# Universe Types for Topology and Encapsulation

Dave Cunningham[1], Werner Dietl[2], Sophia Drossopoulou[1], Adrian Francalanza[3], Peter Müller[2], and Alexander J. Summers[1]

[1] Imperial College London
{david.cunningham04,s.drossopoulou,alexander.j.summers}@imperial.ac.uk
[2] ETH Zurich
{Werner.Dietl,Peter.Mueller}@inf.ethz.ch
[3] University of Southampton
af1@ecs.soton.ac.uk

**Abstract.** The Universe Type System is an ownership type system for object-oriented programming languages that hierarchically structures the object store; it is used to reason modularly about programs.
We formalise Universe Types for a core subset of Java in two steps: We first define a Topological Type System that structures the object store hierarchically into an ownership tree, and demonstrate soundness of the Topological Type System by proving subject reduction. Motivated by concerns of modular verification, we then present an Encapsulation Type System that enforces the owner-as-modifier discipline; that is, that object updates are initiated by the owner of the object.
The contributions of this paper are, firstly, an extensive type-theoretic account of the Universe Type System, with explanations and complete proofs, and secondly, the clean separation of the topological from the encapsulation concerns.

## 1 Introduction

Imperative object-oriented programming languages, such as C++, Java and C#, use references to build object structures and share state. *Aliasing* allows multiple references to the same object and gives much of the power of object-oriented programming. However, it makes several other programming aspects more difficult, including reasoning about programs, garbage collection and memory management, code migration, parallelism and the analysis of atomicity.

To address these issues, different ownership type systems have been proposed: ownership types [10, 9, 6], ownership domains [1], Universe Types [26, 15] and similar other type systems [18, 2]. All have in common that they organise the heap as an *ownership tree* where each object is *owned* by at most one other object. It is common practice to depict ownership through a *box* in an object graph, where all objects that share the same owner are within the box of the owning object. For example, in Figure 1 the dashed box around object 1 of class Bag indicates that it owns objects 2 and 13, while object 2 of class Stack owns objects 3, 4 and 5. Objects that are not owned by any other object, such as 1, 6 and 9 are contained within the outermost dashed box labelled root, which gives us a tree.

However, different ownership type systems enforce different *encapsulation policies*, that is, put different restrictions on what references between objects might exist or limit the use of certain references. Ownership types enforce the owner-as-dominator policy, guaranteeing that every reference chain from an object in the root context to an object *o* goes through *o*'s owner. Ownership domains allow the declaration of flexible encapsulation policies by special *link* declarations. Universe Types enforce the owner-as-modifier discipline that ensures that all modifications of an object are initiated by the object's owner.

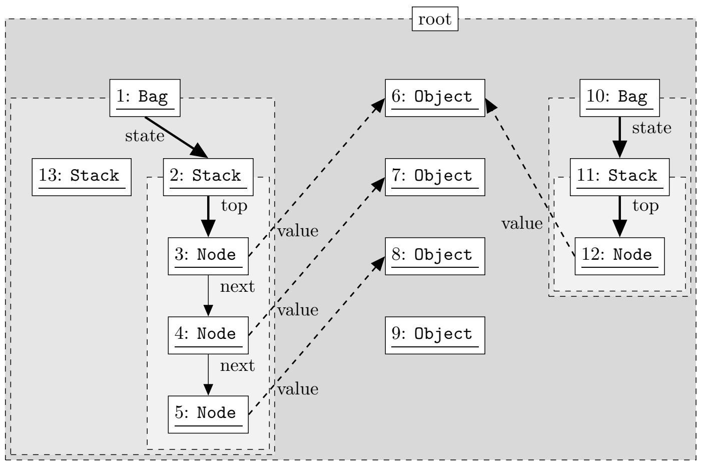

**Fig. 1.** Depicting object ownership and references in a heap.

The hierarchic heap topology and the different encapsulation policies can be exploited in different ways: Andreae et al. [2] speed up garbage collection, because, if the owner-as-dominator policy is enforced, as soon as an owner is unreachable, all owned objects are unreachable, too. Flanagan et al. [18] and Boyapati et al. [5] can guarantee that a program will not have races, because locking an object implicitly locks all owned objects. Cunningham et al. [11] show that Universe Types can be used to guarantee race free programs. Clarke et al. [9] use ownership types to calculate the effects of computations, and thus determine when they do not affect each other. Banerjee et al. [3] use the owner-as-dominator policy to prove representation independence of data structures. Müller et al. [27] use the hierarchic topology for defining modular verification techniques of object invariants.

Universe Types [26, 15] were developed to support modular reasoning about programs. The type system is one of the simplest possible in the family of ownership type systems. There are three *Universe Modifiers*: rep, peer and any, which denote the relative placement of objects in the ownership hierarchy. The qualifier rep (short for representation) expresses that the object is owned by the currently active object, while peer expresses that the object has the same owner as the currently active object. The qualifier any abstracts over the object's position in the hierarchy and does not give any static information about the owner.

```
class Bag {
   rep Stack state;
   impure void add(any Object o) { this.state.push(o); }
   pure Bool isEmpty() { return (this.state.top == null); }
}

class Stack {
   rep Node top;
   impure void push(any Object o) {
      this.top := new rep Node(o, this.top);
   }
   impure any Object pop() {
      any Object o := null;
      if (this.top != null) {
         o := this.top.value;
         this.top := this.top.next;
      }
      return o;
   }
}

class Node {
   any Object value;
   peer Node next;
   Node(any Object v, peer Node n) { value := v; next := n; }
}
```

**Fig. 2.** Code augmented with Universe Modifiers.

An example appears in Figure 2. The Stack field state in class Bag is declared as rep and indeed, we see that in Figure 1 the Stack objects 2 and 11 are owned by Bag objects 1 and 10, respectively; similarly, in class Node the field next is declared as peer, and indeed the Node objects 3, 4 and 5 have the same owner. The field value in class Node is declared as any Object and can therefore refer to any object in the hierarchy. In that sense, the Universe Modifier any plays a similar role to that played by the any ownership parameter in [24].

The code in Figure 2 also augments methods with the keywords pure, qualifying methods without side-effects, and impure, for methods that might affect the state of the program; these purity annotations are used to guarantee encapsulation.

In contrast with those ownership type systems that enforce the owner-as-dominator policy [1, 6, 10, 2], Universe Types do not restrict references into the boxes, as long as they are carried out through any references. Thus, a reference from 3 to 12 would be legal through the field value, because this field has the type any Object.

Nevertheless, Universe Types can be used to impose encapsulation, in the sense of guaranteeing that the state of an object can only be modified when the object's owner is one of the currently active objects. This owner-as-modifier discipline [15] is checked by the type system by forbidding all field updates on any references and requiring that only pure methods can be called when the receiver is an any reference. The owner-as-modifier discipline supports modular reasoning by restricting the objects whose invariants can be broken [23, 27].

In this paper we give a formal, type theoretical description of Universe Types. We distinguish the Topological Type System from the Encapsulation Type System, and describe the latter on top of the Topological System. The reasons for this distinction are:

- the distinction clarifies the rationale for the type systems;
- some systems, such as those required for race detection, atomicity and deadlock detection, only require the topological properties;
- one might want to add an encapsulation part onto a different Topological Type System or use different encapsulation policies.

Universe Types were already introduced and proven sound [26], but the description was in terms of proof theory, rather than the type theoretic machinery we adopt in this report; they were further described in the context of JML [15, 22]. Extensions to Universe Types, such as the addition of generics [14], already use the type theoretic approach, but do not separate topology and encapsulation.

This paper thus aims to fill a gap in the Universe Types literature, by giving a full type theoretical account of the basic type system, proofs, sufficient examples, explanations, and elucidate the distinction between the Topological and the Encapsulation System. We describe Universe Types (UT) for a small Java-like language, which contains classes, inheritance, fields, methods, dynamic method binding and casts.

The rest of the paper is organised as follows: we introduce our base language in Section 2, followed by a discussion of Universe Modifiers and owners in Section 3. In Section 4 we give the operational semantics for our language. Section 5 presents the Topological Type System and states subject reduction. Section 6 covers the Encapsulation Type System. Section 7 discusses related work and Section 8 concludes. Finally, Appendix A gives supporting results and proofs.

## 2 Source Language

UT models a subset of Java supporting class definitions, inheritance, field lookup and update, method invocations and casting. On top of this core subset, we augment types with Universe Modifiers and method signatures with purity annotations.

*Universe Modifiers* were originally defined [26] as the set {rep, peer, readonly}. In this paper, we present a generalisation of this approach. Firstly, we replace the modifier readonly with the modifier any, since the name readonly applies only to the intentions of the Encapsulation System, but not to the Topological. Secondly, we extend the set of modifiers with two new values: self and lost, which do not occur in source programs but are key in the operation of the Topological Type System we will present in Section 5.

Universe Modifiers are attached to object references, and provide *relative* information about the location (that is, position in the heap topology) of the referred-to object with respect to the current object. More specifically:

- rep states that the referred object constitutes part of the current object's *direct representation*. Stated otherwise, the current object *owns* the object pointed to by the reference.
- peer states that the referred object constitutes part of the same representation to which the current object belongs. Stated otherwise, the current object and the object pointed to by the reference are *owned* by the same object.
- self is a specialisation of peer, referring to the current object.
- any does not provide information about the location of the referred object. It is a 'don't care' modifier: such a reference may refer to objects with arbitrary owners.
- lost does not provide information about the location of the referred object. It is a 'don't know' modifier: it is used when the type system cannot accurately describe the location of the object[1].

**Example 2.1 (Comparing any and lost)** *We illustrate the difference between any and lost using the example from Figure 2 and a reference node of type any Node. The field access node.value has type any Object because the declared type of field value uses the any modifier and therefore permits references to arbitrary objects; the field* doesn't care *about the location of the value object. Consequently, an update of value is statically safe regardless of the owners of the receiver and the right-hand side. In particular, the update node.value := y is legal provided that y's class is a subclass of Object. Conversely, the field access node.next has type lost Node because the declared type of the field requires that the next node and the current node have the same owner. We do not statically know the owner of node as its Universe Modifier is any. Hence, we cannot express that the next field has the same owner. Since we* don't know *the accurate ownership information for node.next, the field update node.next := z is potentially unsafe and has to be rejected by the type system, as we cannot ensure the topology of the heap would be preserved.*

[1] As we said earlier, Lu and Potter's any ownership parameter [24] is the counterpart to our any Universe Modifier. Their unknown ownership parameter corresponds to our lost Universe Modifier—however, the main motivation for unknown is to preserve effective ownership rather than topology. Furthermore, our Universe Modifiers any and lost introduce a form of existential types [25].

The example above also illustrates that lost variables cannot be assigned meaningful values, therefore our language does not permit any explicit occurrences of lost in the program. Analogously, variables with modifier self can only be aliases of this and so do not add expressibility. For these reasons, when writing source programs we only make use of Universe Modifiers from the set $\{\mathsf{rep}, \mathsf{peer}, \mathsf{any}\}$.

The syntax of our source language is given in Figure 3. We assume three countably infinite, distinct sets of names: one for classes, $c \in \mathit{Id}_c$, one for fields, $f \in \mathit{Id}_f$, and another for methods, $m \in \mathit{Id}_m$. A program $P$ is defined in terms of three partial functions, $\mathcal{F}$, $\mathcal{M}$ and $\mathcal{MBody}$, and a relation over class names $\leq_c$; these functions and relations are global. $\mathcal{F}$ associates field names accessible[2] in a class to types, $\mathcal{M}$ associates method names accessible in a class to method signatures and $\mathcal{MBody}$ associates method names accessible in a class to method bodies. The reflexive, transitive relation $\leq_c$ denotes subclassing.

Types, denoted by $t$, constitute a departure from standard Java. They consist of a pair, $u\,c$, where class names $c$ are preceded by Universe Modifiers $u$. Source types, denoted by $s$ are the subset of types $t$ which are allowed to occur in source programs; i.e., they feature only Universe Modifiers from the set $\{\mathsf{rep}, \mathsf{peer}, \mathsf{any}\}$. Method signatures also deviate slightly from standard Java. They consist of a triple, denoted as $p : s_1\ (s_2)$ where $s_2$ is the type of the (single) method parameter, $s_1$ is the return type of the method and $p$ is a purity annotation, ranging over the set $\{\mathsf{pure}, \mathsf{impure}\}$. An extension to multiple method parameters is straightforward because each argument can be type-checked independently.

Source expressions, denoted by $e$, are a standard subset of Java. They consist of the self reference this, parameter identifier x, the basic value null, new object creation new $s$, casting $(s)\ e$, field access $e.f$, field update $e.f := e$ and method invocation $e.m(e)$.

$$
\begin{aligned}
\mathcal{F} &: \mathit{Id}_c \times \mathit{Id}_f \rightharpoonup \mathit{SrcType}\\
\mathcal{M} &: \mathit{Id}_c \times \mathit{Id}_m \rightharpoonup \mathit{MethSig}\\
\mathcal{MBody} &: \mathit{Id}_c \times \mathit{Id}_m \rightharpoonup \mathit{SrcExpr}\\
\leq_c &: \mathit{Id}_c \times \mathit{Id}_c \rightarrow \mathit{Bool}
\end{aligned}
$$

$$
\begin{aligned}
e \in \mathit{SrcExpr} &::= \mathsf{this} \mid \mathsf{x} \mid \mathsf{null} \mid \mathsf{new}\ s \mid (s)\ e\\
&\quad\mid e.f \mid e.f := e \mid e.m(e)\\
u \in \mathit{Universe\ Modifiers} &::= \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{self} \mid \mathsf{any} \mid \mathsf{lost}\\
s \in \mathit{SrcType} &::= \mathsf{rep}\ c \mid \mathsf{peer}\ c \mid \mathsf{any}\ c\\
t \in \mathit{Type} &::= s \mid \mathsf{self}\ c \mid \mathsf{lost}\ c\\
\mathit{MethSig} &::= p : s\ (s)\\
p \in \mathit{Purity\ Tag} &::= \mathsf{pure} \mid \mathsf{impure}
\end{aligned}
$$

**Fig. 3.** Syntax of source programs. The arrow $\rightharpoonup$ indicates partial mappings.

[2] By "accessible", we mean those which are either defined in the class or inherited from a superclass. We do not formalise visibility; all declarations are implicitly public.

## 3 Universe Modifiers and Owners

Universe Types structure the heap as an ownership tree. Every object $a$ in a heap is owned by a single owner, $o$, which is either another object in the heap or the root of the ownership tree, $\mathsf{root}$. The *direct representation* of an object in a heap $h$ is defined to be all the objects it owns (i.e., is the owner of). Ownership is required to be acyclic. When we do not want to refer to a particular heap, we find it convenient to refer to objects as pairs with their owners $(a, o)$. For example, $(\mathsf{1}, \mathsf{root})$ and $(\mathsf{2}, \mathsf{1})$, meaning $\mathsf{1}$ owned by $\mathsf{root}$ and $\mathsf{2}$ owned by $\mathsf{1}$, respectively.

