Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Theoretische Informatik

Studienarbeit

Statically Checking Encapsulation in KeY with the Universe Type System

Benjamin Weiß

12. Januar 2006

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Andreas Roth

Danksagung

Ich möchte mich ganz herzlich bei Prof. Dr. Peter H. Schmitt für die Möglichkeit zur Durchführung dieser Arbeit an seinem Lehrstuhl bedanken, und besonders bei Andreas Roth für das interessante Thema sowie die kontinuierliche Unterstützung bei seiner Bearbeitung.

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Benjamin Weiß
Karlsruhe, den 12. Januar 2006

Contents

1 Introduction

1.1 Encapsulation

The concept of *encapsulation* is fundamental to most programming methodologies as a way to manage complexity. In the most general sense, it refers to hiding the internals of some module from the outside world.

Specifically in object oriented programming, these modules can be classes and objects. The implementation of an object often depends on other objects, which can be considered "part" of this object and should be encapsulated. Visibility modifiers help to achieve this. However, declaring e.g. a field as `private` is not sufficient: This only protects the *field*, not the object which it references. If there is another reference—an *alias*—to this object from the outside, it can still be seen and manipulated directly. So, in addition to visibility modifiers, there must be restrictions about which objects may have references to which other objects. This is the aspect of encapsulation which we are concerned with in this work.

A common notion in this context is that of a *representation* of an object, which contains all objects that are logically "part" of that object and may not be referenced from the outside. An object is said to be the *owner* of the objects in its representation.

Example. As a running example throughout this document we will look at a singly linked list using the "iterator" design pattern. An object diagram for this list is given in Figure 1.1.

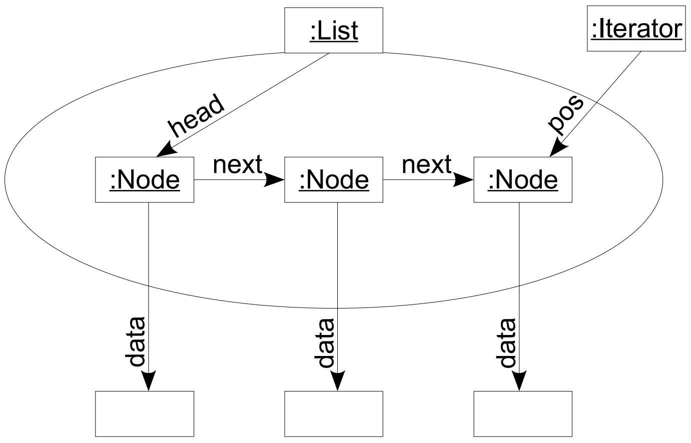

Figure 1.1: Object diagram for a linked list.

The oval line indicates the representation of the list object, which contains the nodes that form the list, but not the values stored in them. Iterator objects need to be allowed direct access to the nodes, as an exemption from the general rule.

1.2 KeY

The *KeY* project [ABB$^+$05] aims at integrating formal specification and verification into commercial software development. The core of the developed tool is a theorem prover based on a dynamic logic for Java, here called *JavaDL*. From within a regular CASE tool or IDE, assertions in the specification languages OCL and JML can be transformed into JavaDL proof obligations and passed on to the prover.

Encapsulation properties are relevant to KeY mainly as a prerequisite for modular verification of functional properties. They can be expressed in JavaDL in a very general way as *encapsulation predicates* [Rot05], and then be verified deductively.

1.3 Goal of this Work

Deductive verification of encapsulation predicates is difficult in general. On the other hand, there are number of approaches where encapsulation properties are encoded as type information, which can then be verified automatically by means of type checking. This work aims at utilising such approaches for statically checking encapsulation predicates within KeY. More precisely, after briefly looking into *Alias Burying* [Boy01], *Ownership Types* [CPN98] and *Confined Types* [VB01], we selected the *Universe Type System* [MPH01, DM05] as the preferred approach, because its idea of encapsulation matches the one needed for modular verification most closely. Such an analysis will necessarily neither be complete nor work for arbitrary encapsulation predicates, but it will be fully automated instead.

1.4 Structure

Chapter 2 provides an introduction to JavaDL, especially to its encapsulation predicates. Chapter 3 does the same for the Universe Type System. Afterwards, both worlds are merged and the outline of the intended analysis is given in Chapter 4. Chapter 5 then describes the type rules of the Universe Type System. These rules form the foundation for the implementation of the analysis, which is the subject of Chapter 6. Finally, Chapter 7 gives a summary and some ideas for future extensions.

2 JavaDL

This chapter presents the logic of the KeY system, JavaDL, as far as it is relevant for this work. First, Section 2.1 sets the scene by introducing some semantic concepts. Then, Section 2.2 provides a brief overview of the basic attributes of JavaDL. And finally, Section 2.3 describes how encapsulation properties can be expressed in JavaDL.

2.1 Semantic Setting

Throughout this document, we assume to have given a program in the Java programming language [GJSB00]. This program and the language itself bring along the following entities, which form the semantic base for both JavaDL and the Universe Type System:

- The set Types of types, partitioned into the set $\mathit{Types}^{\mathit{prim}}$ of primitive types and the set $\mathit{Types}^{\mathit{ref}}$ of reference types.
- The subtype relation $\preceq$.
- The set Fields of fields, partitioned into the set $\mathit{Fields}^{\mathit{prim}}$ of fields with a primitive type and the set $\mathit{Fields}^{\mathit{ref}}$ of fields with a reference type. In order to save notational overhead, we consider array components to be the same as fields: For every $t \in \mathit{Types}$ and $i \in \mathbb{N}$ there is a special symbol $\texttt{[]}_{t,i} \in \mathit{Fields}$, which models the ith array component of arrays with component type t.
- The set $\mathit{LocalVariables}$ of local variables, partitioned into the set $\mathit{LocalVariables}^{\mathit{prim}}$ of local variables with a primitive type and the set $\mathit{LocalVariables}^{\mathit{ref}}$ of local variables with a reference type.
- The set Values of all possible values, partitioned into the set $\mathit{Values}^{\mathit{prim}}$ of values with a primitive type, and the set $\mathit{Values}^{\mathit{ref}}$ of values with a reference type. The latter is also called $\mathit{Objects}$, and includes the special object null.
- The function $\mathit{typeof} : \mathit{Values} \rightarrow \mathit{Types}$, which yields the (dynamic) type of a value, and the function $[\cdot] : \mathit{Fields} \cup \mathit{LocalVariables} \rightarrow \mathit{Types}$, which yields the static type of a field or local variable.
- The "defined for" relation between fields and objects. An instance field f is defined for an object o iff $o \neq \mathit{null}$ and f is declared in $\mathit{typeof}(o)$ or any of its supertypes. We take array components $\texttt{[]}_{t,i}$ to be declared in the array type with component type t. Static fields are defined for all objects, including null.

- The set

$$\begin{aligned} \mathit{Locations} &= \{o.f \mid o \in \mathit{Objects},\ f \in \mathit{Fields},\ f \text{ is defined for } o\} \\ &\quad \cup \mathit{LocalVariables} \\ &\quad \cup \{\texttt{this}\} \end{aligned}$$

  of memory locations. Note that we do not treat static fields separately here, although in reality they only describe single memory locations which are not part of any object. This little trick saves us from having to distinguish between instance and static fields all the time later on, similar to our handling of array components.

- The set *States* of states, which are functions $s : \mathit{Locations} \to \mathit{Values}$. All states satisfy that for all static fields f and all objects o_1 and o_2, $s(o_1.f) = s(o_2.f)$.

2.2 Basics

This section sketches the relevant aspects of JavaDL, based on the corresponding section of [Rot05] and on [Bec01]. We begin with a look at the *syntax* of the logic:

Functions. Local variables are modeled as 0-ary functions of the same name, which are also called *program variables.* These functions are *non-rigid*, i.e. they can differ from state to state. Accordingly, instance fields are modeled as unary non-rigid functions whose argument specifies the receiver object. In this work we also treat array components and static fields as unary non-rigid functions.

Logical variables. Logical variables are variables in the usual logical sense. Unlike program variables, they are rigid, and can be quantified.

Terms. The set *Terms* of terms is constructed inductively from functions, logical variables, and the `null` literal. For a term t and a function f which denotes a field, the function application $f(t)$ is usually written as $t.f$.

Predicates. The set of predicates includes the equality predicate $\doteq$ and an unary predicate $\mathit{instanceof}_{tp}(\cdot)$ for every $tp \in \mathit{Types}^{ref}$.

Formulas. Formulas are constructed inductively from terms, predicates, the literals *true* and *false*, the junctors $\neg$, $\wedge$, $\vee$ and $\rightarrow$, and the quantifiers $\forall$ and $\exists$. Additionally, if π is an executable Java program fragment and φ is a formula, then $[\pi]\varphi$ is a formula.

