayread:
⟦ *CT*, Γ $_e$⊢ *e0*⟧

$\Longrightarrow CT, \Gamma\ {}_e\vdash ArrayProj\ e0\ n$
e-t-arrayupdate :
$[\![\ CT, \Gamma\ {}_e\vdash e0;$
$CT, \Gamma \vdash e0 : sArray\ u\ T;$
$u{=}Peer \vee u{=}Rep;$
$CT, \Gamma\ {}_e\vdash e1]\!]$
$\Longrightarrow CT, \Gamma\ {}_e\vdash ArrayUpdate\ e0\ n\ e1$

As purity of a method is defined in the classtable, there have to be some definitions what an encapsulated method is:

consts *method-typing-enc* :: $(classTable * methodDef * className)\ set$
syntax
-method-typing-enc :: $[classTable, className, methodDef] \Rightarrow bool$ (-, - ${}_m\vdash_e$ - [80,80,80] 80)
translations
$CT, C\ {}_m\vdash_e\ md \rightleftharpoons (CT, md, C) \in method\text{-}typing\text{-}enc$

inductive *method-typing-enc*
intros
m-typing-enc:
$[\![\ CT(C) = Some(CDef);$
$cName\ CDef = C;$
$mBody\ mDef = e;$
$mPure\ mDef = w;$
$mParams\ mDef{=}vDefs;$
$w{=}NonPure \vee\ (\forall\ vDef \in set\ vDefs\ .\ (utOM\ (vdType\ vDef){=}Any));$
$varDefs\text{-}types\ vDefs = vDefsT;$
$varDefs\text{-}names\ vDefs = vDefsN;$
$\Gamma = (map\text{-}upds\ empty\ vDefsN\ vDefsT)(this \mapsto (sObject\ This\ C));$
$w{=}Pure \longrightarrow CT, \Gamma\ {}_p\vdash e;$
$w{=}NonPure \longrightarrow CT, \Gamma\ {}_e\vdash e]\!]$
$\Longrightarrow CT, C\ {}_m\vdash_e mDef$

Also the encapsulation of a class has to be specified.

consts *class-typing-enc* :: $(classTable * classDef)\ set$
syntax
-class-typing-enc :: $[classTable, classDef] \Rightarrow bool$ (- ${}_e\vdash$ - [80,80] 80)
translations
$CT\ {}_e\vdash CDef \rightleftharpoons (CT, CDef) \in class\text{-}typing\text{-}enc$
inductive *class-typing-enc*
intros
t-class:
$[\![\ cName\ cDef = C;$
$cSuper\ cDef = D;$
$cMethods\ cDef = mDefs;$
$cFields\ cDef = fDefs;$
$\forall\ m\ mDefD.\ (mdata(CT, D, m){=}Some(mDefD) \longrightarrow (\exists\ mDefC.\ (mdata(CT, C, m){=}Some(mDefC) \wedge$
$CT, C\ {}_m\vdash_e mDefC)));$
$\forall\ m \in (set\ mDefs).\ CT, C\ {}_m\vdash_e m]\!]$
$\Longrightarrow CT\ {}_e\vdash cDef$

At last the encapsulation of a classtable will be defined.

consts *ct-typing-enc* :: *classTable set*
syntax
-ct-typing-enc :: $classTable \Rightarrow bool$ (${}_e\vdash$ - 80)
translations

$_e\vdash CT \rightleftharpoons CT \in$ *ct-typing-enc*
inductive *ct-typing-enc*
intros
ct-all-ok:
$[\![\forall C\ CDef.\ CT(C) = Some(CDef) \longrightarrow (CT\ {}_e\vdash CDef)\ ]\!]$
$\Longrightarrow {}_e\vdash CT$

end

theory *ownerasmodifier* **imports** *typesafety enctyping* **begin**

4.2.14 Owner as modifier property

Like already seen in the proof to show the typesystem typesafe, there has to be used an alternative view on the lemmas.

