UJ: Type Soundness for Universe Types

Dave Cunningham Adrian Francalanza
Sophia Drossopoulou
{dc04,adrianf,scd}@doc.ic.ac.uk

Werner Dietl Peter Müller
{werner.dietl,peter.mueller}@inf.ethz.ch

December 2006

Abstract

Universe types characterise aliasing in object oriented programming languages and are used to reason modularly about programs. In this report we formalise prior work by Müller and Poetzsch-Heffter, who designed the Universe Type System for a core subset of Java. We present our work in two steps. We first give a Topological Universe Type System and show subject reduction to a small-step dynamic semantics for our language. Motivated by concerns of Modular verification, we then give an Encapsulation Universe Type System (based on the owner-as-modifier principle), prove subject reduction with respect to the former small-step semantics, and show how the type system can be used for modular verification.

1 Introduction

Aliasing is an integral part of imperative object oriented programming, as e.g. in Smalltalk, Java and C++. It allows a natural presentation of real situations such as the sharing of state. However, it makes several other programming aspects more difficult, such as reasoning about programs, garbage collection and memory management, code migration, parallelism, and the analysis of atomicity.

To address these issues, type systems such as universes [15, 9], ownership types[8, 7, 6] and similar others [11, 1] have been introduced. They come in various flavours, but all have in common that they organize the heap *topology* as a tree (or forest) where each object *is owned* by at most a *single* object and the ownership relationship is acyclic. It is common practice to depict ownership through a box in the object graph. For example, in Figure 1 the dotted box around object 1 of class Bag indicates that it owns object 2, while the 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 dotted box labeled root, which gives us a tree (as opposed to a forest).

This hierarchic heap topology can be exploited in several ways: Vitek et al [1] speed up garbage collection, because as soon as an owner is unreachable, all owned objects are unreachable too. Flanagan et al [11], Boyapati et al [5]can guarantee that a program will not have races, because locking an object implicitly locks all owned objects. Clarke et al [7] use ownership types to calculate the effects of computation, and thus determine when they do not affect each other. Clarke et al [8] introduce deep ownership, whereby any

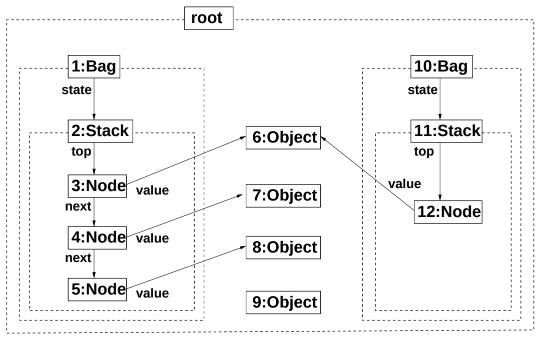

Figure 1: Depicting Object ownership and References in a Heap

path to an object from the root must go through its owner; thus the owned objects are its *representation*, and are encapsulated within the object. Leavens et al [17] use the hierarchic topology for defining modular verification techniques of object invariants.

Universe Types were developed by Müller and Poetzsch-Heffter [16, 15] to support modular reasoning. The type system is one of the simplest possible in the family of ownership related systems. There are three universe qualifiers: rep, peer and any, which denote the relative placement of objects in the the ownership hierarchy.

An example appears in Figure 2. 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. In terms of our example code, the Stack object 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 qualifier any abstracts over the object's position in the hierarchy: the field value in class Node is declared as any and can therefore refer to any object in the hierarchy. The code in Figure 2 also augments methods with the keywords pure and impure, qualifying read-only methods from methods that affect the state of the receiver; though not integral to the universes, these purity annotations are used to guarantee encapsulation, as we discuss later.

Thus, universe types offer a very simple system which describes the topology of the heap. In contrast with several other systems [8], universe types do not restrict access 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 impose *encapsulation*, in the sense that they guarantee that the state of an object can only be modified when one of the object's owners is the currently active object. This is the *owners-as-modifiers* property[15], and it is guaranteed by the type system by requiring that only pure methods can be called when the receiver has the universe qualifier any.

Thus, the owners-as-modifiers discipline supports modular reasoning, because it guarantees that the invariant of an object is preserved by an execution, whose receiver's owner is not an transitive owner of the former object. Modular reasoning using universes has been adopted in Spec# [2, 3]. However, there they also consider a setting where the owners may change dynamically. The use of universe types in modular reasoning is described in various literature, amongst which [14, 9, 17].