The *semantics* depends on a state s and a variable assignment β, which maps logical variables to values:

Terms. The valuation $\mathit{val}_{s,\beta} : \mathit{Terms} \to \mathit{Values}$ is defined as follows:

  - for program variables v: $\mathit{val}_{s,\beta}(v) = s(v)$
  - for terms t and field functions f: $\mathit{val}_{s,\beta}(t.f) = s(\mathit{val}_{s,\beta}(t).f)$

- for logical variables x: $val_{s,\beta}(x) = \beta(x)$
- $val_{s,\beta}(\texttt{null}) = null$

Formulas. The validity $(s, \beta) \models$ of formulas is defined as follows:

- $(s, \beta) \models t_1 \doteq t_2$ iff $val_{s,\beta}(t_1) = val_{s,\beta}(t_2)$
- $(s, \beta) \models instanceof_{tp}(t)$ iff $(s, \beta) \models \exists x : tp\ x \doteq t \wedge \neg(t \doteq \texttt{null})$
- $(s, \beta) \models true$ always holds, $(s, \beta) \models false$ never holds
- $(s, \beta) \models \neg\varphi$ iff not $(s, \beta) \models \varphi$, etc.
- $(s, \beta) \models \forall x : tp\ \varphi$ iff for all values $v \in Values$ whose type is a subtype of tp, $(s, \beta_x^v) \models \varphi$ holds. Analogously for $\exists$.
- $(s, \beta) \models [\pi]\varphi$ iff π, when started in s, either does not terminate or terminates in a state s' with $(s', \beta) \models \varphi$. Throwing an uncaught exception is interpreted as non-termination.

If a formula φ does not contain free occurrences of logical variables, its validity does not depend on the variable assignment, and we can just write $s \models \varphi$.

Deductive verification is done in JavaDL by means of a *sequent calculus*. A sequent is a construct of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas; its semantics is the same as that of the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$.

2.3 Encapsulation Predicates

[Rot05] extends JavaDL by special predicates for specifying encapsulation properties. We will repeat here its most important definitions. The foundation is formed by two *basic encapsulation predicates*, called *acc* and *reachable*:

Definition (*acc*). Let $A \subseteq Fields^{ref}$, and $t_1, t_2 \in Terms$. Then $acc[A](t_1, t_2)$ is a formula.
For a state s and a variable assignment β, $(s, \beta) \models acc[A](t_1, t_2)$ is defined to hold iff $(val_{s,\beta}(t_1), val_{s,\beta}(t_2)) \in Acc[A]$. $Acc[A]$ is defined to be a relation on $Objects$ with $(e_0, e_1) \in Acc[A]$ iff there is a field $f \in A$ such that $e_1 = s(e_0\,.\,f)$.

Definition (*reachable*). Let $A \subseteq Fields^{ref}$, and $t_1, t_2 \in Terms$. Then $reachable[A](t_1, t_2)$ is a formula.
For a state s and a variable assignment β, $(s, \beta) \models reachable[A](t_1, t_2)$ is defined to hold iff there is a finite sequence of objects $(e_0, e_1, \ldots, e_k)$ $(k \in \mathbb{N})$ such that $e_0 = val_{s,\beta}(t_1)$, $e_k = val_{s,\beta}(t_2)$, and for all $i = 1, \ldots, k$: $(e_{i-1}, e_i) \in Acc[A]$.

Basically an object e_1 can access another object e_2 if there is a reference to e_2 stored in some field of e_1; recall that such a "field" may be an instance field, an array component, or a static field. Reachability is the reflexive and transitive closure of accessibility.

Note that no assertion is made about local variables. This is because, when specifying that a method `m` should preserve some encapsulation property φ, we need to look at it only from the outside: The only local variables which `m` can read without assigning to them first are its formal parameters; and the only local variable which it can assign to and whose value can still be read after its termination is its result. So, local variables are irrelevant if we, e.g., use a proof obligation of the form

$$\varphi \rightarrow [\{\texttt{Ano.r = Ano.c.m(Ano.p\_1, ..., Ano.p\_n);}\}]\varphi, \tag{2.1}$$

where `Ano` is some unspecified, "anonymous" class with static fields `r`, `c`, `p_1`, ..., `p_n` of suitable types.

On top of these basic predicates, there are several *macro encapsulation predicates* which provide abbreviations for more complex properties. We will use the following two:

Definition (*guardAcc*). For $A \subseteq \mathit{Fields}^{\mathit{ref}}$ and a formula φ:

$$\mathit{guardAcc}_x[A; \varphi(x)](u) = \forall y \big(\mathit{acc}[A](y, u) \rightarrow \varphi(y)\big)$$

Definition (*guardReg*). For $A \subseteq \mathit{Fields}^{\mathit{ref}}$ and a formula φ:

$$\begin{aligned}\mathit{guardReg}_x[A; \varphi(x)](u) = \forall z \big(&\mathit{reachable}[A](u, z) \\ &\rightarrow \mathit{guardAcc}[\varphi(x) \lor \mathit{reachable}[A](u, x)](z)\big)\end{aligned}$$

The *guardAcc* predicate guards an object by requiring that all objects which hold a reference to it must satisfy some formula φ. The *guardReg* predicate similarly guards an entire representation, which contains all objects reachable from some starting point.

Example. A sensible encapsulation property for our linked list is:

$$\forall l : \texttt{List}\ \mathit{guardReg}_x[\{\texttt{next}\}; x \doteq l \lor \mathit{instanceof}_{\texttt{Iterator}}(x)](l.\texttt{head})$$

It requires that nodes belonging to any list must be accessible only from each other, the respective list object, and arbitrary iterator objects.

3 The Universe Type System

The *Universe Type System* is an approach where encapsulation properties are expressed as type information, so that type rules can enforce the preservation of these properties during program execution. Section 3.1 provides a short introduction to the Universe Type System. Afterwards, its formal background is developed in Section 3.2.

3.1 Overview

The eponymous idea of the Universe Type System is to introduce many "parallel universes" which contain copies of the normal hierarchy of reference types. That is, for each reference type of the normal hierarchy, there is a distinct, but structurally identical, copy in every such parallel universe. The type of each object can then be freely chosen from these copies, and we can consider the object itself part of the chosen universe.

There is a one-to-one relationship between objects and parallel universes: Each object *owns* a unique parallel universe. Only the "normal", or "root" universe does not have an owner.

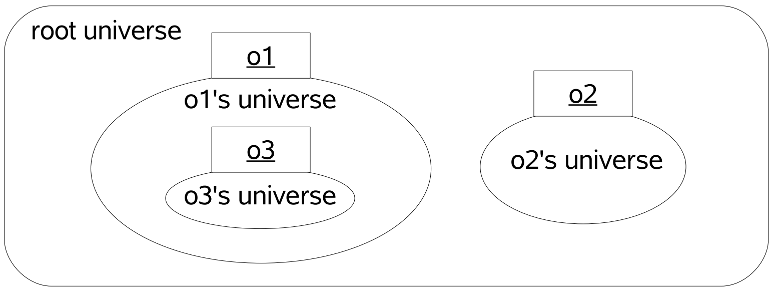

Figure 3.1: Objects *o1* and *o2* are in the root universe. Object *o3* resides in the universe which is owned by *o1*.

Universes directly correspond to representations; the representation of an object can be defined by putting all objects which it owns into its personal universe. The integrity of this representation can then be enforced by type checking: The universe type rules guarantee that the owner of a universe is the only object outside that universe which can get a read-write reference to objects inside it.

The Universe Type System allows programmers to specify universe membership by annotating declarations of reference variables with the following modifiers:

- `rep`: Such variables point into the universe owned by the enclosing object.

- `peer`: Variables of this kind hold references into the universe to which the enclosing object itself belongs.
- `readonly`: This modifier allows a variable to reference objects in arbitrary universes. However, the type rules ensure that such references cannot be used to modify their target in any way.
- `root`: Variables with this modifier always point into the root universe.

The `root` modifier is not part of the standard Universe Type System as described in [MPH01] and [DM05]. It is introduced in [Häc04] under the name of `global` as a modifier for static fields; without it, all static fields have to be `readonly`, since `rep` and `peer` have no well-defined meaning for variables which do not belong to an object. We adopt it here, as we find it useful both for static fields and beyond.

Example. A rudimentary but functional Java implementation for the linked list of our running example follows, together with suitable universe annotations. These are written in the style of the JML specification language [LBR98], into which the Universe Type System is integrated [DM05].

```
class Node {
    public /*@ peer @*/ Node next;
    public /*@ root @*/ Object data;
    public Node(/*@ root @*/ Object o) {
        data = o;
    }
}

public class List {
    private /*@ rep @*/ Node head;
    public void prepend(/*@ root @*/ Object data) {
        /*@ rep @*/ Node newHead = new /*@ rep @*/ Node(data);
        newHead.next = head;
        head = newHead;
    }
    public /*@ root @*/ Iterator iterator() {
        return new /*@ root @*/ Iterator(head);
    }
}

public class Iterator {
    private /*@ readonly @*/ Node pos;
    public Iterator(/*@ readonly @*/ Node n) {
        pos = n;
    }
    public boolean hasNext() {
       return pos != null;
    }
```

```
        public /*@ root @*/ Object next() {
            /*@ root @*/ Object posData = pos.data;
            pos = pos.next;
            return posData;
        }
    }
```

Annotating `head` with `rep` ensures that the head node is in the universe owned by the list object. Since `next` is `peer` and `data` is `root`, this universe also contains all other nodes of the list, but not the stored objects—just as sketched in Figure 1.1. Iterator objects may keep references to nodes of any list in their `pos` field, which is `readonly`.