Before proing the owner as modifier property, it has to be shown that pure expressions do not modify existing objects in the heap.

lemma shows *bigstep-pureX*:
$((CT \vdash h, rEnv, e \rightsquigarrow h3, addr) \Longrightarrow (\bigwedge sEnv.$
$[\![CT, sEnv\ {}_p\vdash e;$
$_e\vdash CT;$
$CT, h\ {}_w\vdash rEnv;$
$CT, h \vdash rEnv : sEnv;$
$\exists T.\ CT, sEnv \vdash e{:}T]\!]$
$\Longrightarrow (\forall a\ x.\ h\ a{=}Some(x) \longrightarrow (h\ a = h3\ a))))$
and *bigsteps-pureX*:
$(CT \vdash h, rEnv, es \rightsquigarrow{+} h2, addrs \Longrightarrow (\bigwedge sEnv.$
$[\![\forall e \in set\ es.\ (CT, sEnv\ {}_p\vdash e \wedge (\exists T.\ CT, sEnv \vdash e : T));$
$_e\vdash CT;$
$CT, h\ {}_w\vdash rEnv;$
$CT{,}h \vdash rEnv : sEnv]\!]$
$\Longrightarrow (\forall a\ x.\ h\ a{=}Some(x) \longrightarrow (h\ a{=}h2\ a))))$
— The proof will run by induction over the bigstep semantics.
proof (*induct rule*:*bigstep-bigsteps-induct*)
— Most of the cases are very simple to prove. Only to show the purity for pure methods that are called is not as simple as one might think.
case (*os-invk CT Env addr addrs e e1 elist fs h h1 h2 hf m mDef newEnv paddr rt vDefs xs sEnv*) **thus** *?case*
proof −
— Using the induction will result in a lot of assumptions:
have *x1*:$CT \vdash h, Env, e \rightsquigarrow h1, paddr$
and *x2*:$\bigwedge sEnv.\ [\![CT, sEnv\ {}_p\vdash e;\ {}_e\vdash CT;\ CT, h\ {}_w\vdash Env;\ CT, h \vdash Env : sEnv;\ \exists T.\ CT, sEnv \vdash e : T]\!]$
$\Longrightarrow \forall a\ x.\ h\ a = Some\ x \longrightarrow h\ a = h1\ a$
and *x3*:$CT \vdash h1, Env, elist \rightsquigarrow{+} h2, addrs$
and *x4*:$\bigwedge sEnv.\ [\![\forall e{\in}set\ elist.\ (CT, sEnv\ {}_p\vdash e \wedge (\exists T.\ (CT, sEnv \vdash e : T)))\ ;\ {}_e\vdash CT;\ CT, h1\ {}_w\vdash Env;$
$CT, h1 \vdash Env : sEnv]\!]$
$\Longrightarrow \forall a\ x.\ h1\ a = Some\ x \longrightarrow h1\ a = h2\ a$
and *x5*:$paddr \neq NullP$
and *x6*:$mdata(CT, aClass\ h2\ paddr, m){=}Some\ mDef$
and *x6a*:$mBody\ mDef{=}e1$
and *x6b*:$mParams\ mDef{=}vDefs$
and *x6c*:$varDefs\text{-}names\ vDefs\ {=}xs$
and *x7*:$[xs\ [\mapsto]\ addrs, this \mapsto paddr] = newEnv$