In this paper we give a formal, type theoretical description of universe types. We dis-

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

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

class Node{
  any Object value;
  peer Node Next;
}
```

Figure 2: Code augmented with Universe Qualifiers

tinguish 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:

- we believe the distinction clarifies the rationale for the type systems.
- some systems, such as those required for races detection, atomicity and deadlock detection, only require the topological properties.
- it is conceivable that one might want to add an encapsulation part onto a different topological type system.

Universe types were already introduced and proven sound in [15], 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 [9]. Extensions to universe types, such as those described in [10] (adding Generics to Universe Types), already use the type theoretic approach.

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 for UJ (Universe Java), a small Java-like language, which contains classes, inheritance, fields, methods, dynamic method binding, and casts.

The rest of the paper is organized as follows: we introduce our base language in Section 2, followed by a discussion on Universes 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 and its relation to Modular Verification. Section 7 concludes.

2 UJ Source Language

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

2.1 Source Program

The syntax of our source language is given in figure 3. We assume three infinite but distinct sets of names: one for classes, $c \in Id_c$, one for fields, $f \in Id_f$, and another for methods, $m \in Id_m$. A program is defined in terms of three partial functions, $\mathcal{F}$, $\mathcal{M}$ and $\mathcal{MBody}$, and a transitive closure relation over class names $\leq_c$; these functions and relations are global. $\mathcal{F}$ associates field names in a class to types, $\mathcal{M}$ associates method names in a class to method signatures and $\mathcal{MBody}$ associates method names in a class to method bodies. The reflexive transitive closure relation $\leq_c$ denotes class inheritance.

Source expressions, denoted by e, are a standard subset of Java. They consist of the basic value, null, the self reference this, parameter identifier, x, new object creation, $\mathsf{new}\ t$, down-casting , $(t)\ e$, field access, $e.f$, fields update, $e.f := e$, and method invocation, $e.m(e)$. Types, denoted by t, constitute a departure from standard Java. They consist of a pair, $w\,c$, where class names c are preceded by universe annotations, denoted as w. Method Signatures also deviate slightly from standard Java. They consist of a triple, denoted as $p : t_1\ (t_2)$ where t_2 is the type of the method parameter, t_1 is the return type of a method, and p is a purity tag, ranging over the set $\{\mathsf{pure}, \mathsf{impure}\}$.

Universes, as defined in [15], are denoted by w and range over the set $\{\mathsf{rep}, \mathsf{peer}, \mathsf{any}\}$. They provide *relative information* with respect to the current object. More specifically,

- rep states that a reference 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 reference constitutes part of the representation the current object belongs to. Stated otherwise, the current object and the object pointed to by the reference are *owned* by the same object (owner).
- any is a form of existential quantification over such information.

For our formalisation we need to extend Universes by another value, self, and refer to the extended set as Extended Universes, denoted by u.

- self is a specialisation of peer, referring to the object itself and not any other peer object in the direct representation the current object belongs to.

We find it convenient to identify a Universe subset called Concrete Universes, z, ranging over $\{\mathsf{self}, \mathsf{rep}, \mathsf{peer}\}$ but not any, which give us concrete representation information about references with respect to the current object. When writing source programs, we only make use of Universes w; extended universes u are used extensively in the topological type system (Section 5); concrete universes are used extensively in the encapsulation type system (Section 6).

3 Universes and Owners