3.2 Formalisation

This section builds up the formal background of the Universe Type System. In principle, this background is taken from [MPH01]. However, besides minor rephrasings, it has also been adapted in several larger ways: Firstly, the `root` modifier has been added. Secondly, the wording has been generalized to cover language features not present in the Java subset used in [MPH01], namely static fields, static methods, and arrays. And thirdly, there are two other modifications which are pointed out where they appear below.

Definition (Universes).

$$\mathit{Universes} = \{\mathit{rootU}\} \cup \{\mathit{repU}(o) \mid o \in \mathit{Objects}\}$$

The set of all universes consists of the root universe and a representation universe for every object. The functions *rootU* and *repU* act like constructors in the sense of abstract data types. In particular, this means that *repU* is injective, and that for all o $\mathit{repU}(o) \neq \mathit{rootU}$.

Definition (Universe Types).

$$\mathit{UniverseTypes} = \{\mathit{refT}(U) \mid U \in \mathit{Universes}\} \cup \{\mathit{roT}, \mathit{primitiveT}, \mathit{nullT}\}$$

Universe types are attached both to values and expressions, in addition to the normal Java type. They include a reference type which specifies a universe, and a read-only reference type which does not. Additionally, there is a primitive type and a null type.

This definition deviates from the corresponding one in [MPH01]: There, each kind of universe type is parametrised by a Java type, as in $\mathit{refT}(U, t)$, where t is a type. Universe types can then completely *replace* the normal Java types, since they contain the Java type information themselves. However, as long as all considered programs are correct Java, this is never needed. Therefore we treat universe types as *separate* entities, which is semantically equivalent but simplifies the presentation.

Definition (Universe Subtype Relation). The *universe subtype relation* $\preceq_U$ is the smallest reflexive, transitive relation on universe types which satisfies the following axioms for all $U \in \mathit{Universes}$:

$$\begin{aligned} &\mathit{nullT} && \preceq_U \mathit{refT}(U) \\ &\mathit{refT}(U) && \preceq_U \mathit{roT} \end{aligned}$$

Any universe type is a subtype of itself, *nullT* is a subtype of all non-primitive types, and *roT* is a supertype of all non-primitive types. We observe that $\mathit{refT}(U_1) \preceq_U \mathit{refT}(U_2)$ is equivalent to $U_1 = U_2$.

Definition (Type Schemes).

$$\mathit{TypeSchemes} = \{\mathit{primitiveS}, \mathit{repS}, \mathit{peerS}, \mathit{readonlyS}, \mathit{rootS}\}$$

The so-called *type schemes* are the formalisation of the modifiers described in the previous section. That is, *repS* is basically identical to the `rep` modifier, *peerS* to the `peer` modifier, and so on. Additionally, there is a type scheme for primitive variables.

Definition (τ_u). For a function $u : \mathit{Objects} \to \mathit{Universes}$:

$$\tau_u : \mathit{TypeSchemes} \times \mathit{Objects} \to \mathit{UniverseTypes}$$

$$\tau_u(s, o) = \begin{cases} \mathit{primitiveT} & \text{if } s = \mathit{primitiveS} \\ \mathit{refT}(\mathit{repU}(o)) & \text{if } s = \mathit{repS} \\ \mathit{refT}(u(o)) & \text{if } s = \mathit{peerS} \\ \mathit{roT} & \text{if } s = \mathit{readonlyS} \\ \mathit{refT}(\mathit{rootU}) & \text{if } s = \mathit{rootS} \end{cases}$$

The function called τ_u connects type schemes and universe types, depending on a given function u which allocates objects to universes. The interpretation of an individual type scheme can depend on an object which serves as a point of origin: The types corresponding to *repS* and *peerS* are the reference type for the representation universe of this object and for the universe of the object itself, respectively. Independently from this object, *readonlyS* is mapped to the read-only reference type, and *rootS* to the reference type for the root universe.

Definition (Dynamic Universe Types). For a function $u : \mathit{Objects} \to \mathit{Universes}$:

$$\mathit{typeof}_u : \mathit{Values} \to \mathit{UniverseTypes}$$

$$\mathit{typeof}_u(v) = \begin{cases} \mathit{primitiveT} & \text{if } v \in \mathit{Values}^{\mathit{prim}} \\ \mathit{nullT} & \text{if } v = \mathit{null} \\ \mathit{refT}(u(v)) & \text{otherwise} \end{cases}$$

The universe type of an object is the reference type for the universe the object is in.

Definition (Well-formedness). Given a function $a : \mathit{Fields} \cup \mathit{LocalVariables} \to \mathit{TypeSchemes}$, a state s is *well-formed* with respect to a—denoted by $\mathit{wf}_a(s)$—iff there is a function $u : \mathit{Objects} \to \mathit{Universes}$ such that

1. for every field f defined for an object o

$$\mathit{typeof}_u(s(o.f)) \preceq_U \tau_u(a(f), o) \tag{3.1}$$

2. for every local variable v

$$\mathit{typeof}_u(s(v)) \preceq_U \tau_u(a(v), s(\texttt{this})) \tag{3.2}$$

This defines the precise meaning of universe annotations, given as a function mapping fields and local variables to type schemes. Basically, there must be a way to allocate objects to universes such that the type of any value is a subtype of the type of its memory location. The latter type is defined by τ_u. For fields, it is based on the type scheme of the field and on the object containing the value; local variables behave like fields of the `this` object. Of course, when control flow is within a static method, there is no `this` object, and $s(\texttt{this})$ is undefined. The type rules will avoid this issue by requiring that local variables in static contexts only have type schemes *primitiveS*, *readonlyS* and *rootS*, for which the object does not matter.

Well-formedness is not a very convenient instrument for reasoning about encapsulation. [MPH01] also gives another property, called the *Universe Invariant*. Extended to cover the *rootS* scheme, it states that if one object has a reference to another, then either (1) they are in the same universe, or (2) the second one is in the representation universe of the first one, or (3) the reference is read-only, or (4) the second one belongs to the root universe. This is an implication of well-formedness, but obviously not an equivalent, as it is too imprecise. We will use the following, more exact version instead:

Definition (Universe Invariant). Given a function $a : \mathit{Fields} \cup \mathit{LocalVariables} \to \mathit{TypeSchemes}$, a state s satisfies the *Universe Invariant* with respect to a iff there is a function $u : \mathit{Objects} \to \mathit{Universes}$ such that

1. for every field f defined for an object o

$$\begin{aligned} a(f) = \mathit{repS},\ s(o.f) \neq \mathit{null} &\ \Rightarrow\ u(s(o.f)) = \mathit{repU}(o) \\ a(f) = \mathit{peerS},\ s(o.f) \neq \mathit{null} &\ \Rightarrow\ u(s(o.f)) = u(o) \\ a(f) = \mathit{rootS},\ s(o.f) \neq \mathit{null} &\ \Rightarrow\ u(s(o.f)) = \mathit{rootU} \end{aligned} \tag{3.3}$$

2. for every local variable v

$$\begin{aligned} a(f) = \mathit{repS},\ s(v) \neq \mathit{null} &\ \Rightarrow\ u(s(v)) = \mathit{repU}(s(\texttt{this})) \\ a(f) = \mathit{peerS},\ s(v) \neq \mathit{null} &\ \Rightarrow\ u(s(v)) = u(s(\texttt{this})) \\ a(f) = \mathit{rootS},\ s(v) \neq \mathit{null} &\ \Rightarrow\ u(s(v)) = \mathit{rootU} \end{aligned} \tag{3.4}$$

This definition is very close to the intuitive description given in the previous section: *repS* variables point into the representation universe of the current object, *peerS* ones into the universe which the object itself is in, and *rootS* ones into the root universe. No restrictions are imposed on the targets of *readonlyS* references.

The validity of this formulation of the Universe Invariant is established by the following lemma:

Lemma 3.1 (Equivalence of Well-formedness and the Universe Invariant). *If* $a : \mathit{Fields} \cup \mathit{LocalVariables} \to \mathit{TypeSchemes}$ *satisfies for all* $x \in \mathit{Fields} \cup \mathit{LocalVariables}$ *that*

1. *if* $[x] \in \mathit{Types}^{prim}$ *then* $a(x) = \mathit{primitiveS}$*, and*
2. *if* $[x] \in \mathit{Types}^{ref}$ *then* $a(x) \in \{\mathit{repS}, \mathit{peerS}, \mathit{readonlyS}, \mathit{rootS}\}$*,*

then (3.1) is equivalent to (3.3), and (3.2) is equivalent to (3.4).