and *x8*:$CT \vdash h2, newEnv, e1 \rightsquigarrow hf, addr$
and *x9*:$\bigwedge sEnv.\ [\![CT, sEnv\ {}_p\vdash e1;\ {}_e\vdash CT;\ CT, h2\ {}_w\vdash newEnv;\ CT, h2 \vdash newEnv : sEnv;\ \exists\ T.\ CT, sEnv \vdash e1 : T]\!]$
$\Longrightarrow \forall\ a\ x.\ h2\ a = Some\ x \longrightarrow h2\ a = hf\ a$
and *x10*:$CT, sEnv\ {}_p\vdash MethodInvk\ e\ m\ elist$
and *x11*:${}_e\vdash CT$
and *x12*:$CT, h\ {}_w\vdash Env$
and *x13*:$CT, h \vdash Env : sEnv$
and *x14*:$\exists\ T.\ CT, sEnv \vdash MethodInvk\ e\ m\ elist : T$
and *x15*:$h2\ paddr{=}Some(oObject\ rt\ fs)$ **.**
— Using these assumptions it is simple to show the purity property for the bigstep from h to h1
from *x2 x10-unfold x11 x12 x13* **have** *x2-res*:$\forall\ a\ x.\ h\ a = Some\ x \longrightarrow h\ a = h1\ a$ **by** *blast*
— and from h1 to h2
from *x4 x11 h1-paddr-T-e elist-enc h1-sEnv* **have** *x4-res*:$\forall\ a\ x.\ h1\ a = Some\ x \longrightarrow h1\ a = h2\ a$ **by** *auto*
— With the help of unfolding rules and some proofs to show that the created runtime environments fits the static environment, it can be shown that the objects in h2 stay the same in hf:
from *mdefencopen x9 x11 h2-wte-newenv newEnvtonewsEnv mdefamdef mdefUnfold* **have** *x9-res*:$\forall\ a\ x.\ h2\ a = Some\ x \longrightarrow h2\ a = hf\ a$
— Those three goals will lead to the final information that a method call does not modify the heap.
from *x2-res x4-res x9-res* **have** *x-res*:$\forall\ a\ x.\ h\ a{=}Some\ x \longrightarrow h\ a = hf\ a$ **qed**

To give a more readable view of the lemma a transformation will be done:

lemma shows *bigstep-pure*:
$[\![CT \vdash h, rEnv, e \rightsquigarrow hf, addr;$
$CT, sEnv\ {}_p\vdash e;$
${}_e\vdash CT;$
$CT, h\ {}_w\vdash rEnv;$
$CT, h \vdash rEnv : sEnv;$
$\exists\ T.\ CT, sEnv \vdash e : T]\!]$
$\Longrightarrow \forall\ a\ x.\ (h\ a{=}Some(x) \longrightarrow h\ a{=}hf\ a)$
by (*rule bigstep-pureX*, *simp-all*)

lemma shows *bisteps-pure*:
$[\![CT \vdash h,rEnv,es \rightsquigarrow+ hf,addrs;$
$\forall\ e \in set\ es.\ (CT, sEnv\ {}_p\vdash e \wedge (\exists\ T.\ CT,sEnv \vdash e : T));$
${}_e\vdash CT;$
$CT, h\ {}_w\vdash rEnv;$
$CT, h \vdash rEnv : sEnv]\!]$
$\Longrightarrow \forall\ a\ x.\ (h\ a{=}Some(x) \longrightarrow h\ a{=}hf\ a)$
by (*rule bigsteps-pureX*, *simp-all*)

Having that information about purity, the owner-as-modifier property can be shown.

lemma shows *bigstep-owner-as-modifierX*:
$((CT \vdash h, rEnv, e \rightsquigarrow h3, addr) \Longrightarrow (\bigwedge\ sEnv.$
$[\![CT, sEnv\ {}_e\vdash e;$
${}_e\vdash CT;$
$CT, h\ {}_w\vdash rEnv;$
$CT, h \vdash rEnv : sEnv;$
$\exists\ T.\ CT, sEnv \vdash e{:}T]\!]$
$\Longrightarrow ((\forall\ a\ x.\ h\ a{=}Some(x) \longrightarrow (h\ a{=}h3\ a) \vee (owner\text{-}is\text{-}this\ h\ rEnv\ a)))))$
and *bigsteps-owner-as-modifierX*:
$(CT \vdash h, rEnv, es \rightsquigarrow+ h2, addrs \Longrightarrow (\bigwedge\ sEnv.$
$[\![\forall\ e \in set\ es.\ (CT, sEnv\ {}_e\vdash e \wedge (\exists\ T.\ CT,sEnv \vdash e : T));$
${}_e\vdash CT;$
$CT, h\ {}_w\vdash rEnv;$
$CT, h \vdash rEnv : sEnv]\!]$