Universes characterise object aliasing in a heap because they structure the heap as an acyclic ownership tree. Every object a in a heap is owned by a single owner, o, which is either

$$\begin{array}{rcl}
\mathcal{F} & : & Id_c \times Id_f \rightharpoonup Type \\
\mathcal{M} & : & Id_c \times Id_m \rightharpoonup TypeSig \\
\mathcal{MBody} & : & Id_c \times Id_m \rightharpoonup SrcExpr \\
\leq_c & : & Id_c \times Id_C \rightarrow Bool
\end{array}$$

$$\begin{array}{rcl}
e \in SrcExpr & ::= & \mathsf{this} \mid \mathsf{x} \mid \mathsf{null} \mid \mathsf{new}\ t \mid (t)\ e \\
 & \mid & e.f \mid e.f := e \mid e.m(e) \\
t \in Type & ::= & u\ c \\
w \in Universe & ::= & \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{any} \\
u \in Extended\ Universes & ::= & \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{any} \mid \mathsf{self} \\
z \in Concrete\ Universes & ::= & \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{self} \\
TypeSig & ::= & p : t\ (t) \\
p \in Purity\ Tag & ::= & \mathsf{pure} \mid \mathsf{impure}
\end{array}$$

Figure 3: Source program definition

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

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

Figure 4: Assigning Universes to Objects

another object in the heap or the root of the ownership tree, root. The *representation* of an object in a heap h is defined as all the objects it transitively owns (all the objects below it). Ownership is acyclic, that is any two distinct objects in h cannot belong one another's representation (they cannot transitively own one another). 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. (*e.g.* 1,root and 2,1, meaning 1 owned by root and 2 owned by 1 respectively).

We recall that (concrete) universes are *relative* with respect to a viewpoint and do not mean anything without such a viewpoint. In class definitions, this viewpoint of universe annotations is implicitly assumed to be the current this object (see the code in Figure 2). An object a,o can be assigned to a universe with respect to another object a',o' using the judgement

$$a', o' \vdash a, o : u \qquad (1)$$

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 u.

Example 3.1 (Universes and addresses). *In the heap depicted in Figure 1, from the point of view of 2 (owned by 1) object 3 (owned by 2) has universe* rep *that is*

$$2, 1 \vdash 3, 2 : \mathit{rep} \qquad (2)$$

Similarly, we can derive

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

Also, we can assign any *to any address from any viewpoint using rule (*ANY*). Thus,*

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

3.1 Universe Ordering

We define the following ordering for universes $u \leq_u u'$:

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

It states that both peer and rep are sub-universes of any, but they are not directly related, and also that self is a sub-universe of peer. In Lemma 3.2 we state that the universe ordering relation ($\leq_u$) is consistent with judgement (1). Thus any address that is assigned rep, peer and self can also be assigned universe any and any address that is assigned self can be assigned universe peer, as we have already seen in Example 3.1.

Lemma 3.2 (Universe Address Judgements respect Universe Ordering)**.**

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

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

3.2 Universe Composition and Decomposition

The universe information given by Universe Types is *relative* with respect to a particular viewpoint. To translate universe types from one viewpoint to another we define composition and decomposition operators over extended universes, u, and universes, w. These operators are denoted as $u \oplus w$ and $u \ominus w$ respectively and are described in Figure 5.

- Universe composition is used to determine the universe of a reference that is *twice removed* from our current viewpoint. If the second reference is outside the current viewpoint's range, then we cannot express it in terms of the concrete universes rep and peer; in such cases $u \oplus w$ is assigned to any. For instance rep $\oplus$ rep = any.