Proof. We only carry out the proof for fields, local variables are analogous. Let a be a function of the required kind, s be a state, and f be a field defined for an object o. We know that the normal Java typing is valid, that is

$$\mathit{typeof}(s(o.f)) \preceq [f] \tag{3.5}$$

"⇒" : We assume (3.1), and we want to show (3.3).

- $a(f) = \mathit{repS}$ and $s(o.f) \neq \mathit{null}$. Because of the restrictions imposed on a, this means $[f] \in \mathit{Types}^{ref}$, which together with (3.5) implies $s(o.f) \in \mathit{Objects}$. Thus, $\mathit{typeof}_u(s(o.f)) = \mathit{refT}(u(s(o.f)))$.
  Also, $\tau_u(a(f), o) = \mathit{refT}(\mathit{repU}(o))$.
  We conclude that (3.1) is equivalent to $\mathit{refT}(u(s(o.f))) \preceq_U \mathit{refT}(\mathit{repU}(o))$, which implies $u(s(o.f)) = \mathit{repU}(o)$.
- $a(f) = \mathit{peerS}$ and $s(o.f) \neq \mathit{null}$. As above, this means $\mathit{typeof}_u(s(o.f)) = \mathit{refT}(u(s(o.f)))$.
  We also get $\tau_u(a(f), o) = \mathit{refT}(u(o))$.
  This turns (3.1) into $\mathit{refT}(u(s(o.f))) \preceq_U \mathit{refT}(u(o))$, which can only hold if $u(s(o.f)) = u(o)$ holds as well.
- $a(f) = \mathit{rootS}$ and $s(o.f) \neq \mathit{null}$. Then, once again, $\mathit{typeof}_u(s(o.f)) = \mathit{refT}(u(s(o.f)))$.
  This time $\tau_u(a(f), o) = \mathit{refT}(\mathit{rootU})$.
  (3.1) now is $\mathit{refT}(u(s(o.f))) \preceq_U \mathit{refT}(\mathit{rootU})$, which implies $u(s(o.f)) = \mathit{rootU}$.

"⇐" : Let (3.3) hold. Our goal is to show (3.1). One of the following cases must apply:

- $[f] \in \mathit{Types}^{prim}$. Then because of (3.5) $\mathit{typeof}(s(o.f)) \in \mathit{Types}^{prim}$, i.e. $s(o.f) \in \mathit{Values}^{prim}$. Thus, $\mathit{typeof}_u(s(o.f)) = \mathit{primitiveT}$.
  Also, $a(f) = \mathit{primitiveS}$, and so $\tau_u(a(f), o) = \mathit{primitiveT}$.
  So, (3.1) is equivalent to $\mathit{primitiveT} \preceq_U \mathit{primitiveT}$, which holds by definition.

- $[f] \in \mathit{Types}^{\mathit{ref}}$ and $s(o.f) = \mathit{null}$. Then $\mathit{typeof}_u(s(o.f)) = \mathit{nullT}$.
  Moreover, $a(f) \neq \mathit{primitiveS}$, and thus $\tau_u(a(f), o) \neq \mathit{primitiveS}$.
  Since $\mathit{nullT} \preceq_U T$ for any non-primitive universe type T, these conclusions imply (3.1).
- $[f] \in \mathit{Types}^{\mathit{ref}}$ and $s(o.f) \neq \mathit{null}$. (3.5) tells us that $s(o.f) \notin \mathit{Values}^{\mathit{prim}}$. Thus, $\mathit{typeof}_u(s(o.f)) = \mathit{refT}(u(s(o.f)))$.
  * $a(f) = \mathit{repS}$. Now (3.3) tells us that $u(s(o.f)) = \mathit{repU}(o)$.
    Additionally $\tau_u(a(f), o) = \mathit{refT}(\mathit{repU}(o))$.
    Altogether (3.1) becomes $\mathit{refT}(\mathit{repU}(o)) \preceq_U \mathit{refT}(\mathit{repU}(o))$, which holds by definition.
  * $a(f) = \mathit{peerS}$. From (3.3) we get $u(s(o.f)) = u(o)$.
    Also, $\tau_u(a(f), o) = \mathit{refT}(u(o))$.
    (3.1) turns into $\mathit{refT}(u(o)) \preceq_U \mathit{refT}(u(o))$, which is true.
  * $a(f) = \mathit{readonlyS}$. In this case $\tau_u(a(f), o) = \mathit{roT}$, so (3.1) becomes $\mathit{refT}(u(s(o.f))) \preceq_U \mathit{roT}$, which holds by definition, no matter what $u(s(o.f))$ may be.
  * $a(f) = \mathit{rootS}$. Because of (3.3), $u(s(o.f)) = \mathit{rootU}$.
    Moreover $\tau_u(a(f), o) = \mathit{refT}(\mathit{rootU})$.
    Now (3.1) is $\mathit{refT}(\mathit{rootU}) \preceq_U \mathit{refT}(\mathit{rootU})$. Once again, this applies by definition. □

Lemma 3.2 (Type Safety). *If a program is type correct regarding the annotations defined by some a, then for each execution of that program starting in a state which is well-formed with respect to a, the terminating state and all intermediate states will also be well-formed with respect to a.*

This is the central guarantee of the Universe Type System. It is proven in [MPH01], albeit for a more basic Universe Type System which lacks the *rootS* scheme and only supports a small, demonstratory subset of Java.

"Type correct" means that the program complies with the universe type rules. We postpone looking at these to Chapter 5, and determine how to utilize type safety for our purposes first.

4 Matching Encapsulation Predicates and Universes

In order to employ the Universe Type System for the static verification of an encapsulation property formulated in JavaDL, a match must be found between the two worlds. That is, we need to identify a class of first-order formulas and corresponding universe annotations, such that both have the same meaning.
Section 4.1 points out some difficulties with doing so in the most direct way. This motivates a different approach, which is presented in Section 4.2. Finally, the correctness of this match is justified in Section 4.3.

4.1 First Attempt

As we have seen on the linked list example, the *guardReg* predicate and the Universe Type System can be used to express very similar properties. So, a conceivable approach would be to use formulas of the form

$$\bigwedge_{r \in R} \forall g : G \; \mathit{guardReg}_x[P; x \doteq g \vee \bigvee_{F \in \mathcal{F}} \mathit{instanceof}_F(x)](g \,.\, r) \tag{4.1}$$

where R and P are sets of fields and $\mathcal{F}$ is a set of “friendly” classes whose objects get the special privilege of being allowed to access the representation of objects of type G. Now the fields in R would be annotated as *repS*, those in P as *peerS*, those declared in friendly classes as *readonlyS*, and all other reference fields as *rootS*.

However, there are a number of somewhat subtle semantical differences between these two assertions:

- The formula (4.1) forbids references to reachable objects from objects which are neither the owner, nor instances of a friendly class, nor reachable themselves. But if such a reference is contained in an element of P, the state can still be well-formed, for instance if the referencing object is itself not referenced from anywhere (Figure 4.1, left). Such a state e.g. occurs in the linked list example if a method is executed which removes the head of the list by doing something like `head = head.next`. Then the former head node is not reachable from the list anymore, but its `next` field still points to a node of the list—which is now illegal in terms of (4.1). Since the head node can still be allocated to the representation universe, the state is nevertheless well-formed.

- With (4.1), several representations are defined for a single object, namely one for each conjunct. References between them (Figure 4.1, right) are not allowed. With the Universe Type System, there is only a single, big representation per object.
- The universe annotations only allow fields annotated *repS* to point into the representation of the current object. But (4.1) allows the owner to store references into its representation in *arbitrary* fields (Figure 4.2).

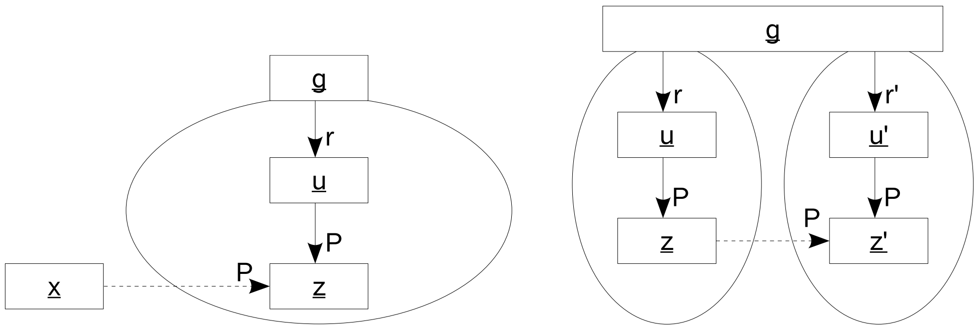

Figure 4.1: The oval shapes depict the representation boundaries as defined by (4.1). The dotted references violate these, but are allowed by the universe annotations.

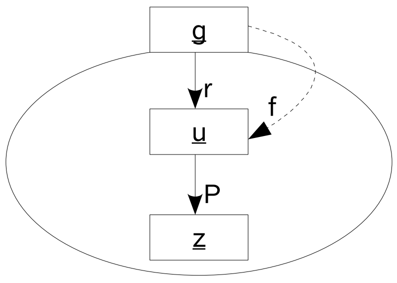

Figure 4.2: The oval shape depicts the representation boundary as defined by the universe annotations. The dotted reference violates it, but is allowed by (4.1).