$\Longrightarrow$ $((\forall\ a\ x.\ h\ a{=}Some(x) \longrightarrow (h\ a{=}h2\ a \vee (owner\text{-}is\text{-}this\ \ h\ rEnv\ a))))))$
— As all the proofs before, it will run with an induction over the bigstep semantics.
proof (*induct rule*:*bigstep-bigsteps-induct*)
— To show the case
case (*os-upd CT Env Fields addr e1 e2 f h h1 h2 hf oa obaddr sEnv*) **thus** *?case*
— is not too complicated, and will be left out here.
— The case for the method call can be done in nearly the same way as for the purity proof.
case (*os-invk CT Env addr addrs e e1 elist fs h h1 h2 hf m mDef newEnv paddr rt vDefs xs sEnv*) **thus** *?case*
proof −
— The main difference is that there has to be a distinction between either being owned by the current viewpoint or keeping the heap untouched.
from *x2 x10-unfold x11 x12 x13* **have** *x2-res*:$\forall\ a\ x.\ h\ a = Some\ x \longrightarrow h\ a = h1\ a \vee owner\text{-}is\text{-}this\ h\ Env\ a$ **by** *blast* — At last there has to be done a distinction between a nonpure method
from *mdefencopenpure mdefencopennpure x9 x11 h2-wte-newenv newEnvtonewsEnv mdefUnfold x8* **have** *x9-resnp*:$mPure\ mDef = NonPure \Longrightarrow \forall\ a\ x.\ h2\ a = Some\ x \longrightarrow h2\ a = hf\ a \vee owner\text{-}is\text{-}this\ h2\ newEnv\ a$
— and a pure method:
from *mdefencopenpure mdefencopennpure x9 x11 h2-wte-newenv newEnvtonewsEnv mdefUnfold x8* **have** *x9-resp*:$mPure\ mDef = Pure \Longrightarrow \forall\ a\ x.\ h2\ a = Some\ x \longrightarrow h2\ a = hf\ a$
— To show that an update of an array element does not destroy the owner as modifier property
case (*os-arrayupd CT Env Fields addr e1 e2 h h1 h2 hf n obaddr own rT sEnv*) **thus** *?case* **proof** −
— the most difficult part is to show that the array is owned by the current viewpoint:
from *wte-h1 wte-h2 x3 x5 uX-def x6* **have** *owner-is-this*:*owner-is-this h2 Env obaddr*
qed

At last a more readable presentation of the owner as modifier property will be proven.

lemma shows *bigstep-owner-as-modifier*:
$[\![CT \vdash h,\ rEnv,\ e \rightsquigarrow hf,\ addr;$
$CT,\ sEnv\ {}_e\vdash e;$
${}_e\vdash CT;$
$CT,\ h\ {}_w\vdash rEnv;$
$CT,\ h \vdash rEnv : sEnv;$
$\exists\ T.\ CT,\ sEnv \vdash e{:}T]\!]$
$\Longrightarrow (\forall\ a\ x.\ h(a){=}Some(x) \longrightarrow (h\ a{=}hf\ a \vee ((a,\ owner\ h\ (the\ (rEnv\ this))) \in (sowner\ h)\hat{}+)))$
apply (*drule bigstep-owner-as-modifierX*,*simp-all add*:*owner-is-this-def*)
apply *clarify*
apply (*erule-tac x*=*a* **in** *allE*,*erule-tac x*=*x* **in** *allE*,*clarsimp*)
done

lemma shows *bisteps-owner-as-modifier*:
$[\![CT \vdash h,\ rEnv,\ es \rightsquigarrow+\ hf,\ addrs;$
$\forall\ e \in set\ es.\ (CT,\ sEnv\ {}_e\vdash e \wedge (\exists\ T.\ CT,sEnv \vdash e : T));$
${}_e\vdash CT;$
$CT,\ h\ {}_w\vdash rEnv;$
$CT,\ h \vdash rEnv : sEnv]\!]$
$\Longrightarrow ((\forall\ a\ x.\ h\ a{=}Some(x) \longrightarrow (h\ a{=}hf\ a \vee ((a,\ owner\ h\ (the\ (rEnv\ this))) \in (sowner\ h)\hat{}+))))$
apply (*drule bigsteps-owner-as-modifierX*,*simp-all add*:*owner-is-this-def*)
apply *clarify*
apply (*erule-tac x*=*a* **in** *allE*,*erule-tac x*=*x* **in** *allE*,*clarsimp*)
done
end