- Universe decomposition is the complement of the former operation: $u \ominus w$ returns a universe qualifier w' such that $u \oplus w' = w$ if it exists and is unique. For instance rep $\ominus$ rep = peer because rep $\oplus$ peer = rep and there is no other universe w such that rep $\oplus\, w$ = rep. When the w' in $u \oplus w' = w$ is not unique or does not exist, then $u \ominus w$ = any. Thus, rep $\ominus$ self = any.

In Lemma 3.4 we show that the intuitions of ($\oplus$) and ($\ominus$) are sound with respect to the interpretation of universes as object ownership in a heap, that is judgement (1). The proof is relegated to Appendix A.

Lemma 3.4 (Sound Universe Composition and Decomposition)**.**

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

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

		w		
	$u \oplus w$	peer	rep	any
u	self	peer	rep	any
	peer	peer	any	any
	rep	rep	any	any
	any	any	any	any

		w		
	$u \ominus w$	peer	rep	any
u	self	peer	rep	any
	peer	peer	any	any
	rep	any	peer	any
	any	any	any	any

Figure 5: Universe composition and decomposition

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

Figure 6: Runtime Syntax

Example 3.5 (Composing and Decomposing Universes)**.** *From judements (2), (3) from Example 3.1, using Lemma 3.4, we derive*

$$2, 1 \vdash 4, 2 : \mathit{rep} \qquad (5)$$

because $\mathit{rep} \oplus \mathit{peer} = \mathit{rep}$*. Conversely, from (2) and (5), using Lemma 3.4 and* $\mathit{rep} \ominus \mathit{rep} = \mathit{peer}$*, we recover (3)*

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

4 Operational Semantics

We give the semantics of UJ in terms of a small-step operational semantics. We assume an infinite set of addresses, denoted by a, b. At runtime, a value, denoted by v, may be either an address or null. Owners, denoted by o, can either be any 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 parameter identifiers x. Thus, a runtime expression is interpreted with respect to a heap, h, which gives meaning to addresses, and a stack, σ, which gives meaning to the keyword this and parameter identifier x.

A heap is defined in Figure 6 as a partial function from addresses to objects.[1] An object is denoted by the triple (o, c, flds). Every object has an immutable owner o, belongs to a fixed class c, and has a state, *flds*, which is a mutable field map (a partial function from

[1]The arrow $\rightharpoonup$ indicates partial mappings.

$$\frac{}{\sigma \vdash \mathsf{x}, h \rightsquigarrow \sigma(\mathsf{x}), h}(\text{rVar}) \qquad \frac{}{\sigma \vdash \mathsf{this}, h \rightsquigarrow \sigma(\mathsf{this}), h}(\text{rThis})$$

$$\frac{h' = h \uplus \{a \mapsto \mathbf{initO}(t, h, \sigma)\}}{\sigma \vdash \mathsf{new}\ t, h \rightsquigarrow a, h'}(\text{rNew}) \qquad \frac{h, \sigma \vdash a : t}{\sigma \vdash (t)\ a, h \rightsquigarrow a, h}(\text{rCast})$$

$$\frac{}{\sigma \vdash a.f, h \rightsquigarrow \mathbf{fields}(h, a)(f), h}(\text{rField}) \qquad \frac{h' = h[(a, f) \mapsto v]}{\sigma \vdash a.f := v, h \rightsquigarrow v, h'}(\text{rAssign})$$

$$\frac{e = \mathcal{MBody}(\mathbf{class}(h, a), m)}{\sigma \vdash a.m(v), h \rightsquigarrow \mathsf{frame}\ (a, v)\ e,\ h}(\text{rCall}) \qquad \frac{\sigma \vdash e, h \rightsquigarrow e', h'}{\sigma \vdash E[e], h \rightsquigarrow E[e'], h'}(\text{rEvalCtx})$$

$$\frac{\sigma' \vdash e, h \rightsquigarrow e', h'}{\sigma \vdash \mathsf{frame}\ \sigma'\ e, h \rightsquigarrow \mathsf{frame}\ \sigma'\ e', h'}(\text{rFrame1}) \qquad \frac{}{\sigma \vdash \mathsf{frame}\ \sigma'\ v, h \rightsquigarrow v, h}(\text{rFrame2})$$

Figure 7: Small step operational semantics

fields names to values). In the remaining text we use the following heap operations:

$$\mathbf{owner}(h, a) \overset{\mathbf{def}}{=} h(a){\downarrow_1} \qquad \mathbf{class}(h, a) \overset{\mathbf{def}}{=} h(a){\downarrow_2} \qquad \mathbf{fields}(h, a) \overset{\mathbf{def}}{=} h(a){\downarrow_3}$$

$$\begin{aligned} h(a.f) &\overset{\mathbf{def}}{=} \mathbf{fields}(h, a)(f) \\ h[(a, f) \mapsto v] &\overset{\mathbf{def}}{=} h\left[a \mapsto \big(\ \mathbf{owner}(h, a),\ \mathbf{class}(h, a),\ \mathbf{fields}(h, a)[f \mapsto v]\ \big)\right] \\ h \uplus \{a \mapsto (o,\ c,\ \mathit{flds})\} &\overset{\mathbf{def}}{=} h[a \mapsto (o,\ c,\ \mathit{flds})] \qquad \text{if } a \notin \mathbf{dom}(h) \end{aligned}$$