We recall that the Universe Modifiers $\mathsf{self}$, $\mathsf{peer}$ and $\mathsf{rep}$ are interpreted with respect to the current object and do not mean anything without this *viewpoint*. One can assign a Universe Modifier to an object $(a', o')$ with respect to another object $(a, o)$ using the judgement

$$(a, o) \vdash (a', o') : u \tag{1}$$

which is defined as the least relation satisfying the rules in Figure 4. It states that, from the point of view of $a$ (owned by $o$), $a'$ (owned by $o'$) has Universe Modifier $u$. The $\mathsf{lost}$ and $\mathsf{any}$ modifiers can always be assigned because they do not express any ownership information.

$$\frac{}{(a, o) \vdash (a, o) : \mathsf{self}}\ (\textsc{Self}) \qquad \frac{}{(\_, o) \vdash (\_, o) : \mathsf{peer}}\ (\textsc{Peer})$$

$$\frac{}{(a, \_) \vdash (\_, a) : \mathsf{rep}}\ (\textsc{Rep})$$

$$\frac{}{(\_, \_) \vdash (\_, \_) : \mathsf{lost}}\ (\textsc{Lost}) \qquad \frac{}{(\_, \_) \vdash (\_, \_) : \mathsf{any}}\ (\textsc{Any})$$

**Fig. 4.** Assigning Universe Modifiers to objects.

**Example 3.1 (Universe Modifiers and objects)** *In the heap shown in Figure 1, from the point of view of* $\mathsf{2}$ *(owned by* $\mathsf{1}$*) object* $\mathsf{3}$ *(owned by* $\mathsf{2}$*) has Universe Modifier* $\mathsf{rep}$ *that is*

$$(\mathsf{2}, \mathsf{1}) \vdash (\mathsf{3}, \mathsf{2}) : \mathsf{rep} \tag{2}$$

*Similarly, we can derive*

$$(\mathsf{3}, \mathsf{2}) \vdash (\mathsf{4}, \mathsf{2}) : \mathsf{peer} \tag{3}$$

*Each object can view itself as* $\mathsf{self}$ *or* $\mathsf{peer}$*:*

$$(\mathsf{2}, \mathsf{1}) \vdash (\mathsf{2}, \mathsf{1}) : \mathsf{self} \qquad (\mathsf{2}, \mathsf{1}) \vdash (\mathsf{2}, \mathsf{1}) : \mathsf{peer} \tag{4}$$

*Also, we can assign* $\mathsf{any}$ *to any object from any viewpoint using rule* (Any)*:*

$$(\mathsf{3}, \mathsf{2}) \vdash (\mathsf{6}, \mathsf{root}) : \mathsf{any} \qquad (\mathsf{2}, \mathsf{1}) \vdash (\mathsf{3}, \mathsf{2}) : \mathsf{any} \qquad (\mathsf{3}, \mathsf{2}) \vdash (\mathsf{4}, \mathsf{2}) : \mathsf{any} \tag{5}$$

## 3.1 Universe Ordering

We define the following reflexive ordering for Universe Modifiers $u \leq_u u'$:

$$u \leq_u u \qquad \mathsf{self} \leq_u \mathsf{peer} \leq_u \mathsf{lost} \qquad \mathsf{rep} \leq_u \mathsf{lost} \qquad \mathsf{lost} \leq_u \mathsf{any}$$

It states that peer and rep are smaller (more precise) than lost, self is similarly a specialisation of peer, and any is the least specific modifier. The 'don't care' modifier any is treated as more general than the 'don't know' modifier lost; this is because we want to be able to assign to an any field even those objects whose location cannot be expressed in the type system.

Lemma 3.2 states that the Universe ordering relation ($\leq_u$) is consistent with the judgement of Figure 4. Thus, any object that is assigned rep, peer or self can also be assigned Universe Modifier any or lost, and any object that is assigned self can be assigned Universe Modifier peer, as we have already seen in Example 3.1.

**Lemma 3.2 (Universe object judgements respect Universe ordering)**

$$\left.\begin{array}{l}(a,o) \vdash (a',o') : u \\ u \leq_u u'\end{array}\right\} \implies (a,o) \vdash (a',o') : u'$$

*Proof.* By a simple case analysis of $(a,o) \vdash (a',o') : u$. □

**Example 3.3 (Subtyping and Universe Modifiers)** *We illustrate how the Universe Modifiers of the fields in the classes of Figure 2 characterise the references in Figure 1. For instance, the* Stack *object 2 has the* rep top *field correctly assigned to* Node 3*, since from (2) above we know that 3 has Universe Modifier* rep *with respect to 2. In fact, this reference can only point to (*Node*) objects 3, 4 and 5 since these are the only objects owned by object 2. Similarly,* Node 3 *has the* peer next *field correctly assigned to* Node 4*, which is owned by the same owner as 3; see (3) above. Trivially, the* any value *field of* Node 3 *assigned to* Object 6 *also respects the Universe Modifier because of (5) above. It can however point to any object in the heap since any type t is a subtype of* any Object*.*

## 3.2 Viewpoint Adaptation

The ownership information given by Universe Modifiers is *relative* with respect to a particular viewpoint. To adapt Universe Modifiers from one viewpoint to another, we define an operation $u_1 \rhd u_2$ called *viewpoint adaptation* [14]. This operation takes two Universe Modifiers $u_1$ and $u_2$, and yields a new Universe Modifier, as defined in Figure 5. The resulting modifier can be intuitively described as follows: "if $a_1$ sees $a_2$ as $u_1$, and $a_2$ sees $a_3$ as $u_2$, then $a_1$ sees $a_3$ as $u_1 \rhd u_2$"[3]. If there is no modifier to explicitly describe this relationship, then the operation yields the modifier lost. For example, $\mathsf{rep} \rhd \mathsf{rep} = \mathsf{lost}$, since there is no modifier to explicitly express that a referred object is in the 'transitive

[3] For readers familiar with work on Ownership Types, this operation is vaguely analogous with the notion there of substitution.

| | | | | $u_2$ | | |
|---|---|---|---|---|---|---|
| | $u_1 \rhd u_2$ | self | peer | rep | any | lost |
| | self | self | peer | rep | any | lost |
| | peer | lost | peer | lost | any | lost |
| $u_1$ | rep | lost | rep | lost | any | lost |
| | any | lost | lost | lost | any | lost |
| | lost | lost | lost | lost | any | lost |

**Fig. 5.** Viewpoint adaptation.

representation' of the current object. Also note that the modifiers any and lost do not depend on a viewpoint. Therefore, if $u_2$ is any or lost, the result will again be any or lost, respectively.

In Lemma 3.4, we show that the intuition of $\rhd$ is sound with respect to the interpretation of Universe Modifiers as object ownership in a heap, that is, judgement (1). We also show how we can recover information from a Universe Modifier $u \rhd u'$, so long as it is not lost.

**Lemma 3.4 (Sound viewpoint adaptation)**

$$\left.\begin{array}{l}(a_1, o_1) \vdash (a_2, o_2) : u_1 \\ (a_2, o_2) \vdash (a_3, o_3) : u_2\end{array}\right\} \Longrightarrow (a_1, o_1) \vdash (a_3, o_3) : u_1 \rhd u_2$$

$$\left.\begin{array}{l}(a_1, o_1) \vdash (a_2, o_2) : u_1 \\ (a_1, o_1) \vdash (a_3, o_3) : u_1 \rhd u_2 \\ u_1 \rhd u_2 \neq \mathit{lost}\end{array}\right\} \Longrightarrow (a_2, o_2) \vdash (a_3, o_3) : u_2$$

*Proof.* By a case analysis of $u_1$ and $u_2$ and inspection of Figures 4 and 5. The argument for the second part depends essentially on the fact that, if there exists a $u_2$ such that $u_1 \rhd u_2 \neq$ lost then it is unique (in other words, $u_1 \rhd u_2 = u_1 \rhd u_2' \neq$ lost implies that $u_2 = u_2'$). This can be seen by inspection of Figure 5. □

**Example 3.5 (Viewpoint adaptation)** *From judgements (2) and (3) from Example 3.1, using Lemma 3.4, we derive*

$$(2, 1) \vdash (4, 2) : \mathit{rep} \tag{6}$$

*because rep $\rhd$ peer $=$ rep. Conversely, from (2) and (6), using Lemma 3.4 and rep $=$ rep $\rhd$ peer $\neq$ lost, we recover (3)*

$$(3, 2) \vdash (4, 2) : \mathit{peer}$$

## 4 Operational Semantics

We give the semantics of our Java subset in terms of a small-step operational semantics. We assume a countably infinite set of addresses, ranged over by $a$, $b$.

$$\begin{array}{rcl}
a, b \in \mathit{Addr} & : & \mathbb{N} \\
o \in \mathit{Owner} & ::= & a \mid \mathsf{root} \\
v \in \mathit{Value} & ::= & a \mid \mathsf{null} \\
\\
\mathit{flds} \in \mathit{Flds} & : & \mathit{Id}_f \rightharpoonup \mathit{Value} \\
\\
h \in \mathit{Heap} & : & \mathit{Addr} \rightharpoonup (\mathit{Owner} \times \mathit{Id}_c \times \mathit{Flds}) \\
\sigma \in \mathit{StackFrame} & : & (\mathit{Addr} \times \mathit{Value}) \\
\\
e \in \mathit{RunExpr} & ::= & v \mid \mathsf{this} \mid \mathsf{x} \mid \mathsf{frame}\ \sigma\ e \mid e.f \mid e.f := e \\
& & \mid e.m(e) \mid \mathsf{new}\ s \mid (s)\ e \\
E[\cdot] & ::= & [\cdot] \mid E[\cdot].f \mid E[\cdot].f := e \mid a.f := E[\cdot] \\
& & \mid E[\cdot].m(e) \mid a.m(E[\cdot]) \mid (s)\ E[\cdot]
\end{array}$$

**Fig. 6.** Syntax of runtime expressions.

At runtime, a value, denoted by $v$, may be either an address or null. Owners, denoted by $o$, can be either an address or the special owner root.

Runtime expressions are described in Figure 6. During execution, expressions may contain addresses as values; they may also contain the keyword this and the parameter identifier x. Thus, a runtime expression is interpreted with respect to a heap, $h$, which gives meaning to addresses, and a stack frame, $\sigma$, which gives meaning to the keyword this and parameter identifier x. Evaluation contexts $E$ define expressions with 'holes' [17]; they are used in the semantics to permit reductions to take place below the top level of the expression.

A heap is defined in Figure 6 as a partial function from addresses to objects. An object is denoted by the triple $(o, c, \mathit{flds})$. Every object has an immutable owner $o$, belongs to a class $c$, and has a state, *flds*, which is a mutable field map (a partial function from field names to values). In the remaining text we use the following heap operations:

$$\begin{array}{rlrl}
\mathbf{owner}(h, a) & \overset{\mathbf{def}}{=} h(a)\downarrow_1 & \mathbf{class}(h, a) & \overset{\mathbf{def}}{=} h(a)\downarrow_2 \\
\mathbf{fields}(h, a) & \overset{\mathbf{def}}{=} h(a)\downarrow_3 & h(a.f) & \overset{\mathbf{def}}{=} \mathbf{fields}(h, a)(f)
\end{array}$$

$$h[(a, f) \mapsto v] \overset{\mathbf{def}}{=} h\left[a \mapsto \big(\mathbf{owner}(h, a),\ \mathbf{class}(h, a),\ \mathbf{fields}(h, a)[f \mapsto v]\,\big)\right]$$

The first three operations extract the components making up an object, where $\downarrow_i$ is used for the $i$-th projection. The fourth operation is merely a shorthand notation for *field access* in a heap. The fifth operation is *heap update*, updating the field $f$ of an object mapped to by the address $a$ in the heap $h$ to the value $v$.

A stack frame $\sigma$ is a pair $(a, v)$, of an address and a value. The address $a$ denotes the currently active object referred to by this whereas $v$ denotes the value of the parameter x. We find it convenient to define the following operations on stack frames:

$$\sigma(\mathsf{this}) \overset{\mathbf{def}}{=} \sigma\downarrow_1 \qquad\qquad \sigma(\mathsf{x}) \overset{\mathbf{def}}{=} \sigma\downarrow_2$$

For evaluating method calls, we require to push and pop new address-value pairs onto the stack. To model this, runtime expressions also include the expression

$\mathsf{frame}\ \sigma\ e$, which denotes that the sub-expression $e$ is evaluated with respect to the inner stack frame $\sigma$.

We employ a further operation called *heap extension*, written $\mathbf{alloc}(h, \sigma, s)$, which extends a heap $h$ with a new mapping from a *fresh* address $a$ to a newly-initialised object of type $s$; it is defined by the following function:

$$\begin{aligned}
&\mathbf{alloc}(h, \sigma, u\, c) \stackrel{\mathbf{def}}{=} (h', a) \quad \textit{if}\ u \in \{\mathsf{rep}, \mathsf{peer}\}\\
&\ \text{where}\\
&\qquad a \notin \mathbf{dom}(h)\\
&\qquad h' = h[a \mapsto (o, c, \mathit{flds})]\\
&\qquad o = \begin{cases} \sigma(\mathsf{this}) & \textit{if}\ u = \mathsf{rep}\\ \mathbf{owner}(h, \sigma(\mathsf{this})) & \textit{if}\ u = \mathsf{peer} \end{cases}\\
&\qquad \mathit{flds} = \{f \mapsto \mathsf{null} \mid \mathcal{F}(c, f) = \_\}
\end{aligned}$$

The owner is initialised according to the Universe Modifier specified in the type $s$ and the current stack frame $\sigma$. The above function is partial: it is only defined for Universe Modifiers $\mathsf{rep}$ and $\mathsf{peer}$ since the owner of a new object cannot be determined if the Universe Modifier is $\mathsf{any}$. The values of the fields of class $c$, and all its superclasses, are initialised to $\mathsf{null}$.