4.2 The *protect* Predicate

The flexibility of the Universe Type System is limited; the incompatibilities described in the previous section cannot be overcome by choosing different annotations. It is however possible to do so by adapting the JavaDL formula.

In order to eliminate the problem sketched in Figure 4.1 (left), the representation needs to be extended to objects like x—that is, objects which are not reachable from u, but linked to it by references with *arbitrary* directions. As a preparation for expressing this, we need a new basic encapsulation predicate which is similar to *reachable*, but ignores the direction of the references in the chain.

Definition (*connected*). Let $A \subseteq \mathit{Fields}^{\mathit{ref}}$, t_1, $t_2 \in \mathit{Terms}$. Then $\mathit{connected}[A](t_1, t_2)$ is a formula.
For a state s and a variable assignment β, $(s, \beta) \models \mathit{connected}[A](t_1, t_2)$ is defined to hold iff there is a finite sequence of objects $(e_0, e_1, \ldots, e_k)$ ($k \in \mathbb{N}$) such that $e_0 = \mathit{val}_{s,\beta}(t_1)$, $e_k = \mathit{val}_{s,\beta}(t_2)$, for all $i = 1, \ldots, k$: $(e_{i-1}, e_i) \in \mathit{Acc}[A]$ or $(e_i, e_{i-1}) \in \mathit{Acc}[A]$, and for all $i = 0, \ldots, k$: $e_i \neq \mathit{null}$.

It is necessary to require that the e_i are different from *null* to prevent objects from getting connected just by having references to *null*. Obviously, connectedness is transitive and symmetric.

On top of this, we can define when one object should own another:

Definition (*own*). For mutually disjoint sets $R, P \subseteq \mathit{Fields}^{\mathit{ref}}$:

$$\mathit{own}[R|P](g, z) = \exists u\big(\mathit{acc}[R](g, u) \land \mathit{connected}[P](u, z)\big)$$

This establishes representation boundaries which conform to those expressible in the Universe Type System: Firstly, the problem of Figure 4.1 (left) is avoided by using *connected* instead of *reachable*. And secondly, this property holds independently of the field in R to whose value an object is connected, which serves the goal to get just one representation for each object, unlike in Figure 4.1 (right).

Now we can finally define our replacement for (4.1):

Definition (*protect*). For mutually disjoint sets $R, P, F \subseteq \mathit{Fields}^{\mathit{ref}}$:

$$\begin{aligned}\mathit{protect}[R|P|F] = \forall g \forall z\big(&\mathit{own}[R|P](g, z)\\ &\rightarrow (\mathit{guardAcc}_x[R; x \doteq g](z)\\ &\quad \land \mathit{guardAcc}_x[\mathit{Fields} \setminus (R \cup P \cup F); \mathit{false}](z))\big)\end{aligned}$$

or equivalently

$$\begin{aligned}\mathit{protect}[R|P|F] = \forall g \forall z\big(&\mathit{own}[R|P](g, z)\\ &\rightarrow (\forall y(\mathit{acc}[R](y, z) \rightarrow y \doteq g)\\ &\quad \land \forall y\ \neg \mathit{acc}[\mathit{Fields} \setminus (R \cup P \cup F)](y, z))\big)\end{aligned}$$

This restricts access by fields in R to the owner and completely forbids access by fields not in R, P or F. What about P and F? Objects holding references to a representation in members of P are part of this representation themselves. And F is a set of "friendly" fields which may point anywhere. The problem of Figure 4.2 is not present with this formulation, since it does not grant the owner access to its representation by arbitrary fields.

Example. Using *protect*, the encapsulation of the nodes of the linked list can be specified as

$$\mathit{protect}[\{\texttt{head}\}|\{\texttt{next}\}|\{\texttt{pos}\}]$$

The adequate universe annotations for $\mathit{protect}[R|P|F]$ are obvious:

Definition (Annotations for *protect*).

$$\mathit{annotate}_{R,P,F} : \mathit{Fields} \cup \mathit{LocalVariables} \to \mathit{TypeSchemes}$$

$$\mathit{annotate}_{R,P,F}(x) = \begin{cases} \mathit{primitiveS} & \text{if } [x] \in \mathit{Types}^{\mathit{prim}} \\ \mathit{repS} & \text{if } x \in R \\ \mathit{peerS} & \text{if } x \in P \\ \mathit{readonlyS} & \text{if } x \in F \\ \mathit{rootS} & \text{if } x \in \mathit{Fields} \setminus (R \cup P \cup F) \end{cases}$$

For $x \in \mathit{LocalVariables}^{\mathit{ref}}$ we only define here that $\mathit{annotate}_{R,P,F}(x) \in \{\mathit{repS}, \mathit{peerS}, \mathit{readonlyS}, \mathit{rootS}\}$, which is required for the applicability of Lemma 3.1. Fixing precise annotations for local reference variables is not necessary at this point, because, like the other encapsulation predicates, *protect* does not say anything about local variables. We will return to this issue in Chapter 6.

The preservation of *protect* by a program π can be proven with the following rule:

Definition (Sequent calculus rule for *protect*).

$$\frac{\Gamma, \mathit{protect}[R|P|F] \;\vdash\; \#\mathit{universes}([\pi]\ \mathit{protect}[R|P|F]), \Delta}{\Gamma, \mathit{protect}[R|P|F] \;\vdash\; [\pi]\ \mathit{protect}[R|P|F], \Delta} \quad \#\mathit{universes}$$

where Γ and Δ are arbitrary sets of formulas and $\#\mathit{universes}(\cdot)$ is a meta construct which evaluates either to its argument or to *true*, and may do the latter only if

1. π does not read the initial value of any local reference variable, and

2. π is type correct regarding $\mathit{annotate}_{R,P,F}$.

By "initial value" of a local variable we mean the value assigned to it in the state in which π is started. The requirement that π does not read such values is necessary because, unlike *protect*, well-formedness does cover local variables. It is for instance satisfied by programs of the kind mentioned in (2.1).

The soundness of this rule will be established in the next section.

4.3 Soundness Proof

Lemma 4.1 (Unique owners). *Let s be a state. If $s \models \mathit{protect}[R|P|F]$, then for each object x there is at most one object g such that $s \models \mathit{own}[R|P](g, x)$.*

Proof. Let $s \models \mathit{protect}[R|P|F]$, and let x, g_1 and g_2 be objects such that both $s \models \mathit{own}[R|P](g_1, x)$ and $s \models \mathit{own}[R|P](g_2, x)$. That means there are objects u_1, u_2 with $s \models \mathit{acc}[R](g_1, u_1)$, $s \models \mathit{acc}[R](g_2, u_2)$, $s \models \mathit{connected}[P](u_1, x)$, and $s \models \mathit{connected}[P](u_2, x)$. Since *connected* is transitive and symmetric, $s \models \mathit{connected}[P](u_1, u_2)$ must hold as well. This means $s \models \mathit{own}[R|P](g_1, u_2)$. Now $s \models \mathit{protect}[R|P|F]$ yields $s \models \mathit{guardAcc}_x[R; x \doteq g_1](u_2)$, and together with $s \models \mathit{acc}[R](g_2, u_2)$, this implies $g_1 = g_2$. □

Lemma 4.2. *For a state s with $s(v) = null$ for all $v \in LocalVariables^{ref}$:*

$$s \models protect[R|P|F] \quad \Rightarrow \quad wf_{annotate_{R,P,F}}(s)$$

Proof. Let s be such a state. Because of Lemma 4.1, we can unambiguously define a mapping of objects to universes for s in the following way:

$$univ : Objects \rightarrow Universes$$

$$univ(x) = \begin{cases} repU(g) & \text{if there is an object } g \text{ such that } s \models own[R|P](g, x) \\ rootU & \text{otherwise} \end{cases}$$

Since $annotate_{R,P,F}$ adheres to the restrictions of Lemma 3.1, we can prove that s is well-formed by showing that it satisfies the Universe Invariant.

1. Let f be a field defined for an object o.

   - $annotate_{R,P,F}(f) = repS$ and $s(o.f) \neq null$. Then $f \in R$, which means $s \models acc[R](o, s(o.f))$, and thus $s \models own[R|P](o, s(o.f))$. By definition we get $univ(s(o.f)) = repU(o)$.
   - $annotate_{R,P,F}(f) = peerS$ and $s(o.f) \neq null$. This implies $f \in P$, and therefore $s \models acc[P](o, s(o.f))$.
     - There is an object g such that $s \models own[R|P](g, o)$, i.e. there is an object u with $s \models acc[R](g, u)$ and $s \models connected[P](u, o)$. Then obviously also $s \models connected[P](u, s(o.f))$, and thus $s \models own[R|P](g, s(o.f))$. We conclude $univ(s(o.f)) = repU(g) = univ(o)$.
     - There is no such g. Assume for a moment that there is a g' with $s \models own[R|P](g', s(o.f))$, i.e. there is an object u' such that $s \models acc[R](g', u')$ and $s \models connected[P](u', s(o.f))$. Then also $s \models connected[P](u', o)$ and thus, contradictorily, $s \models own[R|P](g', o)$. So, there is no such g' either. That means $univ(s(o.f)) = rootU = univ(o)$.
   - $annotate_{R,P,F}(f) = rootS$ and $s(o.f) \neq null$. Then $f \in Fields^{ref} \setminus (R \cup P \cup F)$, and so $s \models acc[Fields \setminus (R \cup P \cup F)](o, s(o.f))$. Now if there were an object g such that $s \models own[R|P](g, s(o.f))$, $s \models protect[R|P|F]$ would imply $s \models false$. So there cannot be such a g, which means that $univ(s(o.f)) = rootU$.