The first three operations extract the components making up an object. 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. The final operation on heaps is *heap extension* with a new mapping from a *fresh* address a.

A stack σ consists of an address and a value (a, v). The address a denotes the current 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 stacks

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

For evaluating method calls, we require to push and pop new address and value pairs on 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 σ.

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

$$\sigma \vdash e, h \rightsquigarrow e,' h' \qquad (6)$$

in terms of the reduction rules in Figure 7. When creating a new object in a heap h, its owner is initialised relative to the universe specified in its declared type t and the current stack σ, whereas all its field values are initialised to null; this initialisation operation is handled by the following function:

$$\mathbf{initO}(u\,c, h, \sigma) \overset{\mathbf{def}}{=} (u_{h,\sigma},\ c,\ \{f \mapsto \mathsf{null} \mid \mathcal{F}(c, f) = t\})$$

where

$$u_{h,\sigma} \overset{\mathbf{def}}{=} \begin{cases} \sigma(\mathsf{this}) & u = \mathsf{rep} \\ \mathbf{owner}(h, \sigma(\mathsf{this})) & u = \mathsf{peer} \end{cases}$$

The above function is partial: it is only defined for universes rep and peer since the owner of a new object cannot be determined if the universe is any. Most of the rules in Figure 7 are more or less straightforward. In (rCall) a method call launches a sub-frame with sub-stack σ' 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.

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

$$\begin{array}{lll}
\sigma \vdash \mathbf{this}.\mathit{push(7)}, h & \rightsquigarrow\ \mathit{2.push(7)}\ ,\ h & (\text{rThis}) \\
 & \rightsquigarrow \mathit{frame}\ \sigma'\ \mathbf{this}.\mathit{top}{:=}\ \mathbf{new\ rep}\ \mathit{Node(x,\ \mathbf{this}.top)}\ ,\ h & (\text{rCall}) \\
 & \rightsquigarrow \mathit{frame}\ \sigma'\ \mathit{2.top}{:=}\ \mathbf{new\ rep}\ \mathit{Node(x,\ \mathbf{this}.top)}\ ,\ h & (\text{rThis}) \\
 & \rightsquigarrow \mathit{frame}\ \sigma'\ \mathit{2.top}{:=}\ \mathbf{new\ rep}\ \mathit{Node(7,\ \mathbf{this}.top)}\ ,\ h & (\text{rVar}) \\
 & \rightsquigarrow \mathit{frame}\ \sigma'\ \mathit{2.top}{:=}\ \mathbf{new\ rep}\ \mathit{Node(7,\ 2.top)}\ ,\ h & (\text{rThis}) \\
 & \rightsquigarrow \mathit{frame}\ \sigma'\ \mathit{2.top}{:=}\ \mathbf{new\ rep}\ \mathit{Node(7,\ 3)}\ ,\ h & (\text{rField}) \\
 & \rightsquigarrow \mathit{frame}\ \sigma'\ \mathit{2.top}{:=}\ \mathit{13}\ ,\ h' & (\text{rNew}) \\
 & \rightsquigarrow \mathit{frame}\ \sigma'\ \mathit{13}\ ,\ h'[(2, \mathit{top}) \mapsto \mathit{13}] & (\text{rAssign}) \\
 & \rightsquigarrow\ \mathit{13}\ ,\ h'[(2, \mathit{top}) \mapsto \mathit{13}] & (\text{rFrame2})
\end{array}$$

where $\sigma' = (2, 7)$, $h' = h \uplus \{13 \mapsto (2, Node, \{\mathsf{value} \mapsto 7,\ \mathsf{next} \mapsto 3\})\}$ and 13 is a fresh address in the heap h.

5 Topological Types

In this section we define the Topological type system for UJ. The formalism is based on earlier work[15, 9] but has some differences: as we said in the introduction, we here focus on the hierarchical topology imposed by Universes but do not enforce the *owner-as-modifier* property at this stage[3] - this is dealt with later in Section 6. The main result of this section is Subject Reduction, stating that a type assigned to an expression and the ownership hierarchical heap structure is preserved during execution.

5.1 Subtyping and Type Composition/Decomposition

As was already stated in Section 2.1, types, t, are made up of two components: a universe u and a class name c. For every program we already assume a class inheritance reflexive transitive closure on class names, $\leq_c$. Using the ordering universe relation $\leq_u$ of Section 3.1 and $\leq_c$ we define the subtype relation as:

$$u\ c \leq u'\ c' \stackrel{\mathbf{def}}{=} u \leq_u u' \text{ and } c \leq_c c' \qquad (7)$$

[2]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 addressed here, derived from the type of the new object and the current active object.

[3]We therefore allow assignments and impure method calls on any objects in the heap.

We extend ($\oplus$) and ($\ominus$), defined earlier in Section 3.2 for universes, to types as $u \oplus t$ and $u \ominus t$ using the straightforward definitions