Expressions $e$ are evaluated in the context of a heap $h$ and a stack frame $\sigma$. We define the small-step semantics

$$\sigma \vdash e, h \rightsquigarrow e', h' \tag{7}$$

in terms of the reduction rules in Figure 7. We will use $\rightsquigarrow^*$ to indicate a consecutive sequence of (zero or more) small-step reductions.

$$\frac{}{\sigma \vdash \mathsf{x}, h \rightsquigarrow \sigma(\mathsf{x}), h}\ (\textsc{rVar}) \qquad \frac{h' = h[(a, f) \mapsto v]}{\sigma \vdash a.f := v, h \rightsquigarrow v, h'}\ (\textsc{rAssign})$$

$$\frac{}{\sigma \vdash \mathsf{this}, h \rightsquigarrow \sigma(\mathsf{this}), h}\ (\textsc{rThis}) \qquad \frac{e = \mathcal{MBody}(\mathbf{class}(h, a), m)}{\sigma \vdash a.m(v), h \rightsquigarrow \mathsf{frame}\ (a, v)\ e,\ h}\ (\textsc{rCall})$$

$$\frac{(h', a) = \mathbf{alloc}(h, \sigma, s)}{\sigma \vdash \mathsf{new}\ s, h \rightsquigarrow a, h'}\ (\textsc{rNew}) \qquad \frac{\sigma \vdash e, h \rightsquigarrow e', h'}{\sigma \vdash E[e], h \rightsquigarrow E[e'], h'}\ (\textsc{rEvalCtx})$$

$$\frac{h, \sigma \vdash a : s}{\sigma \vdash (s)\ a, h \rightsquigarrow a, h}\ (\textsc{rCast}) \qquad \frac{\sigma' \vdash e, h \rightsquigarrow e', h'}{\sigma \vdash \mathsf{frame}\ \sigma'\ e, h \rightsquigarrow \mathsf{frame}\ \sigma'\ e', h'}\ (\textsc{rFrame1})$$

$$\frac{}{\sigma \vdash a.f, h \rightsquigarrow h(a.f), h}\ (\textsc{rField}) \qquad \frac{}{\sigma \vdash \mathsf{frame}\ \sigma'\ v, h \rightsquigarrow v, h}\ (\textsc{rFrame2})$$

**Fig. 7.** Small-step operational semantics.

Most of the rules in Figure 7 are straightforward. When creating a new object (rNew) the $\mathbf{alloc}(h, \sigma, s)$ function defined above determines the new heap $h'$ and fresh address $a$. In (rCall), a method call creates a new stack frame $\sigma'$

to evaluate the body of the method, where $\sigma'(\mathsf{this})$ is the receiver object $a$ and $\sigma'(\mathsf{x})$ is the value passed by the call, $v$. Once a frame evaluates to a value $v$, we discard the sub-frame and return to the outer frame, as shown in (rFrame2). We also note that the rule (rEvalCtx) dictates the evaluation order of an expression, based on the evaluation contexts $E[\cdot]$ defined in Figure 6. The type judgement in rule (rCast) expresses that the object at address $a$ has type $s$; see Subsection 5.3. Note that the source expression null is identical to the value null, which makes a special rule dispensable.

**Example 4.1 (Runtime execution)** *Let $h$ denote the heap depicted in Figure 1 and the current stack frame be $\sigma = (2, 9)$. Then, if we consider the program in Figure 2, and execute the expression* **this** *.push(7) we get the following reductions*[4]*, where the rule names on the side indicate the main reduction rule applied to derive the reduction. We simplify the example slightly, by not mentioning the use of context rules (*rEvalCtxt*) and (*rFrame1*).*

$$
\begin{array}{lr}
\sigma \vdash \textbf{this}\,.\mathit{push(7)}, h \rightsquigarrow \mathit{2.push(7)},\ h & (\textsc{rThis})\\
\quad \rightsquigarrow \mathit{frame}\ \sigma'\ \ \textbf{this}\,.\mathit{top}{:=}\ \textbf{new rep}\ \mathit{Node(x,}\ \textbf{this}.\mathit{top)},\ h & (\textsc{rCall})\\
\quad \rightsquigarrow \mathit{frame}\ \sigma'\ \ \mathit{2.top}{:=}\ \textbf{new rep}\ \mathit{Node(x,}\ \textbf{this}.\mathit{top)},\ h & (\textsc{rThis})\\
\quad \rightsquigarrow \mathit{frame}\ \sigma'\ \ \mathit{2.top}{:=}\ \textbf{new rep}\ \mathit{Node(7,}\ \textbf{this}.\mathit{top)},\ h & (\textsc{rVar})\\
\quad \rightsquigarrow \mathit{frame}\ \sigma'\ \ \mathit{2.top}{:=}\ \textbf{new rep}\ \mathit{Node(7,\ 2.top)},\ h & (\textsc{rThis})\\
\quad \rightsquigarrow \mathit{frame}\ \sigma'\ \ \mathit{2.top}{:=}\ \textbf{new rep}\ \mathit{Node(7,\ 3)},\ h & (\textsc{rField})\\
\quad \rightsquigarrow \mathit{frame}\ \sigma'\ \ \mathit{2.top}{:=}\ \mathit{14},\ h' & (\textsc{rNew})\\
\quad \rightsquigarrow \mathit{frame}\ \sigma'\ \ \mathit{14},\ h'[(\mathit{2}, \mathsf{top}) \mapsto \mathit{14}] & (\textsc{rAssign})\\
\quad \rightsquigarrow\ \ \mathit{14},\ h'[(\mathit{2}, \mathsf{top}) \mapsto \mathit{14}] & (\textsc{rFrame2})
\end{array}
$$

*where $\sigma' = (2, 7)$, $h' = h \uplus \{14 \mapsto (2, \mathit{Node}, \{\mathsf{value} \mapsto 7,\ \mathsf{next} \mapsto 3\})\}$, and 14 is a fresh address in the heap $h$.*

## 5 Topological System

In this section, we define the Topological Type System for UT. The formalism is based on earlier work [26, 15], but has some differences: as we said in the introduction, we focus here on the hierarchical topology imposed by Universe Types, but do not enforce the *owner-as-modifier discipline* at this stage—this is dealt with in Section 6. The main result of this section is Topological Subject Reduction, stating that a type assigned to an expression and the ownership hierarchical heap structure are preserved during execution.

[4] In order to follow the Java code of Figure 2, the reductions use an object constructor that immediately initialises values to the parameters passed. This is more advanced than the simpler new construct considered in our language, which initialises all the fields of a fresh object to null. These details are however orthogonal to the determination of the owner of the object upon creation, which is the relevant issue for our work. Similarly, we return the value of the method body even if the method is declared to be **void**.

### 5.1 Subtyping and Viewpoint Adaptation

As was already stated in Section 2, types, $t$, are made up of two components: a Universe Modifier $u$ and a class name $c$. Using the Universe ordering $\leq_u$ of Section 3.1 and subclassing $\leq_c$, we define the subtype relation as:

$$u\ c \leq u'\ c' \overset{\textbf{def}}{=} u \leq_u u' \text{ and } c \leq_c c' \tag{8}$$

We extend $\triangleright$ defined in Section 3.2 for Universe Modifiers, to an operator on a Universe Modifier and a type, and that produces a type, denoted by $u \triangleright t$:

$$u \triangleright (u'\ c) \overset{\textbf{def}}{=} (u \triangleright u')\ c$$

We use this auxiliary operator whenever we need to change the viewpoint of the types.

### 5.2 Static Types

We type-check UT source expressions with respect to a type environment $\Gamma$, which keeps type information for this and the method parameter x. The types in a method signature are meant to be interpreted with respect to this, the currently active object. We assign the self Universe Modifier to $\Gamma$(this) when type-checking method bodies and note that self $\triangleright\ u = u$ for all $u$.

The use of a specific self Universe Modifier is a variation from previous models of the Universe Type System [15, 14, 26] and of other ownership type systems. In the work on Universe Types, the expression this was treated separately, viewpoint adaptation was omitted for access through this, and additional checks had to be made to ensure the protection of the representation of an object. This special treatment of the this expression can also be compared to the *static visibility constraint* of Ownership Types [10], which ensures that a type that contains rep is only accessible by this. Even when not enforcing the static visibility constraint, the this parameter in a type needs to be treated specially upon type application [1, 4]. The use of a special self ownership modifier makes the special role of the current object more explicit, while at the same time simplifying the overall system[5]. For example, attempting to update the representation of another object using a peer reference results in lost ownership information, i.e., peer $\triangleright$ rep = lost and the update is forbidden. On the other hand, updating the representation of the current object preserves ownership information, i.e., self $\triangleright$ rep = rep and an update is allowed.

**Definition 5.1 (Type environment)** *A type environment $\Gamma$ consists of a pair of types, $(t, t')$, assigning types to the currently active object* this *and the parameter* x*, respectively. We define the following operations on $\Gamma$:*

$$\Gamma(\textit{this}) \overset{\textit{def}}{=} \Gamma{\downarrow_1} \qquad\qquad \Gamma(x) \overset{\textit{def}}{=} \Gamma{\downarrow_2}$$

[5] Note that in the works mentioned on Ownership Types [10, 1, 4], types such as $A$<this> or $A$<self> do *not* correspond with our Universe Modifier self (which indicates the current receiver): their types would instead be represented in our system by the type rep $A$.

The source expression type judgement takes the form

$$\Gamma \vdash e : t$$

denoting that expression $e$ has type $t$ with respect to the type environment $\Gamma$. Note that we do not restrict $t$ to source types; although only source types may be written explicitly in the program, an inferred Universe Modifier may well be $\mathsf{lost}$, for example. The judgement is defined as the least relation satisfying the rules given in Figure 8. We sometimes find it convenient to use the shorthand judgement notation

$$\Gamma \vdash e : u \qquad\qquad \Gamma \vdash e : c$$

whenever components of the type judgement are not important, that is $\Gamma \vdash e : u\ \_$ and $\Gamma \vdash e : \_\ c$, respectively. Most of the rules are standard, with the exception of the type rules (Field), (Assign) and (Call), which use the auxiliary operation $u \rhd t$ to *adapt* types from one viewpoint to another. For example, consider the rule for field lookup (Field). The first premise says that $e$ can be assigned class $c$, and that from the current point of view, $e$'s position in the heap topology can be described by Universe Modifier $u$. The second premise states that the field $f$ is declared in class $c$ as having source type $s$. Since the Universe Modifier of $s$ describes the location of the field with respect to the point of view of $e$, to assign a type for this field from the *current* point of view, we take into account $e$'s relative position; that is, we adapt the type $s$ with respect to $u$.

**Example 5.2 (Type viewpoint adaptation)** *If $\Gamma \vdash$ this.top : rep Node and field next in class Node has type peer Node, then using (*Field*), the dereference this.top.next has type rep $\rhd$ (peer Node) = rep Node, that is*

$$\Gamma \vdash \textit{this.top.next} : \textit{rep Node}$$

*Conversely, we use (*Assign*) to check that when*

$$\Gamma \vdash \textit{this.top} : \textit{rep Node} \qquad \textit{and} \qquad \Gamma \vdash \textit{new rep Node} : \textit{rep Node}$$

*then the assignment this.top.next := new rep Node respects the field type assigned to next in class Node. For this calculation we use*

$$\begin{aligned} \mathcal{F}(\textit{Node}, \textit{next}) &= \textit{peer}\ \mathsf{Node} \\ \textit{rep} \rhd \textit{peer} &= \textit{rep}\ \neq \textit{lost} \end{aligned}$$

The source expression type judgement allows us to define well-formed classes by requiring consistency between subclasses. In particular, we require that field types in a subclass match those in any superclasses in which the same fields are present (this requirement is trivially met if fields cannot be overridden), and method signatures in subclasses are specialisations of the signatures of overridden methods. In addition to the usual variance on argument and return types, we allow pure methods to override impure methods (but not the opposite). Furthermore, we require that method bodies are consistent with their signatures. A program $P$ is *well-formed* if all the defined classes are well-formed:

$$\frac{}{\Gamma \vdash \mathsf{null} : t}\,(\textsc{Null}) \qquad \frac{}{\Gamma \vdash \mathsf{x} : \Gamma(\mathsf{x})}\,(\textsc{Var}) \qquad \frac{}{\Gamma \vdash \mathsf{this} : \Gamma(\mathsf{this})}\,(\textsc{This})$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (s)\ e : s}\,(\textsc{Cast}) \qquad \frac{u \in \{\mathsf{rep}, \mathsf{peer}\}}{\Gamma \vdash \mathsf{new}\ u\ c : u\ c}\,(\textsc{New})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t' \\ t' \leq t\end{array}}{\Gamma \vdash e : t}\,(\textsc{Sub}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : u\ c \\ \mathcal{F}(c, f) = s\end{array}}{\Gamma \vdash e.f : u \rhd s}\,(\textsc{Field})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : u\ c \\ \mathcal{F}(c, f) = s \\ u \rhd s \neq \mathsf{lost}\ \_ \\ \Gamma \vdash e' : u \rhd s\end{array}}{\Gamma \vdash e.f := e' : u \rhd s}\,(\textsc{Assign}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : u\ c \\ \mathcal{M}(c, m) = \_ : s_r\ (s_x) \\ u \rhd s_x \neq \mathsf{lost}\ \_ \\ \Gamma \vdash e' : u \rhd s_x\end{array}}{\Gamma \vdash e.m(e') : u \rhd s_r}\,(\textsc{Call})$$

**Fig. 8.** Source type system.

**Definition 5.3 (Well-formed classes and programs)**

$$\frac{\begin{array}{l}\forall c'\ .\ (c \leq_c c' \ \wedge\ \mathcal{F}(c', f) = s) \Longrightarrow \mathcal{F}(c, f) = s \\ \forall c'\ .\ (c \leq_c c' \ \wedge\ \mathcal{M}(c', m) = p' : s'_r\ (s'_x)) \\ \qquad\qquad\qquad\qquad\qquad \Longrightarrow \mathcal{M}(c, m) = p : s_r\ (s_x) \\ \qquad\qquad \textit{where}\ \ s_r \leq s'_r \quad \textit{and} \quad s'_x \leq s_x \\ \qquad\qquad\ \ \textit{and}\ \ p' = \textit{pure} \Rightarrow p = \textit{pure} \\ \forall m\ .\ \mathcal{M}(c, m) = \_ : s_r\ (s_x) \ \Longrightarrow (\textit{self}\ c, s_x) \vdash \mathcal{MBody}(c, m) : s_r\end{array}}{\vdash c}\,(\text{WF}\textsc{Class})$$

$$\vdash P \iff (\forall c \in \mathit{Id}_c\ .\ \vdash c)$$

### 5.3 Runtime Types

We define a type system for runtime expressions. These are type-checked with respect to the stack frame $\sigma$, which contains actual values for the current receiver $\mathsf{this}$ and the parameter $\mathsf{x}$. Since runtime expressions also contain addresses, we also need to type-check them with respect to the current heap, so as to retrieve the class membership and owner information for addresses.