2. For local reference variables v we postulated that $s(v) = null$, so there is nothing to show. □

Lemma 4.3. *For any state s:*

$$wf_{annotate_{R,P,F}}(s) \quad \Rightarrow \quad s \models protect[R|P|F]$$

Proof. Let s be a state which is well-formed with respect to $annotate_{R,P,F}$, and $univ$ be the function mapping objects to universes such that the criteria of well-formedness are met. Since $annotate_{R,P,F}$ adheres to the restrictions of Lemma 3.1, $univ$ also establishes

the Universe Invariant.
We will do a proof by contradiction and assume that $s \models \mathit{protect}[R|P|F]$ does not hold. That means there are objects g and z such that $s \models \mathit{own}[R|P](g,z)$, but either not $s \models \mathit{guardAcc}_x[\mathit{Fields} \setminus (R \cup P \cup F); \mathit{false}](z)$ or not $s \models \mathit{guardAcc}_x[R; x \doteq g](z)$.
Now $s \models \mathit{own}[R|P](g,z)$ tells us that there is an object u such that $s \models \mathit{acc}[R](g,u)$ and $s \models \mathit{connected}[P](u,z)$. In more detail, the latter means that there is a sequence of objects $(e_0, \ldots, e_k)$ such that $e_0 = u$ and $e_k = z$ and for all $i = 1, \ldots, k$: $(e_{i-1}, e_i) \in \mathit{Acc}[P]$ or $(e_i, e_{i-1}) \in \mathit{Acc}[P]$, and for all $i = 0, \ldots, k$: $e_i \neq \mathit{null}$. So there is always a field $f \in P$ such that either $s(e_{i-1}\,.\,f) = e_i$ or $s(e_i\,.\,f) = e_{i-1}$. Since neither e_{i-1} nor e_i may be null, and since $\mathit{annotate}_{R,P,F}(f) = \mathit{peerS}$, the Universe Invariant in both cases implies $\mathit{univ}(e_i) = \mathit{univ}(e_{i-1})$, and consequently, $\mathit{univ}(z) = \mathit{univ}(u)$.
From $s \models \mathit{acc}[R](g,u)$ we get that there is a field $r \in R$ such that $s(g\,.\,r) = u$. Also, $\mathit{annotate}_{R,P,F}(r) = \mathit{repS}$, and $u = e_0 \neq \mathit{null}$. Together with the Universe Invariant, this means $\mathit{univ}(u) = \mathit{repU}(g)$.
Altogether, we conclude $\mathit{univ}(z) = \mathit{repU}(g)$.
Now one of the following cases must hold:

- Not $s \models \mathit{guardAcc}_x[\mathit{Fields} \setminus (R \cup P \cup F); \mathit{false}](z)$. That means there is an object y such that $s \models \mathit{acc}[\mathit{Fields} \setminus (R \cup P \cup F)](y,z)$, i.e. there is a field $f \in \mathit{Fields} \setminus (R \cup P \cup F)$ and $s(y\,.\,f) = z$. Now $\mathit{annotate}_{R,P,F}(f) = \mathit{rootS}$, and so, since $z = e_k \neq \mathit{null}$, the Universe Invariant yields $\mathit{univ}(z) = \mathit{rootU}$, in contradiction to $\mathit{univ}(z) = \mathit{repU}(g)$.

- Not $s \models \mathit{guardAcc}_x[R; x \doteq g](z)$. Then there is an object $y \neq g$ such that $s \models \mathit{acc}[R](y,z)$, i.e. there is a field $f \in R$ and $s(y\,.\,f) = z$. Also $\mathit{annotate}_{R,P,F}(f) = \mathit{repS}$, and since $z \neq \mathit{null}$, we get from the Universe Invariant $\mathit{univ}(z) = \mathit{repU}(y)$. That means $\mathit{repU}(g) = \mathit{repU}(y)$, and so, contradictorily, $g = y$. $\square$

Lemma 4.4. *The #universes rule is sound, i.e. if*

1. $s \models \mathit{protect}[R|P|F]$, *and*

2. π *does not read the initial value of any local reference variable, and*

3. π *is type correct regarding* $\mathit{annotate}_{R,P,F}$,

then

$$s \models [\pi]\ \mathit{protect}[R|P|F]$$

Proof. Let s be a state with $s \models \mathit{protect}[R|P|F]$, and π be a program which does not read the initial value of any local reference variable and which is type correct regarding $\mathit{annotate}_{R,P,F}$.
Also, let for a moment $s(v) = \mathit{null}$ for all $v \in \mathit{LocalVariables}^{\mathit{ref}}$. Then we know from Lemma 4.2 that $\mathit{wf}_{\mathit{annotate}_{R,P,F}}(s)$. Lemma 3.2 now tells us that if s' is a state in which π terminates after being started in s, then also $\mathit{wf}_{\mathit{annotate}_{R,P,F}}(s')$. Together with Lemma 4.3, this implies $s' \models \mathit{protect}[R|P|F]$. We conclude that $s \models [\pi]\ \mathit{protect}[R|P|F]$ holds if s fulfills this additional restriction.
Because π does not read initial values of local reference variables, its execution is not in

any way influenced by these values. Thus, the restriction does not affect the validity of the formula, and can be omitted. □

5 Universe Type Correctness

In this chapter we deal with the question of what it means for a program to be type correct regarding some universe annotations. The answer is given by a set of type rules with which a program has to comply. Like for Section 3.2, the formal contents of this chapter are based on [MPH01], but have undergone several modifications:

- The *rootS* scheme has been added.
- The rule set has been generalised beyond the demonstratory subset of Java covered by the original paper. As mentioned before, this subset lacks static fields, static methods, and arrays. It also has various simpler restrictions like only allowing methods with exactly one parameter, and only allowing field accesses and method calls on *variables* instead of on arbitrary expressions.
- The restriction that *readonlyS* references may not be used for modifying the object they point to is an extra guarantee that the Universe Type System gives in addition to type safety. In this work, we do not actually need this guarantee anywhere; we only use *readonlyS* references because they can point into arbitrary universes. Thus, strictly enforcing this restriction would mean to needlessly reject certain programs as "not type correct", although they do preserve *protect*. The rules have instead been liberalised in this respect wherever this has been possible without breaking type safety.
- Standard universes allow explicit downcasts from *readonlyS* to other type schemes. As the admissibility of such casts cannot be checked statically, this feature is not usable for our purposes and has been dropped.

Section 5.1 introduces the operators by which the type rules are built. The rules themselves are then presented in sections 5.2, 5.3 and 5.4, grouped by whether they work on primitive expressions, composite expressions, or statements.

5.1 Preliminaries

Despite their name, the type rules do not operate on universe types but on type schemes, since only these are available statically. Type schemes have so far been attached to fields and local variables; in order to extend this to arbitrary expressions, we need a few additional ones.

Definition (Extended Type Schemes).

$$\mathit{ExtTypeSchemes} = \mathit{TypeSchemes} \cup \{\mathit{nullS}, \mathit{thisS}, \mathit{readonlyS}'\}$$

The purpose of *nullS* and *thisS* is straightforward, namely to describe `null` and `this`. *readonlyS*$'$ is used for expressions that—unlike *readonlyS* expressions—are "read only" in the sense that nothing may be assigned to them *themselves.* In all other respects, *readonlyS*$'$ behaves like *readonlyS*.