4.3 Problems and the resolution

During the formalization using Isabelle/HOL, there have been different problems. To make life more easy for someone extending the formalization, and to understand why some things have been done in a specific way, the following aspects should show the biggest problems and how they have been resolved:

- Subsumption rule: As the subsumption rule in typing definition gives you more than one possible type for most of the expressions , the automatically generated unfolding rules in Isabelle/HOL could not be used. Instead there had to be defined an unfolding lemma, which was a little bit tricky to prove. The unfolding lemma is used to unfold a typing expression to get the different parts of that typing expression form the definition of the typing. For example in the case of an environment read "$CT, \Gamma^s \vdash Var\ v : T$" will result in a lookup for the type of v in the static environment "$\Gamma^s(v) = T'$" and a subtyping relation from T' to T "$CT \vdash T' <: T$".

- Induction over the semantics: To apply the induction rules that are generated by the inductive definition of the bigstep semantics, was complicated. The lemmas had to be written down in a special way to apply the induction on them. With the result, a better looking lemma can be shown easily.

- Unification of variables: The lemmas about type safety and the owner as modifier property have quite a lot of variables. Together with the induction rule the quantity exceeded the default limitations for the unification. The internal limits had to be raised to a higher number. Afterwards the unification did work.

- Type dynamization: With the differentiation between topological and encapsulation typesystem, and the possibility of creating objects with the any_u modifier, to be able to do a runtime subtyping there has to be differentiated between any_a and the $anyowner_a$. Without that differentiation there have been cases where it was not possible to differentiate between $peer$ and any_u during the proofs of viewpoint adaptation.

5 Conclusion

The goal of this diploma thesis was to formalize the Universe Type System with the help of Isabelle/HOL. That was done successfully.

5.1 Results

Looking at the formalization of the Universe Type System the following conclusions can be drawn:

- It is possible to split up the type system into two parts: a topological one and an encapsulation one. With the topological type system the type safety can be shown and most parts of the formalization are done. The encapsulation limitations are required only to limit the type system in a way, to be able to show the owner-as-modifier property.
- The formalization confirms the proofs done by pen and paper. There has been found only a small imprecision during the definition of the runtime subtyping rules, which has been resolved by the differentiation of any_a and $anyowner_a$.
- The new specification of arrays, and how they are handled, has been shown sound with respect to the typesystem.

At last some figures about the amount of the formalization:

- The formalization consists of about 6700 lines.
- 180 lemmas have been proven.
- The whole formalization takes about eight and a half minutes to proof on a macbook.

5.2 Future work

This formalization can be a good basis to settle other parts on top of it or extend it:

- One of the interesting things would be the extension with generic universe types. As generics are one of the current research areas surrounding universes, a computer proved formalization of the type system would be quite helpful. A useful preparation for generics was the implementation of arrays.
- As the formalization was done with a very small subset of Java, it would be interesting to extend that subset. As there already exists a formalization of a huge part of Java called Jinja, a good starting point would be merging both formalizations.
- At last there are some properties not formalized in this paper. Extending it with a small step semantics and a proof of progress would finalize the type safety proof. Besides the proof of progress, using the encapsulation type system it would be great to formalize the framing conditions that result from the type system.

A Appendix: Viewpoint adaptation of the ownership modifier

Because the ownership modifier depends on the viewpoint, when changing the viewpoint the ownership modifier of a variable has to be adapted to the new viewpoint.

There are alternative possibilities to do the viewpoint adaptation.
Two different ways are used in the paper about generic universe types and the paper about universe java. To be sure both of them do the same, they are compared in the next section. As result a new infix function ▷ will be defined, combining the two functions into one single function using the OM unknown.