The runtime Universe Type System allows us to assign Universe Types to runtime expressions with respect to a particular heap $h$ and stack frame $\sigma$, through a judgement of the form

$$h, \sigma \vdash e : t$$

It is defined as the least relation satisfying the rules in Figure 9. Once again, we use the shorthand notation $h, \sigma \vdash e : u$ and $h, \sigma \vdash e : c$ whenever the other component of $t$ in the judgement is not important. In the rule (tAddr), the type of an address in a heap is derived from the class of the object and the Universe Modifier obtained using judgement (1) of Section 3. The three rules (tField), (tAssign) and (tCall) use viewpoint adaptation in the same way as their static-expression counterparts in Figure 8. The new rule (tFrame) also uses viewpoint adaptation to adapt the type of the sub-expression, obtained with respect to the local stack frame, to the current frame's viewpoint.

$$\frac{}{h,\sigma \vdash \mathsf{null} : t}\,(\textsc{tNull}) \qquad \frac{h,\sigma \vdash \sigma(\mathsf{x}) : t}{h,\sigma \vdash \mathsf{x} : t}\,(\textsc{tVar}) \qquad \frac{h,\sigma \vdash \sigma(\mathsf{this}) : t}{h,\sigma \vdash \mathsf{this} : t}\,(\textsc{tThis})$$

$$\frac{h,\sigma \vdash e : t}{h,\sigma \vdash (s)\ e : s}\,(\textsc{tCast}) \qquad \frac{\begin{array}{c} h,\sigma \vdash e : t' \\ t' \leq t \end{array}}{h,\sigma \vdash e : t}\,(\textsc{tSub}) \qquad \frac{u \in \{\mathsf{rep}, \mathsf{peer}\}}{h,\sigma \vdash \mathsf{new}\ u\ c : u\ c}\,(\textsc{tNew})$$

$$\frac{\begin{array}{c} \mathbf{class}(h, a) = c \\ (\sigma(\mathsf{this}), \mathbf{owner}(h, \sigma(\mathsf{this}))) \vdash (a, \mathbf{owner}(h, a)) : u \end{array}}{h,\sigma \vdash a : u\ c}\,(\textsc{tAddr})$$

$$\frac{\begin{array}{c} h,\sigma \vdash e : u\ c \\ \mathcal{F}(c, f) = s \end{array}}{h,\sigma \vdash e.f : u \rhd s}\,(\textsc{tField}) \qquad \frac{\begin{array}{c} h,\sigma \vdash e : u\ c \quad \mathcal{F}(c, f) = s \\ u \rhd s \neq \mathsf{lost}\ \_ \\ h,\sigma \vdash e' : u \rhd s \end{array}}{h,\sigma \vdash e.f := e' : u \rhd s}\,(\textsc{tAssign})$$

$$\frac{\begin{array}{c} h,\sigma \vdash e : u\ c \\ \mathcal{M}(c, m) = \_ : s_r\ (s_x) \\ u \rhd s_x \neq \mathsf{lost}\ \_ \\ h,\sigma \vdash e' : u \rhd s_x \end{array}}{h,\sigma \vdash e.m(e') : u \rhd s_r}\,(\textsc{tCall}) \qquad \frac{\begin{array}{c} h,\sigma' \vdash e : t \\ h,\sigma \vdash \sigma'(\mathsf{this}) : u \end{array}}{h,\sigma \vdash \mathsf{frame}\ \sigma'\ e : u \rhd t}\,(\textsc{tFrame})$$

**Fig. 9.** Runtime type system.

Lemma 5.4 shows that viewpoint adaptation respects the judgements of the runtime type system. The viewpoint adaptations of Lemma 5.4 trivially hold for the case $v = \mathsf{null}$ since rule (tNull) immediately yields the desired judgements. The proof is relegated to Appendix A.

**Lemma 5.4 (Determining the relative Universe Types of values)**

*(i) If $h, \sigma \vdash a : u\ \_$ and $h, (a, \_) \vdash v : t$ then $h, \sigma \vdash v : u \rhd t$*
*(ii) If $h, \sigma \vdash a : u\ \_$ and $h, \sigma \vdash v : u \rhd t$ and $u \rhd t \neq \mathsf{lost}\ \_$ then, for any value $v'$ we have $h, (a, v') \vdash v : t$*

**Example 5.5 (Relative viewpoints in a heap)** *In Figure 1, using* (2) *and* (3) *from Example 3.1 and rule (*tAddr*) we derive*

$$h, (2, \_) \vdash 3 : \mathit{rep\ Node} \qquad \text{and} \qquad h, (3, \_) \vdash 4 : \mathit{peer\ Node}$$

*From Lemma 5.4(i) we immediately derive*

$$h, (2, \_) \vdash 4 : \mathit{rep\ Node}$$

*Conversely, using $h, (2, \_) \vdash 3 : \mathit{rep\ Node}$, $h, (2, \_) \vdash 4 : \mathit{rep\ Node}$ as well as Lemma 5.4(ii), we can recover $h, (3, \_) \vdash 4 : \mathit{peer\ Node}$.*

We now have enough machinery to define well-formed addresses, heaps and stack frames (Definition 5.7). An address is well-formed in a heap whenever its owner is valid (that is, it is another address in the heap or $\mathsf{root}$) and the types of its fields respect the types of the fields defined in $\mathcal{F}$. A heap is well-formed, denoted as $\vdash h$, if transitive ownership always includes $\mathsf{root}$ (this implies that the ownership relation is acyclic, since each address has one owner and $\mathsf{root}$ has no owner) and all its addresses are well-formed. Finally, a stack frame is well-formed with respect to a heap if the receiver address it contains is defined in the heap (we make no requirements about the argument on the stack, since these are enforced where necessary by the type system). We use $\mathbf{owner}^{+}(h, o)$ to denote the transitive closure of $\mathbf{owner}(h, o)$.

**Definition 5.6 (Transitive ownership)**

$$\boldsymbol{owner}^{+}(h,o) \overset{def}{=} \begin{cases} \{\boldsymbol{owner}(h,o)\} \cup \boldsymbol{owner}^{+}(h, \boldsymbol{owner}(h,o)) & \mathit{if}\ o \neq \mathit{root} \\ \emptyset & \mathit{if}\ o = \mathit{root} \end{cases}$$

**Definition 5.7 (Well-formed addresses, heaps and stack frames)**

$$\frac{\begin{array}{l} \boldsymbol{owner}(h,a) \in (\boldsymbol{dom}(h) \cup \{\mathit{root}\}) \\ \boldsymbol{class}(h,a) = c \\ \forall f\ .\ \mathcal{F}(c,f) = s \implies h,(a,\_) \vdash h(a.f) : s \end{array}}{h \vdash a}\ (\text{WFA}\textsc{ddr})$$

$$\frac{\forall a\ .\ a \in \boldsymbol{dom}(h) \implies \begin{cases} \mathit{root} \in \boldsymbol{owner}^{+}(h,a) \\ h \vdash a \end{cases}}{\vdash h}\ (\text{WFH}\textsc{eap})$$

$$\frac{\sigma(\mathit{this}) \in \boldsymbol{dom}(h)}{h \vdash \sigma}\ (\text{WFS}\textsc{tack})$$

We conclude the subsection by showing the correspondence between the source type system and runtime type system. Lemma 5.8 below states that, with respect to a suitable stack frame $\sigma$, where $\sigma(\mathsf{this})$ and $\sigma(\mathsf{x})$ match the respective type assignments in $\Gamma$, a well-typed source expression is also a well-typed runtime expression.

**Lemma 5.8 (Source typing to runtime typing)**

$$\left.\begin{array}{l} \Gamma \vdash e : t \\ h,\sigma \vdash x : \Gamma(x) \\ h,\sigma \vdash \mathit{this} : \Gamma(\mathit{this}) \end{array}\right\} \implies h,\sigma \vdash e : t$$

*Proof.* By induction on the structure of the derivation $\Gamma \vdash e : t$, considering the last rule applied. Comparing the two type systems, all cases follow by straightforward induction except for the rules (Var) and (This). These are guaranteed by the conditions on $\sigma$. $\square$

### 5.4 Subject Reduction

In this subsection, we present the first main result of the paper. It states that if a well-typed runtime expression $e$ reduces with respect to a stack frame $\sigma$, and a well-formed heap $h$, then the resulting expression preserves its type (with respect to the new heap), and the resulting heap preserves its well-formedness as well as the well-formedness of the stack frame. Because our definition of well-formed heaps imposes strong topological constraints in correspondence with the Universe Modifiers in the program, this result means in particular that the implied topology is preserved during execution.

**Theorem 5.9 (Topological Subject Reduction)**
*For well-formed programs, the following property holds:*

$$\left.\begin{array}{l} \vdash h \\ h \vdash \sigma \\ h, \sigma \vdash e : t \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \vdash h' \\ h' \vdash \sigma \\ h', \sigma \vdash e' : t \end{array}\right.$$

*Proof.* We build up to this result by first proving a number of intermediary lemmas concerning the evolution of the heap under reduction and extracting object information from types (see Appendix A). The owner and class components of an object in a heap are immutable during execution (Lemma A.2). During execution, we never remove existing addresses from the heap (Lemma A.3). Earlier in Section 4, we discussed how reduction rules make use of two operations to update a heap in the form of $\mathbf{alloc}(h, \sigma, s)$ and $h[(a, f) \mapsto v]$. Lemma A.4 shows that the heap extension operation creates a new object with the requested type in the heap. Lemma A.6 states that under appropriate conditions, heap update and heap extension operations preserve heap well-formedness. Lemma A.7 states that the type judgement $h, \sigma \vdash a : u\ c$ implies that the class of $a$ in $h$ is a subclass of $c$ and that $a$ has Universe Modifier $u$ from the current viewpoint $\sigma(\mathsf{this})$. We relegate these five lemmas to Appendix A. The proof uses also Lemma 5.4 and Lemma 5.8 from Section 5.3. The main cases of the Subject Reduction proof are given in Appendix A. □

## 6 Encapsulation System

In Section 5, we showed how the *Topological* Type System guarantees that the topology of the objects in the heap agrees with the one described by the Universe Types. In this section, we enhance the Topological Type System and obtain the *Encapsulation* Type System. We show that the latter system guarantees the owner-as-modifier discipline [15], which localises the effects of execution in a heap with respect to the currently active object.

We prove two related theorems: the Encapsulation Theorem (6.8) guarantees that an encapsulated expression can only modify objects transitively owned by the owner of the current receiver, while the Owner-as-Modifier Theorem (6.14) guarantees that execution of an encapsulated expression starting from the initial configuration may update an object only when the object's owner is on the call stack. Notice that although related, the two theorems do not follow from each other.

In terms of our running example, the Encapsulation Theorem guarantees that execution of a method by receiver 13 can modify—at most—objects 2, 3, 4, 5 and 13. On the other hand, the Owner-as-Modifier Theorem guarantees that execution of an encapsulated expression starting from the initial configuration may modify 13 only while 1 is on the stack.

## 6.1 Encapsulation Types

For the subsequent discussion we find it convenient to define contexts $C[\cdot]$ which are generally used to describe the field updates and method calls present within an expression. These are more liberal than the evaluation contexts $E[\cdot]$ previously defined, which are used to specify where evaluation should next take place. For example, $\mathsf{x.f.m(\cdot)}$ is a $C[\cdot]$, but not an $E[\cdot]$ context. Like $E[\cdot]$ contexts, $C[\cdot]$ contexts, are restricted to not include frame expressions, which allows us to express relationships between the sequence of stack frames in the expression (e.g., see rule (Enc) in Definition 6.2 below).

**Definition 6.1 (Frame-free contexts)**

$$\begin{aligned} C[\cdot] ::= {} & [\cdot] \mid C[\cdot].f \mid C[\cdot].f := e \mid e.f := C[\cdot] \\ \mid {} & C[\cdot].m(e) \mid e.m(C[\cdot]) \mid (s)\ C[\cdot] \end{aligned}$$

We will write $\mathbf{pure}(c,\, m)$ to mean that $m$ is declared to be pure in $c$:

$$\mathbf{pure}(c,\, m) \stackrel{\mathbf{def}}{=} \mathcal{M}(c, m) = \mathsf{pure} : \_\ (\_)$$

The Encapsulation Type System imposes extra restrictions so as to enforce the owner-as-modifier discipline and to guarantee restrictions on the effect of method calls. We define an encapsulation judgement for expressions, $\Gamma \vdash_{\mathbf{enc}} e$, reflecting the expression restrictions needed to enforce the owner-as-modifier discipline. These restrictions state that for an expression $e$ to respect encapsulation, it can only assign to and call impure methods on the current object, on $\mathsf{rep}$ receivers or on $\mathsf{peer}$ receivers. To determine (conservatively) when a method is actually pure, we require a purity judgement for expressions $\Gamma \vdash_{\mathbf{pure}} e$. An expression $e$ is pure by this judgement if it never assigns to fields and *only* calls methods declared to be pure. We use this very strict notion of purity to simplify the rules. Weaker purity requirements [14, 30] suffice to enforce the owner-as-modifier discipline.

**Definition 6.2 (Purity and encapsulation for source expressions)**

$$\frac{\begin{array}{l}\Gamma \vdash e : t \\ \forall C, e_1, e_2, f \;.\; e \neq C[e_1.f := e_2] \\ \forall C, e_1, e_2, m \;.\; e = C[e_1.m(e_2)] \Longrightarrow \exists c \;.\; \Gamma \vdash e_1 : c \;\wedge\; \textit{\textbf{pure}}(c,\, m)\end{array}}{\Gamma \vdash_{\textit{\textbf{pure}}} e}\;(\textsc{Pure})$$

$$\frac{\begin{array}{l}\Gamma \vdash e : t \\ \forall C, e_1, e_2, f \;.\; e = C[e_1.f := e_2] \Longrightarrow \exists u \;.\; \Gamma \vdash e_1 : u \;\wedge \\ \hfill u \in \{\textsf{peer}, \textsf{rep}\} \\ \forall C, e_1, e_2, m \;.\; e = C[e_1.m(e_2)] \Longrightarrow \exists u, c \;.\; \Gamma \vdash e_1 : u\ c \;\wedge \\ \hfill (u \in \{\textsf{peer}, \textsf{rep}\} \vee \textit{\textbf{pure}}(c,\, m))\end{array}}{\Gamma \vdash_{\textit{\textbf{enc}}} e}\;(\textsc{Enc})$$

Note that if an expression is considered pure, it automatically respects encapsulation; i.e., $\Gamma \vdash_{\mathbf{pure}} e \Rightarrow \Gamma \vdash_{\mathbf{enc}} e$.

In terms of our example code (Figure 2), suppose $\Gamma = (\textsf{self Bag}, \textsf{any Object})$. Then we can derive $\Gamma \vdash_{\mathbf{enc}}$ this.state.push$(x)$, since the expression is typeable, contains no field updates and the only method call has this.state as receiver, where $\Gamma \vdash$ this.state : rep Stack.

A class is well-formed with respect to encapsulation, denoted as $\vdash_{\mathbf{enc}} c$, if and only if all pure methods have bodies that are pure, and all impure methods have bodies that are encapsulated, according to the corresponding definitions above. We recall that according to Definition 5.3, a method declared to be pure can only be overridden by another method declared to be pure. A program $P$ is well-formed with respect to encapsulation if all its classes are encapsulated.