Definition (Subscheme Relation). The *subscheme relation* $\preceq_S$ is the smallest transitive relation on extended type schemes satisfying the following axioms:

$$\begin{array}{ll}
\mathit{primitiveS} & \preceq_S \mathit{primitiveS} \\
\mathit{repS} & \preceq_S \mathit{repS} \\
\mathit{peerS} & \preceq_S \mathit{peerS} \\
\mathit{readonlyS} & \preceq_S \mathit{readonlyS} \\
\mathit{rootS} & \preceq_S \mathit{rootS} \\
\mathit{repS} & \preceq_S \mathit{readonlyS} \\
\mathit{peerS} & \preceq_S \mathit{readonlyS} \\
\mathit{rootS} & \preceq_S \mathit{readonlyS} \\
\mathit{nullS} & \preceq_S \mathit{repS} \\
\mathit{nullS} & \preceq_S \mathit{peerS} \\
\mathit{nullS} & \preceq_S \mathit{rootS} \\
\mathit{thisS} & \preceq_S \mathit{peerS} \\
\mathit{readonlyS}' & \preceq_S \mathit{readonlyS}
\end{array}$$

This relation describes assignability between type schemes. It is also depicted graphically in Figure 5.1. Essentially, each regular type scheme is assignable to itself, all non-primitive regular type schemes are assignable to *readonlyS*, *nullS* is assignable to any non-primitive regular type scheme, *thisS* is assignable to *peerS* and *readonlyS*, and *readonlyS*$'$ is assignable to *readonlyS*.

$$\begin{tikzcd}
 & & \mathit{readonlyS} \arrow[loop left] & & \\
\mathit{primitiveS} \arrow[loop left] & \mathit{repS} \arrow[loop left] \arrow[ur] & \mathit{peerS} \arrow[loop left] \arrow[u] & \mathit{rootS} \arrow[loop left] \arrow[ul] & \mathit{readonlyS'} \arrow[ull] \\
 & & \mathit{nullS} \arrow[ul] \arrow[u] \arrow[ur] & \mathit{thisS} \arrow[ul] &
\end{tikzcd}$$

Figure 5.1: The subscheme relation as a graph. Transitive edges are not shown.

The meaning of a type scheme is normally given relative to the enclosing object. However, in a field access expression e`.`f, f is a field of the object described by expression e, and so the type scheme of f must be interpreted in relation to this object. The same thing is true for the type schemes of formal parameters and the method result in method call expressions e`.`m`(`$e_1, \ldots, e_n$`)`. The *type scheme combinator* $*$ describes how this is done: The type scheme of the receiver expression e and the type scheme of the field, formal parameter or method result are combined into a type scheme which is again relative to the enclosing object. For example, if TS_1 is the type scheme of e and TS_2 the type scheme of f, the type scheme of the expression $e.f$ is calculated as $TS_1 * TS_2$.

Definition (Type Scheme Combinator).

$$* : \mathit{ExtTypeSchemes} \times \mathit{TypeSchemes} \rightarrow \mathit{ExtTypeSchemes}$$

The values are defined by the following table (first argument: rows, second argument: columns; combinations with *nullS* are undefined):

	primitiveS	*repS*	*peerS*	*readonlyS*	*rootS*
thisS	*primitiveS*	*repS*	*peerS*	*readonlyS*	*rootS*
repS	*primitiveS*	$\mathit{readonlyS}'$	*repS*	*readonlyS*	*rootS*
peerS	*primitiveS*	$\mathit{readonlyS}'$	*peerS*	*readonlyS*	*rootS*
readonlyS	*primitiveS*	$\mathit{readonlyS}'$	$\mathit{readonlyS}'$	*readonlyS*	*rootS*
$\mathit{readonlyS}'$	*primitiveS*	$\mathit{readonlyS}'$	$\mathit{readonlyS}'$	*readonlyS*	*rootS*
rootS	*primitiveS*	$\mathit{readonlyS}'$	*rootS*	*readonlyS*	*rootS*

The first argument is the type scheme of the receiver expression, which can be any of the extended schemes. The second argument is the type scheme of a variable, so it must be one of the regular schemes. We will go over the table column by column:

- *primitiveS*: This type scheme is not relative to any object, so the receiver does not matter when accessing such a variable. If we were to enforce that *readonlyS* references cannot be used to modify the target object, then $\mathit{readonlyS} * \mathit{primitiveS}$ and $\mathit{readonlyS}' * \mathit{primitiveS}$ would have to be $\mathit{readonlyS}'$, because the overall expression in these cases describes—when used on the left hand side of an assignment—a variable of this target object. But since *primitiveS* means the same on any object, we can relax this without breaking anything.

- *repS*: With any other receiver than `this`, the resulting expression points into a foreign representation universe. The only type schemes which allow this are *readonlyS* and $\mathit{readonlyS}'$. Since *readonlyS* would allow almost *anything* to be assigned to the expression, although the variable which it describes is not *readonlyS*, the type scheme of the expression has to be $\mathit{readonlyS}'$.

- *peerS*: Such variables point into the same universe as the receiver, so the type scheme normally does not change. An obvious exception is `this`, whose peer is a peer but not `this` itself. Another one are *readonlyS* receivers: Like above, using *readonlyS* for the resulting expression would make almost anything assignable to it, although it does not stand for a *readonlyS* variable. Therefore, $\mathit{readonlyS}'$ has to be used.

- *readonlyS*, *rootS*: What has been written about *primitiveS* variables also applies to *readonlyS* and *rootS* variables.

We can now move on to the type rules themselves. Propositions of the kind "$\vdash e : TS$" mean that the expression e is type correct and has type scheme TS. Similarly, "$\vdash s$" means that the statement s is type correct. A program is type correct iff all of its statements are type correct. A program is type correct regarding some annotations iff it is type correct and all variables have the type schemes specified by the annotations.

Note that the notation of the rules is somewhat informal: Some relevant conditions (e.g. that a field is static) are only expressed by the name or the accompanying text of a rule. Also be aware that some uninteresting rules are omitted, e.g. those for arithmetic expressions, if-then-else statements, and loops.

5.2 Rules for Primitive Expressions

(`null`) $\vdash \texttt{null} : \mathit{nullS}$

(`this`) $\vdash \texttt{this} : \mathit{thisS}$

(`super`) $\vdash \texttt{super} : \mathit{thisS}$

These rules are straightforward: `null` always has type scheme *nullS*, `this` type scheme *thisS*, and `super` also type scheme *thisS*, since it refers to the same object as `this`.

(String literal)
$$\frac{TS \in \{\mathit{repS}, \mathit{peerS}, \mathit{rootS}\}}{\vdash \texttt{"..."} : TS}$$

Here `"..."` stands for an arbitrary string literal. Its type scheme is freely selectable from the given set. Using *readonlyS* for new objects (including string literals) is generally not allowed; the reason is not that it would break anything to do so, but rather that it would not make much sense.

(Local variable in instance context)
$$\frac{TS \in \mathit{TypeSchemes}}{\vdash v : TS}$$

(Local variable in static context)
$$\frac{TS \in \{\mathit{primitiveS}, \mathit{readonlyS}, \mathit{rootS}\}}{\vdash v : TS}$$

In an instance context, the type scheme of a local variable can be any of the regular schemes. In static contexts, there is no `this` object, so only type schemes whose interpretation does not depend on it are allowed there. Altough this is not expressed by the rules, it is naturally necessary that the same type scheme is used for *all* occurrences of any given variable.

(Instance field)
$$\frac{TS \in \mathit{TypeSchemes}}{\vdash f : TS}$$

(Static field)
$$\frac{TS \in \{\mathit{primitiveS}, \mathit{readonlyS}, \mathit{rootS}\}}{\vdash f : TS}$$

Instance and static fields behave like local variables in instance and static contexts, respectively.

5.3 Rules for Composite Expressions

(Instance field access)
$$\frac{\vdash e : TS_1 \quad \vdash f : TS_2}{\vdash e\texttt{.}f : TS_1 * TS_2}$$

(Static field access)
$$\frac{\vdash f : TS}{\vdash C\texttt{.}f : TS}$$

The type scheme resulting from accessing an instance field is obtained by combining the type schemes of the receiver expression and of the field. For static fields, there is no receiver object.

(Array component access)
$$\frac{\vdash e_1 : TS_1 \quad \text{for all } i \in \mathbb{N}\ (\vdash \texttt{[]}_{t,i} : TS_2)}{\vdash e_1\texttt{[}e_2\texttt{]} : TS_1 * TS_2}$$

Structurally, the rule for array component accesses is similar to the one for instance field accesses. Type t here is the component type of the array denoted by expression e_1. Things are complicated slightly by the fact that the value of expression e_2 cannot in general be determined statically. Thus, the index i is universally quantified, which means that all array components of a type must have the same type scheme.

(Instance method call)
$$\frac{\vdash e : TS \quad \vdash e_i : TS_i \quad \vdash \mathit{par}_i(m) : TS'_i \quad \vdash \mathit{res}(m) : TS'' \\ TS_i \preceq_S TS * TS'_i}{\vdash e\texttt{.}m\texttt{(}e_1, \ldots, e_n\texttt{)} : TS * TS''}$$

(Static method call)
$$\frac{\vdash e_i : TS_i \quad \vdash \mathit{par}_i(m) : TS'_i \quad \vdash \mathit{res}(m) : TS'' \\ TS_i \preceq_S TS'_i}{\vdash C\texttt{.}m\texttt{(}e_1, \ldots, e_n\texttt{)} : TS''}$$

Here n denotes the number of parameters of method m; $i = 1, \ldots, n$; $\mathit{par}_i(m)$ stands for the ith formal parameter of m, and $res(m)$ for its result. Basically, both rules require that the actual parameters are assignable to the formal parameters, and both have the overall type scheme be determined by the one of the method result. In an instance method call, the formal parameters and method result have to be interpreted in relation to the receiver. Of course, for `void` methods the method call is really a statement, and as such does not have a type scheme.