A.1 Existing Functions in GUT

The ▷ Function It takes two arguments u and u' with u' the ownership part of a field in the viewpoint of a variable with the ownership part u.

The ▷ function has the signature OM x OM -> OM
u ▷ u' = u"

u ▷ u'	*peer*	*rep*	any_u
$this_u$	*peer*	*rep*	any_u
peer	*peer*	any_u	any_u
rep	*rep*	any_u	any_u
any_u	any_u	any_u	any_u

Note: On the right side of the ▷ function there can not be $this_u$ because $this_u$ is not allowed in field or variable definitions.

The ▷ function is used in the following inference rules and does the following things there:

1. WFM-1
   Check if a pure method has only arguments of type any_u.
2. WFT-2
   Used for the type check of the generic types , so not needed for a type system without generics
3. GT-READ
   Adapt the OM of field to the local view o.f => o ▷ f
4. GT-Upd
   Adapt type of field to local view o.f => o ▷ f besides this the right handed expression has to have the OM of f adapted to the local viewpoint.
5. GT-Invk
   Adapt argument OM to local view and check if they fit to the expression result types.
   Adapt return value OM to local view.

.
.
.

The RP function It takes two arguments u and u' with u' the ownership part of a field in the viewpoint of a variable with the ownership part u.

The rp function has the signature OM x OM -> bool
rp(u,u')=bool

rp(o,o')	*peer*	*rep*	any_u
$this_u$	true	true	true
peer	true	false	true
rep	true	false	true
any_u	true	false	true

Note: The real rp function is a little bit different and accepts a list of types as input.
Note: the real rp functions checks for every type in the list if it evaluates to true.

The rp() function is used in:

1. GT-Upd
   Check if the OM of the variable has the correct OM in the view of the receiving variable e.g. o1.f=o2 here f has a OM depending on the view of o1. What has to be done is to transform o2 into the view of o1 and check if it is a subtype of f's type.

2. GT-Invk
   Check if the arguments of the method m have the correct OM in the view of the receiving Variable o1.
   E.g. o1.m(o2) here o1 has a method m, where its first argument has an ownership modifier depending on the view of o1. What has to be done is to transform o2 into the view of o1 and check if it is a subtype of the type of the parameter.

A.2 UJ

The lookup tables for the $\oplus$ and $\ominus$ operators can be found in the UJ paper.

What does the $\oplus$ operator?

1. Field e.f
   Adapt the type of f to the local view.

2. Call e.m(x)
   Adapt the return value of m to the local view

3. tfield e.f
   see "Field e.f"

4. tcall e.m(x)
   see "Call e.m(x)"

5. tframe
   Adapt the return value when returning from a call (not needed with big step semantics so far I can see).

What does the $\ominus$ operator

1. Assign o1.f=o2
   Adapt the type of o2 to the viewpoint of o1 and check if it is a subtype and return this value. Besides this, the return value has the type of o2.
2. Call o1.m(o2)
   note: o1 has a method m with an argument of type t Adapt the OM of o2 to the viewpoint of o1. Then it is checked if the adapted type is a subtype of the type t.
3. tassign
   see "Assign"
4. tcall
   see "Call"

A.3 Comparison of the methods in both papers

There are five positions where the OM of a variable has to be adapted because of the change of viewpoint:
a) Field Read
b) Return Values
c) Field Updates
d) Method arguments of non-pure functions
e) Method arguments of pure functions
The field read and the return value can be analyzed together, because both of them adapt the type of a field or variable to the current viewpoint.

The field updates and the method arguments of non-pure methods can be analyzed together, too, because both of them adapt the type of a field or variable in the current viewpoint to the viewpoint of the Receiving Variable.

A.3.1 Cases a and b

In both cases there is a variable o1 and a field f or a return value f which has to be adapted to the current viewpoint. o1 has the OM u and f has the OM u'.
In the GUT paper the $\triangleright$ function is used with u $\triangleright$ u' to adapt the viewpoint in the cases a and b.
In the UJ paper the $\oplus$ operator is used with u $\oplus$ u' to adapt the viewpoint in the cases a and b.
By comparing the result table of both of them it is easy to see that both of them return the same values when called with the same values u and u'. So it is easy to understand that both of them do the same and are used in the cases a and b in the same way.