**Definition 6.3 (Encapsulated well-formed classes and programs)**

$$\frac{\begin{array}{l}\forall m \;.\; \mathcal{M}(c, m) = \textsf{pure} : s_r\ (s_x) \Longrightarrow \\ \hfill (\textsf{self}\ c, s_x) \vdash_{\textit{\textbf{pure}}} \mathcal{MBody}(c, m) \\ \forall m \;.\; \mathcal{M}(c, m) = \textsf{impure} : s_r\ (s_x) \Longrightarrow \\ \hfill (\textsf{self}\ c, s_x) \vdash_{\textit{\textbf{enc}}} \mathcal{MBody}(c, m)\end{array}}{\vdash_{\textit{\textbf{enc}}} c}\;(\textsc{WFEncClass})$$

$$\vdash_{\textit{\textbf{enc}}} P \iff (\vdash P \;\wedge\; \forall c \in \mathrm{Id}_c \;.\; \vdash_{\textit{\textbf{enc}}} c)$$

**Example 6.4 (Comparing Topological and Encapsulation Systems)** *We compare the Topological Type System and the Encapsulation Type System in the context of Example 2.1 (that is, the program from from Figure 2 and a variable* node *of type* any Node*). We explained in Example 2.1 that the update* node.value := y *is valid in the Topological Type System provided that* y*'s class is a subclass of* Object*. Since the viewpoint-adapted type of field* value *is not* lost*, the conditions of rule (*Assign*) in Figure 8 are satisfied. However, the encapsulation rule (*Enc*) in Definition 6.2 forbids the update through the* any *reference* node*.*

We also define encapsulation and purity judgements for *runtime expressions* subject to a heap $h$ and a stack frame $\sigma$; these judgements are denoted as $h, \sigma \vdash_{\mathbf{enc}} e$ and $h, \sigma \vdash_{\mathbf{pure}} e$, respectively. Encapsulation and purity for runtime expressions impose similar requirements to those for source expressions but add an extra clause for frame expressions. In particular, encapsulation for frames, $h, \sigma \vdash_{\mathbf{enc}} \mathsf{frame}\ \sigma'\ e'$, requires that the receiver in $\sigma'$, that is $\sigma'(\mathsf{this})$, is a self, peer or rep of that in $\sigma$. This condition is expressed through the predicate $h \vdash \sigma' \preceq_{\mathbf{enc}} \sigma$, defined below.

**Definition 6.5 (Frame encapsulation)**

$$h \vdash \sigma' \preceq_{enc} \sigma \overset{def}{=} \exists u \in \{\mathit{peer}, \mathit{rep}\}\ .\ h, \sigma \vdash \sigma'(\mathit{this}) : u$$

**Definition 6.6 (Purity and encapsulation for runtime expressions)**

$$\frac{\begin{array}{l} h, \sigma \vdash e : t \\ \forall C, e_1, \sigma_1\ .\ e = C[\mathit{frame}\ \sigma_1\ e_1] \Longrightarrow h, \sigma_1 \vdash_{pure} e_1 \\ \forall C, e_1, e_2, f\ .\ e \neq C[e_1.f := e_2] \\ \forall C, e_1, e_2, m\ .\ e = C[e_1.m(e_2)] \Longrightarrow \exists c\ .\ h, \sigma \vdash e_1 : c\ \wedge\ \boldsymbol{pure}(c, m) \end{array}}{h, \sigma \vdash_{pure} e}\ (\text{rPure})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : t \\ \forall C, e_1, \sigma_1\ .\ e = C[\mathit{frame}\ \sigma_1\ e_1] \Longrightarrow \\ \qquad\qquad (h \vdash \sigma_1 \preceq_{enc} \sigma\ \wedge\ h, \sigma_1 \vdash_{enc} e_1)\ \vee h, \sigma_1 \vdash_{pure} e_1 \\ \forall C, e_1, e_2, f\ .\ e = C[e_1.f := e_2] \Longrightarrow \\ \qquad\qquad \exists u\ .\ h, \sigma \vdash e_1 : u\ \wedge\ u \in \{\mathit{peer}, \mathit{rep}\} \\ \forall C, e_1, e_2, m\ .\ e = C[e_1.m(e_2)] \Longrightarrow \\ \qquad\qquad \exists u, c\ .\ h, \sigma \vdash e_1 : u\ c\ \wedge\ (u \in \{\mathit{peer}, \mathit{rep}\} \vee \boldsymbol{pure}(c, m)) \end{array}}{h, \sigma \vdash_{enc} e}\ (\text{rEnc})$$

We can show that the source and runtime notions of purity and encapsulation are closely related.

**Lemma 6.7 (Source encapsulation to runtime encapsulation)**

$$\left.\begin{array}{l} \Gamma \vdash_{pure} e \\ h, \sigma \vdash \mathsf{x} : \Gamma(\mathsf{x}) \\ h, \sigma \vdash \mathit{this} : \Gamma(\mathit{this}) \end{array}\right\} \Longrightarrow h, \sigma \vdash_{pure} e$$

$$\left.\begin{array}{l} \Gamma \vdash_{enc} e \\ h, \sigma \vdash \mathsf{x} : \Gamma(\mathsf{x}) \\ h, \sigma \vdash \mathit{this} : \Gamma(\mathit{this}) \end{array}\right\} \Longrightarrow h, \sigma \vdash_{enc} e$$

*Proof.* Using Lemma 5.8. □

We now state the Encapsulation Theorem. It says that if an expression respects encapsulation (with respect to some $h$, $\sigma$), then during its execution it will only update objects that form part of the representation of the owner of the currently active object.

**Theorem 6.8 (Encapsulation)**

$$\left.\begin{array}{l} \vdash_{\boldsymbol{enc}} P \\ h, \sigma \vdash_{\boldsymbol{enc}} e \\ \sigma \vdash e, h \leadsto^{*} e', h' \\ a \in \boldsymbol{dom}(h) \\ \boldsymbol{owner}(h, \sigma(\mathit{this})) \notin \boldsymbol{owner}^{+}(h, a) \end{array}\right\} \Longrightarrow h(a) = h'(a)$$

In terms of our running example, the Encapsulation Theorem guarantees, that execution of a method by receiver 2 (i.e., $\sigma(\mathsf{this}) = 2$) will not modify the objects 1, 6, 7, 8, 9, 10, 11 and 12. It may, however, modify the fields in 2, 3, 4, 5 and 13.

On the other hand, consider the further Stack object 13, which is also owned by 1; execution of a method by 13 would be allowed to update the fields of 2, 3, 4 and 5, for instance, by calling an impure method on 2, which in turn would update the fields of 2, 3, 4 and 5. These updates are permissible, according to our theorem, because 1, the owner of 13, is among the transitive owners of 2, 3, 4 and 5.

Before we can prove Theorem 6.8, we need to introduce a number of auxiliary lemmas. In the following lemma we show that execution preserves purity and encapsulation, and that the execution of pure expressions preserves the contents of allocated objects.

**Lemma 6.9 (Preservation of purity and encapsulation )**
*For any program such that* $\vdash_{\boldsymbol{enc}} P$*, if* $\sigma \vdash e, h \leadsto e', h'$ *then:*

1. *If* $h, \sigma \vdash_{\boldsymbol{pure}} e$ *then*
   *(a)* $h', \sigma \vdash_{\boldsymbol{pure}} e'$
   *(b)* $a \in \boldsymbol{dom}(h) \Rightarrow h'(a) = h(a)$
2. *If* $h, \sigma \vdash_{\boldsymbol{enc}} e$ *then*
   *(a)* $h', \sigma \vdash_{\boldsymbol{enc}} e'$

*Proof.* See Appendix A. □

The following definition of extended runtime contexts, $D[\cdot]$, allows for contexts within any number of nested calls: An expression $e$ can be decomposed as $e = D[\mathsf{frame}\ \sigma\ e']$ if and only if it contains a nested method call with receiver and argument as described by $\sigma$ and method body $e'$[6].

**Definition 6.10 (Extended runtime contexts)**

$$D[\cdot] ::= E[\cdot] \ \mid\ E[\mathit{frame}\ \sigma\ D[\cdot]]$$

[6] $D[\cdot]$ contexts are more liberal than $E[\cdot]$ contexts, however no such relation exists between $D[\cdot]$ and $C[\cdot]$ contexts. For example, $\mathsf{x.f.m}(\cdot)$ is a $C[\cdot]$ but not a $D[\cdot]$ context, while $a.\mathsf{m}(\mathsf{frame}\ \sigma\ \cdot)$ is a $D[\cdot]$ but not a $C[\cdot]$ context.

Lemma 6.11 guarantees that the execution of an encapsulated expression $e$ can only modify an object $a$ if it is directly mentioned in one of the nested calls ($e = D[\mathsf{frame}\ \sigma' \ \ E[a.f := v]]$ or $e = E[a.f := v]$, and furthermore, $a$ must be a $\mathsf{rep}$ or $\mathsf{peer}$ of the receiver of the nested call which causes the modification. In terms of our running example, if execution of an encapsulated expression were to modify object 2, then one of the objects 1, 2 or 13 will be either the outermost receiver, or the receiver in one of the stack frames in the expression itself.

**Lemma 6.11 (Encapsulated expressions have limited write effects)** *If $\vdash_{enc} P$, and $h, \sigma \vdash_{enc} e$, and $\sigma \vdash e, h \rightsquigarrow e', h'$, and $h(a) \neq h'(a)$ for some $a \in \boldsymbol{dom}(h)$, then there exist $\sigma', f, v, D[\cdot]$, and $E[\cdot]$ such that*

1. *$e = D[\mathsf{frame}\ \sigma' \ \ E[a.f := v]]$ or ($\sigma' = \sigma$ and $e = E[a.f := v]$)*
   *and*
2. *$h, \sigma' \vdash a : \mathsf{rep}$ or $h, \sigma' \vdash a : \mathsf{peer}$*

*Proof.* The proof proceeds by induction on the derivation of $\sigma \vdash e, h \rightsquigarrow e', h'$ considering cases for the last rule applied in the derivation, and using the preservation of encapsulation (Lemma 6.9), and the definition of encapsulated expressions (Definition 6.6). In Appendix A we outline some interesting cases. □

Lemma 6.12 guarantees that for an encapsulated expression $e$, the outermost receiver ($\sigma(\mathsf{this})$), is either a peer, or a transitive owner of any of the receivers of non-pure method calls in $e$. In terms of our running example, if an expression $e$ were encapsulated from the point of view of $\sigma$ and contained a method call with receiver 2, i.e., if $h, \sigma \vdash_{\mathbf{enc}} \ldots \mathsf{frame}\ (2, \ldots)\ \ldots$, then the outermost receiver, i.e., $\sigma(\mathsf{this})$, will be either 13, 2 or 1.

**Lemma 6.12 (Owners of receivers precede them on the stack)**

$$\left.\begin{array}{l} \vdash_{enc} P \\ h, \sigma \vdash_{enc} e \\ e = D[\mathit{frame}\ \sigma' \ e'] \\ \sigma' = (a, \_) \end{array}\right\} \Longrightarrow \begin{array}{c} h, \sigma' \vdash_{pure} e' \\ \mathit{or} \\ \boldsymbol{owner}(h, \sigma(\mathit{this})) \in \boldsymbol{owner}^{+}(h, a) \end{array}$$

*Proof.* By induction on the definition of $D[\cdot]$ (c.f., Definition 6.10). We freely use the fact that any evaluation context $E[\cdot]$ is trivially an expression context $C[\cdot]$ (note that neither kind of context contain frames).

(Case: $D[\cdot] = E[\cdot]$) Then $e = E[\mathsf{frame}\ \sigma' \ e']$. By Definition 6.6, we obtain that either $h, \sigma' \vdash_{\mathbf{pure}} e'$ (in which case we are done) or $h \vdash \sigma' \preceq_{\mathbf{enc}} \sigma$. In the latter case, by Definition 6.5, we obtain that either $\sigma(\mathsf{this}) = \mathbf{owner}(h, a)$ or $\mathbf{owner}(h, \sigma(\mathsf{this})) = \mathbf{owner}(h, a)$. In either case, $\mathbf{owner}(h, \sigma(\mathsf{this})) \in \mathbf{owner}^{+}(h, a)$ as required.

(Case: $D[\cdot] = E[\mathsf{frame}\ \sigma'' \ D'[\cdot]]$) Then $e = E[\mathsf{frame}\ \sigma'' \ D'[\mathsf{frame}\ \sigma' \ e']$. By Definition 6.6, we obtain that either $h, \sigma' \vdash_{\mathbf{pure}} e'$ (in which case we are done) or both $h \vdash \sigma' \preceq_{\mathbf{enc}} \sigma$ and $h, \sigma' \vdash_{\mathbf{enc}} e'$. By induction, we obtain that either $h, \sigma' \vdash_{\mathbf{pure}} e'$ (and we are done) or $\mathbf{owner}(h, \sigma'(\mathsf{this})) \in \mathbf{owner}^{+}(h, a)$. By combining this latter statement with $h \vdash \sigma' \preceq_{\mathbf{enc}} \sigma$, we can show that $\mathbf{owner}(h, \sigma(\mathsf{this})) \in \mathbf{owner}^+(h, a)$ by a similar argument to the previous case. $\square$

Using the lemmas above, we can now prove the encapsulation theorem itself.

**Theorem 6.8 (Encapsulation)**

$$\left.\begin{array}{l} \vdash_{\boldsymbol{enc}} P \\ h, \sigma \vdash_{\boldsymbol{enc}} e \\ \sigma \vdash e, h \leadsto^* e', h' \\ a \in \boldsymbol{dom}(h) \\ \boldsymbol{owner}(h, \sigma(\mathit{this})) \notin \boldsymbol{owner}^+(h, a) \end{array}\right\} \Longrightarrow h(a) = h'(a)$$

*Proof.* We prove the equivalent assertion that $\vdash_{\mathbf{enc}} P$, and $h, \sigma \vdash_{\mathbf{enc}} e$, and $\sigma \vdash e, h \leadsto^* e', h'$, and $a \in \mathbf{dom}(h)$, and $h(a) \neq h'(a)$ imply that $\mathbf{owner}(h, \sigma(\mathsf{this})) \in \mathbf{owner}^+(h, a)$. The proof proceeds by induction over the length of the reduction of $\sigma \vdash e, h \leadsto^* e', h'$.

The base case trivially holds, since we have an execution of length zero, and thus $h = h'$.

For the inductive step, we have $\sigma \vdash e, h \leadsto^* e'', h'' \leadsto e', h'$. By application of Lemma 6.9 we obtain $h'', \sigma \vdash_{\mathbf{enc}} e''$.