(Allocation)
$$\frac{TS \in \{repS, peerS, rootS\} \quad \vdash e_i : TS_i \quad \vdash par_i(C) : TS'_i \qquad TS_i \preceq_S TS * TS'_i}{\vdash \texttt{new } C\texttt{(}e_1, \ldots, e_n\texttt{)} : TS}$$

(Array allocation)
$$\frac{TS \in \{repS, peerS, rootS\}}{\vdash \texttt{new } t\texttt{[]} : TS}$$

Like a string literal, an object allocation can have any non-primitive regular type scheme except *readonlyS*. The constructor invocation behaves like an ordinary method call on the newly created object. An array allocation is like any other allocation where the constructor does not have parameters.

(Conditional expression)
$$\frac{TS \in TypeSchemes \quad \vdash e_2 : TS_2 \quad \vdash e_3 : TS_3 \qquad TS_2 \preceq_S TS \quad TS_3 \preceq_S TS}{\vdash \texttt{(}e_1 \texttt{ ? } e_2 \texttt{ : } e_3\texttt{)} : TS}$$

The type scheme of a conditional expression can be any regular type scheme to which the type schemes of both branches are assignable.

(Assignment)
$$\frac{\vdash e_1 : TS_1 \quad \vdash e_2 : TS_2 \quad TS_2 \preceq_S TS_1}{\vdash e_1 \texttt{ = } e_2 : TS_1}$$

Unsurprisingly, an assignment is valid only if the type scheme of the right hand side is assignable to the one of the left hand side.

5.4 Rules for Statements

(Return)
$$\frac{\vdash res(m) : TS_1 \quad \vdash e : TS_2 \quad TS_2 \preceq_S TS_1}{\vdash \texttt{return } e}$$

The only relevant thing a return statement does is assigning a value to the method result. Thus, this rule is very similar to the one for assignments.

(Catch)
$$\frac{\vdash e : readonlyS}{\vdash \texttt{catch(}E\ e\texttt{)}}$$

Exceptions can be propagated to other contexts in a way which completely bypasses the usual control mechanisms, namely the rules for assignments and method calls. Several ways to handle this are discussed in [DM04]; the authors recommend either forcing all exception objects to have type scheme *rootS*, or forcing exceptions to be *caught* only as *readonlyS*. We adopt the latter approach for its simplicity.

6 Implementation

The analysis outlined in the previous chapters has been implemented as a part of the KeY tool. Section 6.1 gives an overview of the structure of the implementation. Afterwards, Section 6.2 describes a few experiences with testing it.

6.1 Design

The two basic steps which an implementation of *#universes* must perform have been defined in Section 4.2:

1. *Ensure that π does not read the initial value of any local reference variable.* This is implemented as a check whether all local reference variables occurring in π are also *declared* within π. If this is the case, Java does not allow them to be read before they have been assigned a new value.

2. *Ensure that π is type correct regarding $annotate_{R,P,F}$.* If the type schemes of all expressions would be dictated either by $annotate_{R,P,F}$ or by the type rules, then this could be accomplished by a simple type checker algorithm which walks through the abstract syntax tree and decides locally for each node whether it fulfills the type rules or not. However, the type schemes for local variables, allocation expressions, and string literals are not fixed by either of them. These type schemes can therefore be chosen freely; the analysis has to find out whether it is *possible* to do so in a consistent way such that the program as a whole complies to the type rules. In other words, applying the type rules to the program yields a *set of constraints*, which contain placeholders representing the type schemes of local variables, allocation expressions and string literals; and the question to be answered is that of its *satisfiability.* So the task of ensuring type correctness is further divided into the following two sub-steps:

    a) *Generate the proper constraints for π.* Basically, this is done by walking through the abstract syntax tree and applying the appropriate type rule at each node, which in some cases results in a constraint. In the end, all of these constraints are returned.
    This process has to be applied to *all* code that *could* potentially be executed when running π. To do so, method calls are expanded to the respective implementations on-the-fly. In case there are several possible implementations for a method call because of dynamic binding, all of them are expanded.

    b) *Try to solve the constraints.* This is implemented as a simple backtracking algorithm, which performs an exhaustive search of the solution tree but does not descend into clearly unsatisfiable branches. As such, its worst-case execution time increases exponentially with the number of local reference variables, allocation expressions, and string literals in the analysed program, including all expanded method bodies. However, there is hope that the average behaviour will be clearly better, since it seems likely that in many practical cases the constraints are simple enough so that a lot of combinations can be ruled out quickly.

Example. The following proof obligation specifies that the method `next` of our iterator must preserve the encapsulation of the list:

$$\begin{aligned}&\mathit{protect}[\{\texttt{head}\}|\{\texttt{next}\}|\{\texttt{pos}\}]\\ &\quad\rightarrow [\{\texttt{Ano.r = Ano.it.next();}\}]\ \mathit{protect}[\{\texttt{head}\}|\{\texttt{next}\}|\{\texttt{pos}\}]\end{aligned}$$

The code that has to be analysed in order to prove this is repeated below. The constraints which are generated in this process are attached to the respective lines as comments; the placeholders for unknown type schemes are denoted as $TS(\cdot)$, the subscheme relation as $<$.

```
Ano.r = Ano.it.next();//rootS ∗ TS(res(next)) < rootS

public Object next() {
    Object posData = pos.data;
                        //(thisS ∗ readonlyS) ∗ rootS < TS(posData)
    pos = pos.next;     //readonlyS ∗ peerS < readonlyS
    return posData;     //TS(posData) < TS(res(next))
}
```

The single solution in this case is $TS(\mathit{res}(\texttt{next})) = TS(\texttt{posData}) = \mathit{rootS}$.

6.2 Practical Experiences

The implementation has so far been tested on two different examples, the first of which is our well-known linked list. This does not pose any significant problem to the analysis. The same should be true for other, larger examples, as long as they are similarly self-contained.

The second test-case is `java.util.TreeMap`, a red-black-tree implementation out of the Java library. Unlike the linked list, this class has a significant size on its own, and—more critically—it also contains numerous calls to methods of other library classes. Firstly, such dependencies are inconvenient since KeY does not normally provide the implementations for library methods, so each of them has to be made available by hand. Secondly, many dependencies mean that the result of an analysis run also depends on a lot of code, and can thus be invalidated by a change anywhere in this code. And thirdly, this naturally leads to an amount of type scheme placeholders which poses a challenge to the exponential-time solving algorithm; depending on the analysed method, the needed time varied between no noticeable delay at all and several hours.

7 Conclusion

7.1 Summary

This work has presented a static analysis for checking encapsulation properties formulated in JavaDL which is based on the Universe Type System. This has been achieved by introducing a tailor-made predicate into JavaDL that captures the precise meaning of universe type annotations. The preservation of this predicate by a program can then be checked by applying the type rules of the Universe Type System. The analysis is integrated into the JavaDL sequent calculus as a special rule, whose soundness has been justified based on the formal foundation of the Universe Type System.

The analysis has been implemented within the KeY prover. It is called by applying the above mentioned rule, and runs fully automatically. Internally, it generates a set of typing constraints and checks their satisfiability, which is quick for smaller, more self-contained programs, but may take a significant amount of time if the code under test has a lot of dependencies on other code fragments. If the analysis succeeds, the proof branch can be closed immediately; if it fails, nothing is said about the validity of the current goal, and the proof must be continued by other means.

7.2 Future Work

So far, the following areas have been identified for possible future improvements:

- A more sophisticated technique could be employed for solving the constraints. For example, Prolog could be used, as it is done in [Kel05] to deal with the closely related problem of statically inferring all possible universe annotations for a program.
- In case of failure, the analysis could provide some assistance for investigating the reason. This would be helpful at least to those users who have an understanding of the Universe Type System.
- Programs could be allowed to contain explicit universe annotations. These would improve performance by reducing the solution space for the solving algorithm. Also, they could potentially be used to modularise the checking process: If the annotations of a method signature and of all fields which can be accessed within the method were fixed between analysis runs, the method body would only have to be checked once.

Bibliography

[ABB$^+$05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[Boy01] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[CPN98] Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, October 1998.

[DM04] Werner Dietl and Peter Müller. Exceptions in ownership type systems. In Erik Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.

[DM05] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[Häc04] Thomas Hächler. Statische Felder im Universe Type System. Semesterarbeit, Software Component Technology Group, ETH Zürich, 2004.

[Kel05] Nathalie Kellenberger. Static universe type inference. Master thesis, Software Component Technology Group, ETH Zürich, September 2005.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06, Department of Computer Science, Iowa State University, 1998.

[MPH01] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[Rot05] Andreas Roth. Specification and verification of encapsulation in Java programs. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005, Proceedings*, volume 3535 of *Lecture Notes in Computer Science*, pages 195–210. Springer, June 2005.

[VB01] Jan Vitek and Boris Bokowski. Confined types in Java. *Software—Practice and Experience*, 31(6):507–532, 2001.