A.3.2 Cases c and d

In both cases there is a variable o1 a field or argument f and a variable o2 which is assigned to f.
o1 has the OM u, f has the OM u' and o2 has the OM u". The only difference in both cases is the

handling when o1 has the OM any_u. In the case of a field update this is not allowed. In the case of method parameters this is only allowed if the method is pure. In the case of method parameters of non-pure methods this is not allowed.
Because there are only a limited number of OM combinations possible to adapt o2 to the viewpoint of the Receiving Variable o1 and then to be a subtype of f, the following table shows which combinations of u and u' require which OM for u".
The entries in the table FU with false value are the cases where no OM exists in the current viewpoint, such that it can be adapted to the viewpoint of u and be a subtype of f. With this table the function FU(u,u') can be defined. To show the equality of the OM adaptation checks in both papers, the methods in the papers are compared with the function FU(u,u').

FU(u,u')	peer	rep	any
this	peer	rep	any
peer	peer	false	any
rep	rep	false	any

Some notes:
As Rep variables of a Peer variable are Any variables in the current viewpoint, there exists no counterpart in the current viewpoint.
As Rep variables of a Rep variable are Any variables in the current viewpoint, there exists no counterpart in the current viewpoint.
As Peer variables of a Rep variable are Rep variables in the current viewpoint, the counterpart in the current viewpoint are Rep variables.
As Peer variables of a Peer variable are Peer variables in the current viewpoint, the counterpart in the current viewpoint are Peer variables.

In the GUT paper two things are done in cases c and d. First the rp function is used with the OM of o1 and the type of f to test if the update is allowed. If it is allowed, then the type of f is adapted to the current viewpoint with the $\triangleright$ function. Those two steps together should result in using the function FU(u,u').
Comparing rp with FU, it can be seen that for every combination u and u' where rp is false, FU is false, too, and vice versa.
This means that the first step in the GUT paper can be done with both functions rp and FU. Comparing $\triangleright$ with FU in the cases where the combination of u and u' returns an OM in the function FU, it can be seen that both functions return the same values.
So the combination of rp and $\triangleright$ does the same as the function FU in the cases c and d.

Have a look at the $\ominus$ operator in the UJ Paper: This operator adapts the OM of a variable in the current viewpointto the viewpoint of a variable o1 (u" $\ominus$ u = u'), The comparison of $\ominus$ with the function is not as easy as in the GUT paper.

The following lookup table LT(u,u') should help comparing them. It looks for every combination of u and u', in which u" is allowed such that u" $\ominus$ u = u'.
With that precondition we know that u is not allowed to be of OM Any and u' is not allowed to be of OM This.
In those cases where no u" exists for a combination of u and u', the lookup table should return false, meaning this combination is not possible.

LT(u,u')	peer	rep	any
this	peer	rep	any
peer	peer	false	any
rep	rep	false	any

Because FU(u,u')=LT(u,u') for every combination of u and u' both are equal. So using FU(u,u') results in the same checks for viewpoint adaptation as using $\ominus$. So they do the same in the context of viewpoint adaptation checks for the cases c and d. Thus rp and $\triangleright$does the same as $\ominus$ in the cases c and d.

A.3.3 Case e

In a pure method m every parameter has to have the OM Any. As every variable with OM t and Class X is a subtype of a variable of OM Any and class X, there has to be done no viewpoint adaptation checks.

Bibliography

[1] D. Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, 2001.

[2] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, and P. Müller. UJ: Type Soundness for Universe Types. In *development*, 2007.

[3] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006.

[4] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[5] J. N. Foster and D. Vytiniotis. A theory of Featherweight Java in Isabelle/HOL. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net`, Apr. 2006.

[6] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[7] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[8] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, Mar. 2004.

[9] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[10] P. Müller. Reasoning about object structures using ownership. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[11] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[12] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer, 2003.

[13] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. `http://www.in.tum.de/~nipkow/LNCS2283/`.