**1st Case** $h(a) \neq h''(a)$. The assertion follows from the inductive hypothesis.

**2nd Case** $h(a) = h''(a)$. Because of the assumption that $h(a) \neq h'(a)$ we obtain $h''(a) \neq h'(a)$. Therefore, by the fact that $h'', \sigma \vdash_{\mathbf{enc}} e''$ and Lemma 6.11, we obtain that there exist $D[\cdot]$, $E[\cdot]$, $\sigma'$ such that

$$(\ h'', \sigma' \vdash a : \mathsf{rep} \text{ or } h'', \sigma' \vdash a : \mathsf{peer})$$
$$\text{and}$$
$$(e'' = D[\mathsf{frame}\ \sigma'\ \ E[a.f := v]] \text{ or } (\sigma' = \sigma \text{ and } e''\ = E[a.f := v])).$$

The first part of the conjunction gives $\mathbf{owner}(h'', \sigma'(\mathsf{this})) \in \mathbf{owner}^+(h'', a)$, while the latter, together with the fact that $h'', \sigma \vdash_{\mathbf{enc}} e''$ and application of Lemma 6.12 gives that $\mathbf{owner}(h'', \sigma(\mathsf{this})) \in \mathbf{owner}^+(h'', \sigma'(\mathsf{this}))$. The last two assertions give that $\mathbf{owner}(h'', \sigma(\mathsf{this})) \in \mathbf{owner}^+(h'', a)$, and because $a$ and $\sigma(\mathsf{this})$ were already defined in $h$, and owners do not change during execution, we also obtain that $\mathbf{owner}(h, \sigma(\mathsf{this})) \in \mathbf{owner}^+(h, a)$. $\square$

## 6.2 Owner-As-Modifier Discipline

The owner-as-modifier discipline [15] guarantees that any update to the field of an object is initiated by the object's owner. By "initiated", we mean that the owner is still on the stack when the modification takes place. This guarantee can only be made if we consider executions starting at the root of our heap topology, otherwise there is no guarantee that the call-stack will reflect the hierarchy of the heap topology.

We formalise the notion of an initial heap and stack as follows: $h_{\mathsf{init}}$ indicates an initial heap which only contains one object (belonging to $\mathsf{root}$) at address $1$, while $\sigma_{\mathsf{init}}$ indicates an initial stack, where $\sigma_{\mathsf{init}} = (1, \mathsf{null})$.

Theorem 6.14 states formally the owner-as-modifier guarantee. In terms of our running example, any modification of the fields of, say, the object 4 is, according to the owner-as-modifier discipline, guaranteed to happen only while 2 is on the stack or the outermost receiver (i.e., either a direct or an indirect caller).

We first prove Lemma 6.13 below, which guarantees that, if we consider reductions that begin from an initial heap and stack, then the resulting sequence of stack frames has the property that: either the corresponding expression is pure (in which case the frame may result from a call in an arbitrary position in the heap topology, via an any or a lost reference), or else all of the (transitive) owners (except root which is not an object anyway) of the receiver in the stack frame, are receivers in a preceding stack frame. Note that this is subtly different from the requirements on the sequence of stacks imposed by the judgement $h_{\mathsf{init}}, \sigma_{\mathsf{init}} \vdash_{\mathbf{enc}} e$, which says that *if* a stack frame is in the sequence, *then* it will conform to the restrictions imposed by the $h \vdash \preceq_{\mathbf{enc}}$ relation.

Applying Lemma 6.13 to our running example, execution of an encapsulated expression starting from the initial configuration and leading to an impure expression containing a method call with receiver 12, is guaranteed to have a method call with receiver 10, enclosing the earlier method call.

**Lemma 6.13 (All owners are preserved on the stack)**

$$\left.\begin{array}{l} \vdash_{enc} P \\ h_{init}, \sigma_{init} \vdash_{enc} e \\ \sigma_{init} \vdash e, h_{init} \rightsquigarrow^* e', h' \\ e' = D[\mathit{frame}\ \sigma''\ e''] \\ a \in \boldsymbol{owner}^{+}(h', \sigma''(\mathit{this})) \setminus \{\mathit{root}, 1\} \end{array}\right\} \Longrightarrow \begin{array}{c} h', \sigma'' \vdash_{pure} e'' \\ or \\ \exists D'[\cdot], D''[\cdot], \sigma \ such\ that \\ D[\cdot] = D'[\mathit{frame}\ \sigma\ D''[\cdot]], \\ and \\ \sigma(\mathit{this}) = a \end{array}$$

*Proof.* By induction on the length of the reduction $\sigma_{\mathsf{init}} \vdash e, h_{\mathsf{init}} \rightsquigarrow^* e', h'$. For the base case, i.e., when $e = e'$, we use induction over the structure of $D[\cdot]$. For the inductive step, i.e., when $\sigma_{\mathsf{init}} \vdash e, h_{\mathsf{init}} \rightsquigarrow^* e'', h'' \rightsquigarrow^* e', h'$ by case analysis over the last step in the derivation. $\square$

We now state the owner-as-modifier guarantee, and prove it using the lemma from above, the preservation of encapsulation (Lemma 6.9), and the fact that encapsulated expressions have limited write effects (Lemma 6.11).

**Theorem 6.14 (Owner-as-modifier)**

$$\left.\begin{array}{l} \vdash_{enc} P \\ h_{init}, \sigma_{init} \vdash_{enc} e \\ \sigma_{init} \vdash e, h_{init} \rightsquigarrow^* e', h' \rightsquigarrow e'', h'' \\ a \in \boldsymbol{dom}(h') \\ h'(a) \neq h''(a) \\ a' = \boldsymbol{owner}(h', a) \neq \mathit{root} \end{array}\right\} \Longrightarrow \begin{array}{c} \sigma_{init}(\mathit{this}) = a' \\ or \\ \exists D[\cdot], D'[\cdot], \sigma, e''' \ such\ that \\ e' = D[\mathit{frame}\ \sigma\ D'[e''']], \\ and \\ \sigma(\mathit{this}) = a' \end{array}$$

*Proof.* If $a' = 1$, then we are done by construction of $\sigma_{\mathsf{init}}$. Therefore, we can proceed assuming that $a' \notin \{\mathsf{root}, 1\}$.

The first three premises and Lemma 6.9 give that $h', \sigma_{\mathsf{init}} \vdash_{\mathbf{enc}} e'$. This, together with the fourth and fifth premises, and Lemma 6.11 give that there exist $f$, $v$, $D_1[\cdot]$, $E_1[\cdot]$, $\sigma'$ such that

$$( \; h', \sigma' \vdash a : \mathsf{rep} \text{ or } \; h', \sigma' \vdash a : \mathsf{peer} \; )$$
$$\text{and}$$
$$( \; e' = D_1[\mathsf{frame}\ \sigma' \;\; E_1[a.f := v]] \text{ or } \; (\sigma' = \sigma_{\mathsf{init}} \text{ and } e' = E_1[a.f := v]) \; ).$$

The second part of the conjunction above gives the following two cases:

**1st Case** $\sigma' = \sigma_{\mathsf{init}}$ and $e' = E_1[a.f := v]$. Then, because $h', \sigma_{\mathsf{init}} \vdash_{\mathbf{enc}} e'$, using the definition of encapsulated expressions, we obtain that $h', \sigma_{\mathsf{init}} \vdash a : \mathsf{rep}$ (which gives that $a' = 1$, in which case we are done), or $h', \sigma_{\mathsf{init}} \vdash a : \mathsf{peer}$, (which gives that $a' = \mathsf{root}$, and then we are done again).

**2nd Case** $e' = D_1[\mathsf{frame}\ \sigma' \;\; E_1[a.f := v]]$. The first part of our conjunction from earlier on gives that either $\sigma'(\mathsf{this}) = a'$, or $\mathbf{owner}(h', \sigma'(\mathsf{this})) = a'$.

**2.1st Case** $\sigma'(\mathsf{this}) = a'$. We choose $\sigma = \sigma'$, and $D[\cdot] = D_1[\cdot]$, and $D'[\cdot] = E_1[\cdot]$, and $e''' = a.f := v$. This concludes the case.

**2.2nd Case** $\mathbf{owner}(h', \sigma'(\mathsf{this})) = a'$. Because $a' \notin \{\mathsf{root}, 1\}$ we can apply Lemma 6.13, and obtain that there exist further contexts $D_3[\cdot]$, $D_4[\cdot]$, and frame $\sigma$, such that $D_1[\cdot] = D_2[\mathsf{frame}\ \sigma\ D_3[\cdot]]$, and $\sigma(\mathsf{this}) = a'$. We now choose $D[\cdot] = D_2[\cdot]$, and $D'[\cdot] = \mathsf{frame}\ \sigma\ D_3[\cdot]$, and $e''' = \mathsf{frame}\ \sigma'\ E_1[a.f := v]$, and conclude the case. $\square$

## 7 Related Work

Over the past ten years, there have been a large number of publications on ownership and ownership type systems. In this section, we discuss work that is most closely related to the focus of this paper, namely the separation of ownership topologies from encapsulation policies and the formalisation of ownership type systems.

Most ownership type systems combine the enforcement of an ownership topology and an encapsulation policy. Ownership Types [10] and its descendants [4, 6, 8, 9, 29] enforce an ownership topology as well as the owner-as-dominator encapsulation policy, which guarantees that every reference chain from an object in the root context to an object goes through the object's owner. Similarly, Universe Types [14, 15, 26, 28] enforce an ownership topology as well as the owner-as-modifier encapsulation policy, which guarantees that every modification of an object is initiated by the object's owner. In this paper, we showed how to separate the Topological System from the Encapsulation System. This separation is facilitated by distinguishing between the 'don't care' modifier any and the 'don't know' modifier lost because the Topological Type System treats them differently.

Ownership domains [1] was the first ownership system that separated the ownership topology from the encapsulation policy. This is achieved by allowing programmers to distinguish between private and public ownership domains and to declare links between ownership domains. While the Encapsulation Type System presented in this paper enforces a fixed encapsulation policy, it is possible to combine our Topological System with various encapsulation policies.

Dietl and Müller [16] encoded ownership types on top of Dependent Classes [19]. Dependent Classes are used to enforce the ownership topology, whereas encapsulation has to be enforced separately.

Most ownership type systems have been formalised for a small programming language similar to the one used in this paper. The formalisation of OGJ [29] is based on Java generics. Ownership information is encoded in the type parameters, which makes the formalisation simple.

Dynamic ownership [23] as available in Spec# uses ghost state to encode the ownership topology and the Boogie verification methodology to enforce an encapsulation policy similar to the one of Universe Types. The Topological Type System presented in this paper can be combined with the Boogie methodology.

Type checkers for the Universe Type System are implemented in the JML tools [22, 12] and as a pluggable type system for Scala [13].

In this work, in keeping with most works on Universe or Ownership Types, each object is owned directly by at most one other object, and the ownership hierarchy forms a tree. This view can, however, be generalised to allow several direct owners, and the ownership hierarchy to form a DAG [7].

## 8 Conclusion

We presented UT, a new formalisation of the Universe Type System, which is given in two steps: first presenting a Topological Type System that builds the ownership topology and then augmenting it to the Encapsulation Type System. The two-step formalisation permits a gentler presentation of the mathematical machinery we develop and primarily allows for separation of concerns when extending this work, as some extensions and applications of Universe Types do not require encapsulation properties. Both of these factors facilitate the adoption of the work as a starting point for further work.

We introduced the distinction between the 'don't care' modifier any and the 'don't know' modifier lost. We proved subject reduction (for both the Topological and the Encapsulation Type System) for a small-step operational semantics of a subset of Java. Like UT most ownership type systems have been formalised on paper. We also formalised a version of Universe Types including arrays in Isabelle and proved type safety [21]. The main difference is that there we use a big-step semantics, whereas here we use a small-step semantics.

This formalisation of the Universe Type System is the basis for various future extensions. We plan to extend our work on Generic Universe Types [14] to also separate topology from encapsulation. We are also planning to improve the expressiveness of Universe Types by adding path-dependent types. Adapting the type system to Java bytecode is other future work. This will permit the use of Universe Types for the verification of mobile bytecode in a Proof-Carrying-Code architecture such as the one proposed by the Mobius project [20].

## Acknowledgements

We thank our reviewer for extensive feedback and many useful suggestions.

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and the EPSRC grant Practical Ownership Types for Objects and Aspect Programs, EP/D061644/1.

## References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.
2. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped Types and Aspects for Real-Time Java. In D. Thomas, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 124–147. Springer-Verlag, 2006.
3. A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, pages 166–177. ACM, 2002.
4. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming.* PhD thesis, MIT, 2004.
5. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2002.
6. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of programming languages (POPL)*, pages 213–223. ACM, 2003.
7. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 441–460. ACM, 2007.
8. D. Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, 2001.
9. D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 292–310. ACM, 2002.
10. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 33:10, pages 48–64. ACM, 1998.
11. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, 2007.
12. W. Dietl. JML2 Eclipse plug-in. Available from `pm.inf.ethz.ch/research/universes/tools/eclipse/`.
13. W. Dietl. Universe type system tools for Scala. Available from `pm.inf.ethz.ch/research/universes/tools/scala/`.
14. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 28–53. Springer-Verlag, 2007.

15. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.
16. W. Dietl and P. Müller. Ownership type systems and dependent classes. In *Foundations of Object-Oriented Languages (FOOL)*, 2008.
17. M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Journal of Theoretical Computer Science*, 52:205–237, 1987.
18. C. Flanagan and S. Qadeer. Types for atomicity. In *Types in Language Design and Implementation (TLDI)*, pages 1–12. ACM, 2003.
19. V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–152. ACM, 2007.
20. Global Computing Proactive Initiative. Mobius: Mobility, Ubiquity and Security. `http://mobius.inria.fr/`. IST-15905.
21. M. Klebermaß. An Isabelle formalization of the Universe Type System. Master's thesis, Technical University Munich and ETH Zurich, 2007. Available from `pm.inf.ethz.ch/projects/student_docs/Martin_Klebermass/`.
22. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML reference manual. Department of Computer Science, Iowa State University. Available from `www.jmlspecs.org`, 2008.
23. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
24. Y. Lu and J. Potter. Protecting Representation with Effect Encapsulation. In *Principles of programming languages (POPL)*, pages 359–371. ACM, 2006.
25. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.
26. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
27. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
28. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM, 2007.
29. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 311–324. ACM, October 2006.
30. A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.

## A Supporting Results and Proofs

**Lemma A.1 (Owners are defined in well-formed heap)** *If* $\vdash h$ *and* $a \in \boldsymbol{dom}(h)$ *then* $(\boldsymbol{owner}^{+}(h,a) \setminus \{\textsf{root}\}) \subseteq \boldsymbol{dom}(h)$.