$$\begin{aligned} u \oplus (u'\ c) &\overset{\mathbf{def}}{=} (u \oplus u')\ c \\ u \ominus (u'\ c) &\overset{\mathbf{def}}{=} (u \ominus u')\ c \end{aligned}$$

We use these two auxillary operators whenever we need to change the viewpoint of the types. We prove that our Universe composition and decomposition operations on types respect the subtype relation (7). The proof is relegated to the Appendix.

Lemma 5.1 (Universe Composition and Decomposition preserves Subtyping)**.**

$$t' \leq t \implies \begin{cases} u \oplus t' \leq u \oplus t \\ u \ominus t' \leq u \ominus t \end{cases}$$

5.2 UJ Source Language Types

We typecheck UJ source expressions with respect to a typing environment Γ, which keeps typing information for this and the method parameter x. The universe type derived from this judgement is interpreted with respect to the current this in Γ.

Definition 5.2 (Type Environment)**.** *A type environment Γ consists of a pair of types, (t, t'), assigning types to the current active object* this *and the parameter* x *respectively. We define the following operations on Γ:*

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

The source expression type-judgement takes the form

$$\Gamma \vdash e : t$$

denoting that expression e has Universe Type t with respect to the typing environment Γ. It 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 judgements (Field), (Assign) and (Call) which use the auxillary operations $u \oplus t$ and $u \ominus t$ defined in Section 3.2 to *translate* types from one viewpoint to another.

Example 5.3 (Type Viewpoint Translation)**.** *If $\Gamma \vdash$ this.top : rep Node and field* next *in class* Node *has type* peer Node*, then using (*Field*), from the viewpoint Γ, the dereference* this.top.next *has type* rep $\oplus$ (peer Node) = rep Node*, that is*

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

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

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

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

$$\mathcal{F}(\mathit{Node}, \mathit{next}) = \mathit{peer}\ \mathsf{Node} = \mathit{rep}\ \ominus (\mathit{rep}\ \mathsf{Node})$$

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

$$\frac{\Gamma \vdash e : t'}{\Gamma \vdash (t)\ e : t}\,(\textsc{Cast}) \qquad \frac{\Gamma \vdash e : t' \quad t' < t}{\Gamma \vdash e : t}\,(\textsc{Sub}) \qquad \frac{w \neq \mathsf{any}}{\Gamma \vdash \mathsf{new}\ w\ c : w\ c}\,(\textsc{New})$$

$$\frac{\Gamma \vdash e : u\ c \quad \mathcal{F}(c,f) = t}{\Gamma \vdash e.f : u \oplus t}\,(\textsc{Field}) \qquad \frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{F}(c,f) = u \ominus t}{\Gamma \vdash e.f := e' : t}\,(\textsc{Assign}) \qquad \frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{M}(c,m) = p : t_r\ (u \ominus t)}{\Gamma \vdash e.m(e') : u \oplus t_r}\,(\textsc{Call})$$

Figure 8: Source type system

The source expression type judgement allows us to formally define well-formed classes by requiring consistency between subclasses, that is field types of the class concord with the field types of any superclass of the class and method signatures are specialisations of the signatures of overridden methods, and that method bodies are consistent with the signature of that method.

The types in a method signature are meant to be interpreted with respect to this, the current active object. Thus when typing a method body, the observer is implicitly assumed to be this, even though there are no actual objects at compile-time. We assign the self (Extended) Universe to $\Gamma(\mathsf{this})$ when type-checking method bodies because we want the universes of the method calls and field accesses in the method bodies to be exactly the annotation w given in the class. We note that $\forall w.\ \mathsf{self} \oplus w = w$.[4]

Definition 5.4 (Well-Formed Class)**.**

$$\frac{\begin{array}{l} \forall c' \geq_c c\ .\ \mathcal{F}(c',f) = t \Longrightarrow \mathcal{F}(c,f) = t \\ \forall c' \geq_c c\ .\ \mathcal{M}(c',m) = p : t'_r\ (t'_x) \Longrightarrow \begin{cases} \mathcal{M}(c,m) = p : t_r\ (t_x) \\ \textit{where}\ t_r \leq t'_r\ \textit{and}\ t_x \geq t'_x \end{cases} \\ \forall \mathcal{M}(c,m) = p : t_r\ (t_x)\ .\ (\mathit{self}\ c, t_x) \vdash \mathcal{MBody}(c,m) : t_r \end{array}}{\vdash c}\,(\text{WF}\textsc{Class})$$