*Proof.* By induction on the definition of $\mathbf{owner}^{+}(h,a)$ using the rule (WFADDR). □

**Lemma A.2 (Object owner and class preservation)**

$$(i)\quad \left.\begin{array}{l} h(a) = (o, c, \_) \\ (h', a') = \mathbf{alloc}(h, \sigma, t) \end{array}\right\} \implies \left\{\begin{array}{l} h'(a) = (o, c, \_) \\ \mathbf{owner}^{+}(h', a) = \mathbf{owner}^{+}(h, a) \end{array}\right.$$

$$(ii)\quad \left.\begin{array}{l} h(a) = (o, c, \_) \\ h' = h[(a', f) \mapsto v] \end{array}\right\} \implies \left\{\begin{array}{l} h'(a) = (o, c, \_) \\ \mathbf{owner}^{+}(h', a) = \mathbf{owner}^{+}(h, a) \end{array}\right.$$

$$(iii)\quad \left.\begin{array}{l} h(a) = (o, c, \_) \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array}\right\} \implies \left\{\begin{array}{l} h'(a) = (o, c, \_) \\ \mathbf{owner}^{+}(h', a) = \mathbf{owner}^{+}(h, a) \end{array}\right.$$

*Proof.*

(i) Immediate from the definition of $\mathbf{alloc}(h,\sigma,t)$, noting it is not possible that $a' = a$ since $a \in \mathbf{dom}(h)$ and $a' \notin \mathbf{dom}(h)$.
(ii) Follows from the definition of $h[(a', f) \mapsto v]$, in which the owner and class information is explicitly preserved.
(iii) By induction on the derivation of $\sigma \vdash e, h \rightsquigarrow e', h'$, using parts (i) and (ii). □

**Lemma A.3 (Heap domain inclusion)**

*(i) If* $(h', a) = \boldsymbol{alloc}(h, \sigma, t)$ *then* $a \notin \boldsymbol{dom}(h)$ *and* $\boldsymbol{dom}(h') = \boldsymbol{dom}(h) \cup \{a\}$
*(ii) If* $a \in \boldsymbol{dom}(h)$ *and* $h' = h[(a, f) \mapsto v]$ *then* $\boldsymbol{dom}(h') = \boldsymbol{dom}(h)$
*(iii) If* $\sigma \vdash e, h \rightsquigarrow e', h'$ *then* $\boldsymbol{dom}(h) \subseteq \boldsymbol{dom}(h')$

*Proof.*

(i) Immediate from the definition of $\mathbf{alloc}(h,\sigma,t)$.
(ii) Immediate from the definition of $h[(a, f) \mapsto v]$.
(iii) By induction on the derivation of $\sigma \vdash e, h \rightsquigarrow e', h'$, using parts (i) and (ii). □

**Lemma A.4 (Soundness of object creation)**

$$\left.\begin{array}{l} u \in \{\textsf{rep}, \textsf{peer}\} \\ h \vdash \sigma \\ (h', a) = \boldsymbol{alloc}(h, \sigma, u\,c) \end{array}\right\} \implies h', \sigma \vdash a : u\,c$$

*Proof.* By case analysis of $u$, the definition of $\mathbf{alloc}(h, \sigma, u\,c)$ and using rule (TADDR). □

**Lemma A.5 (Heap operations preserve value types)** *If* $h, \sigma \vdash v : t$ *then*

*(i) If* $(h', a) = \textbf{\textit{alloc}}(h, \sigma, t')$ *then* $h', \sigma \vdash v : t$
*(ii) If* $h' = h[(a, f) \mapsto v']$ *then* $h', \sigma \vdash v : t$

*Proof.* There are two sub-cases:

$v =$ **null:** we can still derive the type judgement using (tNull).
$v = b$**:** Neither of the operations change the ownership and class information of an existing object in a heap, as we saw in Lemma A.2. Thus we can still derive $h', \sigma \vdash b : t$ in both cases, using (tAddr). $\square$

**Lemma A.6 (Heap operations and well-formedness)** *If* $\vdash h$ *and* $h \vdash \sigma$ *then*

*(i) If* $u \in \{\mathsf{rep}, \mathsf{peer}\}$ *and* $(h', a) = \textbf{\textit{alloc}}(h, \sigma, u\, c)$ *then* $\vdash h'$
*(ii) If* $h, \sigma \vdash a : u\, c$*,* $\mathcal{F}(c, f) = s$ *and* $h, (a, \_) \vdash v : s$ *then* $\vdash h[(a, f) \mapsto v]$

*Proof.*

(i) First, we notice that by Lemma A.3(i) and Lemma A.2(i) we know:

$$a \notin \mathbf{dom}(h) \;\wedge\; \mathbf{dom}(h') = \mathbf{dom}(h) \cup \{a\} \tag{9}$$
$$\forall b \in \mathbf{dom}(h).\; \mathbf{class}(h', b) = \mathbf{class}(h, b) \tag{10}$$
$$\forall b \in \mathbf{dom}(h).\; \mathbf{owner}(h', b) = \mathbf{owner}(h, b) \tag{11}$$
$$\forall b \in \mathbf{dom}(h).\; \mathbf{owner}^{+}(h', b) = \mathbf{owner}^{+}(h, b) \tag{12}$$

We aim to show $\vdash h'$ by rule (WFHeap, Definition 5.7). By the assumption $\vdash h$, and the form of the rule (which is the only rule which can derive such judgements), we must have:

$$b \in \mathbf{dom}(h) \Rightarrow \mathsf{root} \in \mathbf{owner}^{+}(h, b) \tag{13}$$
$$\forall b \in \mathbf{dom}(h).\; h \vdash b \tag{14}$$

We show the second premise of (WFHeap) first; i.e., that $\forall b \in \mathbf{dom}(h').\; h' \vdash b$. To show this, we consider two cases for $b$:

**$(b \neq a)$:** Then by (9), we know $b \in \mathbf{dom}(h)$. By (14) we have $h \vdash b$. From the form of (WFAddr), and using (10) and (11), it suffices to prove (where $\mathbf{class}(h', b) = \mathbf{class}(h, b) = c$) that $\mathcal{F}(c, f){=}s \Rightarrow h', (a, \_) \vdash h'(a.f) : s$. Since $h \vdash b$, we know that $\mathcal{F}(c, f){=}s \Rightarrow h, (b, \_) \vdash h(b.f) : s$. We complete the case by applying Lemma A.5(ii).

**$(b = a)$:** We show that the premises of (WFAddr) hold, directly to deduce $h' \vdash b$. Since $h \vdash \sigma$, we have in particular that $\sigma(\mathsf{this}) \in \mathbf{dom}(h)$. By Lemma A.1, we also know that either $\mathbf{owner}(h, \sigma(\mathsf{this})) \in \mathbf{dom}(h)$ or $\mathbf{owner}(h, \sigma(\mathsf{this})) = \mathsf{root}$. From the definition of $\mathbf{alloc}(h, \sigma, u\; c)$ we can then show $\mathbf{owner}(h', a) \in (\mathbf{dom}(h) \cup \{\mathsf{root}\}) \subseteq (\mathbf{dom}(h') \cup \{\mathsf{root}\})$. Furthermore, by Lemma A.4, we know that $h', \sigma \vdash a : u\, c$ as required.

To show the first premise of (WFHeap), suppose that $b \in \mathbf{dom}(h')$. By (9), we know that either $b \in \mathbf{dom}(h)$ or $b = a$. In the former case, we have (by (13) and (12)), $\mathsf{root} \in \mathbf{owner}^+(h, b) = \mathbf{owner}^+(h', b)$ as required. In the latter case, from the argument above we know that $\mathbf{owner}(h', a) \in (\mathbf{dom}(h) \cup \{\mathsf{root}\})$ Therefore by (13) and the definition of $\mathbf{owner}^+(a, h')$, we know that $\mathsf{root} \in \mathbf{owner}^+(a, h')$ as required. Thus we have all the premises needed to apply the rule (WFHeap) and obtain $\vdash h'$.

(ii) Let $h' = h[(a, f) \mapsto v]$ in what follows. We aim to deduce $\vdash h'$ by rule (WFHeap). By Lemma A.3(ii) we know that $\mathbf{dom}(h') = \mathbf{dom}(h)$. By Lemma A.2(ii), we therefore also know that the ownership and class information defined in $h$ is exactly that defined in $h'$. Therefore, given the assumption $\vdash h$, and considering the premises of the rules (WFHeap) and (WFAddr), it suffices to prove that:

$$\left.\begin{array}{l} a' \in \mathbf{dom}(h') \\ \mathbf{class}(h', a') = c' \\ \mathcal{F}(c', f') = s' \end{array}\right\} \implies h', (a', \_) \vdash h'(a'.f') : s'$$

We now consider two cases.

Firstly, if either $a \neq a'$ or $f \neq f'$, then by definition of $h[(a, f) \mapsto v]$, we have $h'(a'.f') = h(a'.f')$. Therefore $h', (a', \_) \vdash h'(a'.f') : s'$ follows from the assumption $\vdash h$ (examining the premises of (WFHeap) and (WFAddr)).

On the other hand, if both $a = a'$ and $f = f'$ then $c' = \mathbf{class}(h', a') = \mathbf{class}(h, a) = c$ and so $s' = \mathcal{F}(c', f') = \mathcal{F}(c, f) = s$. Therefore, it suffices to prove $h', (a, \_) \vdash h'(a.f) : s$. This follows from applying Lemma A.5(ii) to the assumption $h, (a, \_) \vdash v : s$, noting that by definition of $h[(a, f) \mapsto v]$ we have $h'(a.f) = v$. □

**Lemma A.7 (Extracting information from address type judgements)** *If $h, \sigma \vdash a : u\ c$ then*

*(i)* $a \in \boldsymbol{dom}(h)$
*(ii)* $\boldsymbol{class}(h, a) \ \leq_c \ c$
*(iii)* $(\sigma(\mathit{this}), \boldsymbol{owner}(h, \sigma(\mathit{this}))) \vdash (a, \boldsymbol{owner}(h, a)) : u$

*Proof.* By induction on the derivation of $h, \sigma \vdash a : u\ c$, specifically using the rules (tADDR), (tSUB) and Lemma 3.2. □

**Lemma A.8 (Extracting information from method type judgements)** *If $h, \sigma \vdash e_1.m(e_2) : t$ then there exist $u_1$ and $c_1$ such that:*

*(i)* $h, \sigma \vdash e_1 : u_1\ c_1$
*(ii)* $\mathcal{M}(c_1, m) = p : s_r(s_x)$
*(iii)* $u_1 \rhd s_x \neq \mathit{lost}$
*(iv)* $h, \sigma \vdash e_2 : u_1 \rhd s_x$

Note that we make no requirements on how $u_1$ and $c_1$ are related to $t$; our assumption simply insists that the call is typeable *somehow*.

*Proof.* By induction on the derivation of $h, \sigma \vdash e_1.m(e_2) : t$, specifically using the rules (tCALL) and (tSUB). □

**Lemma A.9 (Viewpoint adaptation preserves subtyping)**

$$s \leq s' \implies u \rhd s \leq u \rhd s'$$

*Proof.* By case analysis of $u$ and the relation $u_1 \leq_u u_2$. □

**Lemma 5.4 (Determining the relative Universe Types of values)**

*(i) If $h, \sigma \vdash a : u$ _ and $h, (a, \_) \vdash v : t$ then $h, \sigma \vdash v : u \rhd t$*
*(ii) If $h, \sigma \vdash a : u$ _ and $h, \sigma \vdash v : u \rhd t$ and $u \rhd t \neq$ lost then, for any value $v'$ we have $h, (a, v') \vdash v : t$*

*Proof.* Uses Lemma A.7(iii) to extract Universe determination judgement for addresses. We have two cases for $v$:

– If $v =$ null then we can trivially derive any type judgement using (tNull).
– If $v = b$ we use Lemma A.7(iii) to extract Universe judgements for the addresses $a$ and $b$. Then we apply Lemma 3.4 and the rule (tADDR) to obtain the desired judgement. □

**Theorem 5.9 (Topological Subject Reduction)** *If a program is well-formed, then*

$$\left.\begin{array}{l} \vdash h \\ h \vdash \sigma \\ h, \sigma \vdash e : t \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array}\right\} \implies \left\{\begin{array}{l} \vdash h' \\ h' \vdash \sigma \\ h', \sigma \vdash e' : t \end{array}\right.$$

*Proof.* By induction over the structure of

$$h, \sigma \vdash e : t \tag{15}$$

The most interesting cases are when the last rules used to derive (15) are (tField),(tAssign) and (tCall), which we show here. We leave the other cases for the interested reader. In all cases, the conclusion $h' \vdash \sigma$ follows straightforwardly, using Lemmas A.3 and A.7 as necessary.

**(tField):** From the premise of the rule we know

$$e = e_1.f \tag{16}$$
$$t = u \rhd s' \tag{17}$$
$$h, \sigma \vdash e_1 : u\ c \tag{18}$$
$$\mathcal{F}(c, f) = s' \tag{19}$$

From (16) we know the reduction was derived using either (rField) or (rEvalCtxt).

**(rField):** we know:

$$e_1 = a \tag{20}$$
$$h' = h \tag{21}$$
$$e' = h(a.f) = v \tag{22}$$

From (18) and (20), and applying Lemma A.7(i), we know that

$$a \in \mathbf{dom}(h) \tag{23}$$

From $\vdash h$, the premises of the rule (WFHeap) and (23), we know in particular that

$$h \vdash a \tag{24}$$

From (24) and the premise of (WFAddr), along with (19) and (22) we deduce

$$h, (a, \_) \vdash v : s' \tag{25}$$

By (21), (20), (18), (25) and Lemma 5.4 we obtain

$$h, \sigma \vdash v : u \rhd s'$$

The resultant heap $h'$ is trivially shown to be well-formed from the assumption $\vdash h$ and (21).

**(rEvalCtxt):** we know:

$$e' = e_1'.f \tag{26}$$
$$\sigma \vdash e_1, h \rightsquigarrow e_1', h' \tag{27}$$

By $\vdash h$, (18), (27) and the inductive hypothesis we know

$$h, \sigma \vdash e_1' : u\ c \tag{28}$$
$$\vdash h' \tag{29}$$

By (28), (19), (17), (26) and (tField) we derive our first required conclusion

$$h, \sigma \vdash e' : t$$

and (29) gives us the second required conclusion.

**(tAssign):** From the premises of the rule we know

$$e\ =\ e_1.f := e_2 \tag{30}$$
$$h, \sigma \vdash e_1 : u\ c \tag{31}$$
$$\mathcal{F}(c, f) = s \tag{32}$$
$$u \rhd s \neq \mathsf{lost} \tag{33}$$
$$h, \sigma \vdash e_2 : u \rhd s \tag{34}$$

From the structure of $e$, (30), we know the reduction could have been derived using either (rAssign) or (rEvalCtxt). We here consider the former case, (rAssign), and leave the (easier) latter case for the interested reader. From (rAssign) we know:

$$e_1 = a \tag{35}$$
$$e_2 = e' = v \tag{36}$$
$$h' = h[(a, f) \mapsto v] \tag{37}$$

Applying Lemma 5.4(ii), using (35), (31), (36), (34), (33), we obtain

$$h, (a, \_) \vdash v : s \tag{38}$$

By the assumption $\vdash h$, (35), (31), (32), (38) and Lemma A.6 we get

$$\vdash h[(a, f) \mapsto v]$$

Also, by (36) and (34) we obtain

$$h, \sigma \vdash e' : u \rhd s \tag{39}$$

and by (39) and Lemma A.5 we get

$$h[(a, f) \mapsto v], \sigma \vdash e' : u \rhd s$$

as required.

**(tCall):** From the premises of this rule we know

$$e = e_1.m(e_2) \tag{40}$$
$$h, \sigma \vdash e_1 : u_1\ c_1 \tag{41}$$
$$\mathcal{M}(c_1, m) = p : s_1^r\ (s_1^x) \tag{42}$$
$$u_1 \rhd s_1^x \neq \mathsf{lost} \tag{43}$$
$$h, \sigma \vdash e_2 : u_1 \rhd s_1^x \tag{44}$$
$$t = u_1 \rhd s_1^r \tag{45}$$

From the structure of $e$ derived from (40), we know the reduction could have been derived using either (rCall) or (rEvalCtx). We here consider the case for (rCall) and leave the other case for the interested reader. From (rCall) and its assumption we know

$$e_1 = a \text{ and } e_2 = v \tag{46}$$
$$h' = h \tag{47}$$
$$\sigma' = (a, v) \tag{48}$$
$$c_a = \mathbf{class}(h, a) \tag{49}$$
$$e_b = \mathcal{MBody}(c_a, m) \tag{50}$$
$$e' = \mathsf{frame}\ \sigma'\ e_b \tag{51}$$

From (47) and the assumption $\vdash h$ we know that the resulting heap is well-formed. Thus from (51), (45) we only need to show that

$$h, \sigma \vdash \textsf{frame}\ \sigma\ 'e_b : u_1 \rhd s_1^r \tag{52}$$

The rest of the proof is dedicated to showing this.
From (41), (46), (49) and Lemma A.7 we derive

$$c_a \leq c_1 \tag{53}$$

From (53), (42), the premises of the rule (WFClass) and the assumption of well-formed programs, giving $\vdash c_a$, we derive

$$\mathcal{M}(c_a, m) = p : s^r\ (s^x) \tag{54}$$

$$s^r \leq s_1^r \tag{55}$$

$$s_1^x \leq s^x \tag{56}$$

Also, by (50), (54), $\vdash c_a$ and the premises of (WFClass) we derive

$$\Gamma \vdash e_b : s^r \tag{57}$$

$$\text{where}\quad \Gamma = (\textsf{self}\ c_a,\ s^x) \tag{58}$$

Since $\sigma'(\textsf{this}) = a$ (using (48)), we can use the rule (Self) (of Figure 4) to derive

$$(\sigma'(\textsf{this}), \mathbf{owner}(h, \sigma'(\textsf{this}))) \vdash (a, \mathbf{owner}(h, a)) : \textsf{self} \tag{59}$$

Using (49), (59) and (tAddr) we derive

$$h, \sigma' \vdash a : \textsf{self}\ c_a \tag{60}$$

Also, using Lemma 5.4(ii) with (46), (48), (41), (44), we get

$$h, \sigma' \vdash v : s_1^x \tag{61}$$

and by (61), (56) and (tSub) we derive

$$h, \sigma' \vdash v : s^x \tag{62}$$

By (48), (57), (58), (60), (62) and Lemma 5.8 we derive

$$h, \sigma' \vdash e_b : s^r \tag{63}$$

and by (63), (41), (46),(48) and (tFrame) we get

$$h, \sigma \vdash \textsf{frame}\ \sigma'\ e_b : u_1 \rhd s^r \tag{64}$$

From (55) and Lemma A.9 we derive

$$u_1 \rhd s^r \leq u_1 \rhd s_1^r \tag{65}$$

and thus by (64), (65) and (tSub) we get

$$h, \sigma \vdash \textsf{frame}\ \sigma'\ e_b : u_1 \rhd s_1^r$$

as required by (52).

□

**Lemma 6.9 Preservation of purity and encapsulation** *For any program such that* $\vdash_{\mathbf{enc}} P$*, if* $\sigma \vdash e, h \rightsquigarrow e', h'$ *then:*

1. *If* $h, \sigma \vdash_{\mathbf{pure}} e$ *then*
   (a) $h', \sigma \vdash_{\mathbf{pure}} e'$
   (b) $a \in \mathbf{dom}(h) \Rightarrow h'(a) = h(a)$
2. *If* $h, \sigma \vdash_{\mathbf{enc}} e$ *then*
   (a) $h', \sigma \vdash_{\mathbf{enc}} e'$

*Proof.* The proof proceeds by induction on the derivation of

$$\sigma \vdash e, h \rightsquigarrow e', h' \tag{66}$$

considering cases for the last rule applied in the derivation. We show here the interesting cases and leave the simpler ones for the interested reader.

**(rAssign):** Then we know

$$e = (b.f := v) \tag{67}$$
$$e' = v \tag{68}$$
$$h' = h[(b, f) \mapsto v] \tag{69}$$

From (67) we know $h, \sigma \not\vdash_{\mathbf{pure}} e$. So we do not need to consider case (1). To show case (2), we assume

$$h, \sigma \vdash_{\mathbf{enc}} b.f := v \tag{70}$$

By using (70) and unravelling Definition 6.6, we obtain that (for some type $t$):

$$h, \sigma \vdash e : t \tag{71}$$

By applying the Topological Subject Reduction Theorem 5.9 and using (68), we therefore know

$$h, \sigma \vdash v : t \tag{72}$$

Combining this with (69), we obtain $h', \sigma \vdash v : t$, and then apply Definition 6.6 to obtain $h', \sigma \vdash_{\mathbf{enc}} v$ as required.

**(rCall):** Let $c' = \mathbf{class}(h, b)$. Then we know

$$e = b.m(v) \tag{73}$$
$$e' = \mathsf{frame}\ (b, v)\ e_b \tag{74}$$
$$e_b = \mathcal{MBody}(c', m) \tag{75}$$
$$h' = h \tag{76}$$

We consider the two cases we need to show in turn:

**1. ($h, \sigma \vdash_{\mathbf{pure}} e$):** From Definition 6.6 we know that:

$$h, \sigma \vdash b.m(v) : t \tag{77}$$

$$h, \sigma \vdash b : c \tag{78}$$

$$\mathbf{pure}(c,\ m) \tag{79}$$

Using the rule (SELF) of Figure 4, and the rule (TADDR) we can derive

$$h, (b, v) \vdash b : \mathsf{self}\ c' \tag{80}$$

Applying rule (TTHIS), we can then deduce

$$h, (b, v) \vdash \mathsf{this} : \mathsf{self}\ c' \tag{81}$$

From the assumption $\vdash_{\mathbf{enc}} P$ we know $\vdash_{\mathbf{enc}} c$ and thus by (75) and Definition 6.3 we can write:

$$\mathcal{M}(c, m) = \mathsf{pure} : s_r\ (s_x) \tag{82}$$

By (78) and Lemma A.7, we deduce

$$c' \leq_c c \tag{83}$$

By Definition 5.3 and (82) and (83), we obtain:

$$\mathcal{M}(c', m) = \mathsf{pure} : s'_r\ (s'_x) \tag{84}$$

$$s_x \leq s'_x \tag{85}$$

$$s'_r \leq s_r \tag{86}$$

From the assumption $\vdash_{\mathbf{enc}} P$ we know $\vdash_{\mathbf{enc}} c'$ and thus by (75) and Definition 6.3 we know that:

$$(\mathsf{self}\ c',\ s'_x) \vdash_{\mathbf{pure}} e_b \tag{87}$$

Returning to (77), and applying Lemma A.8, we obtain (for some $u''$, $c''$):

$$h, \sigma \vdash b : u''\ c'' \tag{88}$$

$$\mathcal{M}(c'', m) = p : s''_r(s''_x) \tag{89}$$

$$u'' \rhd s''_x \neq \mathsf{lost} \tag{90}$$

$$h, \sigma \vdash v : u'' \rhd s''_x \tag{91}$$

We can now take (88), (90) and (91) and apply Lemma 5.4(ii) to obtain:

$$h, (b, v) \vdash v : s''_x \tag{92}$$

By (88) and Lemma A.7, we deduce

$$c' \leq_c c'' \tag{93}$$

Combining this with Definition 5.3 and (89), we obtain in particular:

$$s''_x \leq s'_x \tag{94}$$

By (92), (94), Lemma A.9 and (tSub), we obtain

$$h, (b, v) \vdash v : s'_x \tag{95}$$

From this, we apply the rule (tVar) to obtain

$$h, (b, v) \vdash \mathsf{x} : s'_x \tag{96}$$

and as a result of Lemma 6.7, (87), (81) and (96) we get:

$$h, (b, v) \vdash_{\mathbf{pure}} e_b \tag{97}$$

By (77), (66), (74) and the Topological Subject Reduction Theorem 5.9 we get

$$h, \sigma \vdash \mathsf{frame}\ (b, v)\ e_b : t \tag{98}$$

and hence by (97), (98), (74), (76), and Definition 6.6 we obtain

$$h', \sigma \vdash_{\mathbf{pure}} e'$$

which completes the case.

**2.** $(h, \sigma \vdash_{\mathbf{enc}} e)$**:** From Definition 6.6 we know we have two sub-cases. The first sub-case states that the method called is pure and the proof then progresses as the previous case for $h, \sigma \vdash_{\mathbf{pure}} e$. Therefore, it suffices to consider the case when, for some $u \in \{\mathsf{peer}, \mathsf{rep}\}$ and source types $s_r$, $s_x$, we have:

$$h, \sigma \vdash b : u\ c \tag{99}$$

$$h, \sigma \vdash b.m(v) : t \tag{100}$$

$$\mathcal{M}(c, m) = \mathsf{impure} : s_r\ (s_x) \tag{101}$$

Since the method $m$ is declared impure in $c$, and we have assumed $\vdash_{\mathbf{enc}} P$ (and thus $\vdash_{\mathbf{enc}} c$), it follows from (75) and Definition 6.3 that we know

$$(\mathsf{self}\ c, s_x) \vdash_{\mathbf{enc}} e_b \tag{102}$$

By similar argument to the previous case, we can deduce from (99) and (100) that

$$h, (b, v) \vdash \mathsf{this} : \mathsf{self}\ c \tag{103}$$

$$h, (b, v) \vdash \mathsf{x} : s_x \tag{104}$$

and thus by (103), (104), (102) and Lemma 6.7 we get

$$h, (b, v) \vdash_{\mathbf{enc}} e_b \tag{105}$$

From (99) we derive (see Definition 6.5):

$$h \vdash (b, v) \preceq_{\mathbf{enc}} \sigma \tag{106}$$

Also, by (100), (66), (74) and the Topological Subject Reduction we know

$$h, \sigma \vdash \mathsf{frame}\ (b, v)\ e_b : t \tag{107}$$

Thus by (107), (106), (105), (74), (76). and Definition 6.6 we conclude

$$h', \sigma \vdash_{\mathbf{enc}} e'$$

as required.

**(rFrame2): 1. ($h, \sigma \vdash_{\mathbf{pure}} e$):** Similar to the following case.

**2. ($h, \sigma \vdash_{\mathbf{enc}} e$):** From the rule we know

$$e = \mathsf{frame}\ \sigma'\ v \tag{108}$$

$$e' = v \tag{109}$$

$$h' = h \tag{110}$$

From Definition 6.6 we know that either $h, \sigma' \vdash_{\mathbf{pure}} v$ or

$$h, \sigma \vdash \mathsf{frame}\ \sigma'\ v : t \tag{111}$$

$$h \vdash \sigma' \preceq_{\mathbf{enc}} \sigma \tag{112}$$

$$h, \sigma \vdash_{\mathbf{enc}} v \tag{113}$$

By (111), (66), the Topological Subject Reduction Theorem 5.9 and (109) we get

$$h, \sigma \vdash v : t \tag{114}$$

and by (114), (76), and Definition 6.6 we obtain

$$h', \sigma \vdash_{\mathbf{enc}} v$$

as required. □

**Lemma 6.11 (Encapsulated expressions have limited write effects)**
*If $\vdash_{enc} P$, and $h, \sigma \vdash_{enc} e$, and $\sigma \vdash e, h \rightsquigarrow e', h'$, and $h(a) \neq h'(a)$ for some $a \in \boldsymbol{dom}(h)$, then there exist $\sigma', f, v, D[\cdot]$, and $E[\cdot]$ such that*

1. *$e = D[\mathsf{frame}\ \sigma'\ \ E[a.f := v]]$ or $(\sigma' = \sigma$ and $e\ = E[a.f := v])$. and*
2. *$h, \sigma' \vdash a : \mathsf{rep}$ or $h, \sigma' \vdash a : \mathsf{peer}$*

*Proof.* The proof proceeds by induction on the derivation of $\sigma \vdash e, h \rightsquigarrow e', h'$ considering cases for the last rule applied in the derivation We show here the interesting cases:

**(rAssign):** Then, because $h(a) \neq h'(a)$, we know that there exists a field $f$, and value $v$ such that $e = (a.f := v)$, and $h' = h[(a, f) \mapsto v]$. From the latter, the encapsulation property, and the conditions of Definition 6.6 we know there exists $u \in \{\mathsf{self}, \mathsf{rep}, \mathsf{peer}\}$ such that: $h, \sigma \vdash a : u$. We choose $\sigma' = \sigma$ and $E[\cdot] = [\cdot]$, and the rest follows easily.

**(rCall):** Then $h' = h$, and thus the case is vacuous.

**(rFrame2):** Then $h' = h$, and thus the case is vacuous.

**(rFrame1):** Then we know that there exist $\sigma_1$, $e_1$ and $e_1'$, such that $e = \mathsf{frame}\ \sigma_1\ e_1$, and $\sigma_1 \vdash e_1, h \rightsquigarrow e_1', h'$, and $e' = \mathsf{frame}\ \sigma_1\ e_1'$. By application of the induction hypothesis, we obtain that there exists a $\sigma', f, v, D_1[\cdot]$, and $E_1[\cdot]$ such that

1. $h, \sigma' \vdash a : \mathsf{rep}$ or $h, \sigma' \vdash a : \mathsf{peer}$
   and
2. $e_1 = D_1[\mathsf{frame}\ \sigma'\ \ E_1[a.f := v]]$, or $(\sigma' = \sigma_1$ and $e_1 = E_1[a.f := v])$.

The second part of the conjunction above gives two cases:

**1st Case** $e_1 = D_1[\mathsf{frame}\ \sigma'\ \ E_1[a.f := v]]$. We then choose $E[\cdot] = E_1[\cdot]$, and $D[\cdot] = \mathsf{frame}\ \sigma_1\ D_1[\mathsf{frame}\ \sigma'\ \ E_1[\cdot]]$, and the rest follows.

**2nd Case** $\sigma' = \sigma_1$ and $e_1 = E_1[a.f := v]$. We then choose $E[\cdot] = E_1[\cdot]$, and $D[\cdot] = \mathsf{frame}\ \sigma_1\ E[\cdot]$, and the rest follows. □