The program is said to be *well-formed* if all the defined classes are well-formed.

5.3 Runtime Types

We define a type system for runtime expressions. These are type-checked with respect to the base stack frame σ, which contains actual values for the current receiver this and the parameter x. Since runtime expressions also contain addresses, we also need to typecheck 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 σ, through a judgement of the form

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

[4]The inquisitive reader may wonder whether peer could have been used instead of self to type $\Gamma(\mathsf{this})$. We note that peer would cause us to loose information when the universe w is rep, since $\mathsf{peer} \oplus \mathsf{rep} = \mathsf{any}$, thereby making the type system unnecessarily restrictive.

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

$$\frac{h, \sigma \vdash e : t'}{h, \sigma \vdash (t)\ e : t}(\text{tCast}) \qquad \frac{\begin{array}{l} h, \sigma \vdash e : t' \\ t' < t \end{array}}{h, \sigma \vdash e : t}(\text{tSub}) \qquad \frac{}{h, \sigma \vdash \mathsf{new}\ t : t}(\text{tNew})$$

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

$$\frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ \mathcal{F}(c, f) = t \end{array}}{h, \sigma \vdash e.f : u \oplus t}(\text{tField}) \qquad \frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ h, \sigma \vdash e' : t \\ \mathcal{F}(c, f) = u \ominus t \end{array}}{h, \sigma \vdash e.f := e' : t}(\text{tAssign})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ h, \sigma \vdash e' : t \\ \mathcal{M}(c, m) = p : t_r\ (u \ominus t) \end{array}}{h, \sigma \vdash e.m(e') : u \oplus t_r}(\text{tCall}) \qquad \frac{\begin{array}{l} h, \sigma' \vdash e : t \\ h, \sigma \vdash \sigma'(\mathsf{this}) : u \end{array}}{h, \sigma \vdash \mathsf{frame}\ \sigma'\ e : u \oplus t}(\text{tFrame})$$

Figure 9: Runtime type system

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 components of t in the judgement are 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 obtained using judgement (1) of Section 3. The three rules (tField), (tAssign) and (tCall) use the universe composition and decomposition operators in the same way as their static-expression counterparts in Figure 8. The new rule (tFrame) also uses the composition universe operation to translate the type of the sub-expression, obtained with respect to the sub-stack of the frame, to the current frame's viewpoint.

Lemma 5.5 states the composition and decomposition operations correctly characterise the translation of value that types from one viewpoint, denoted by σ in the runtime type system, to another. The viewpoint tranlations of Lemma 5.5 trivially apply to null values since we assign artitrary types to such values using rule (tNull). The proof is relegated to the Appendix.

Lemma 5.5 (Determining the relative Universe Types of Values)**.**

(i) If $h, \sigma \vdash a : u\ \_$ *and* $h, (a, \_) \vdash v : t$ *then* $h, \sigma \vdash v : u \oplus t$

(ii) If $h, \sigma \vdash a : u\ \_$ *and* $h, \sigma \vdash v : t$ *then* $h, (a, \_) \vdash v : u \ominus t$

Example 5.6 (Relative Viewpoints in a Heap)**.** *In Figure 1, using* (2) *and* (3) *from Example 3.1 and rule (*tAddr*) we derive*

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

From Lemma 5.5(i) we immediately derive

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

Conversely, using $h, (2, \_) \vdash 3 : \mathsf{rep\ Node}$*,* $h, (2, \_) \vdash 4 : \mathsf{rep\ Node}$ *and Lemma 5.5(ii) we can recover* $h, (3, \_) \vdash 4 : \mathsf{peer\ Node}$*.*

At this point, we have enough machinery to define well-formed addresses and heaps. An address is well-formed in a heap whenever its owner is valid (that is it is another address in the heap or root) and the type of its fields respect the type of the fields defined in $\mathcal{F}$. As described in Definition 5.8, a heap is well-formed, denoted as $\vdash h$, if transitive ownership, denoted by $\mathbf{owner}^*(h, o)$, is acyclic and all its addresses are well-formed.

Definition 5.7 (Transitive Ownership)**.**

$$\begin{array}{lcl} \boldsymbol{owner}^*(h,o) & \overset{def}{=} & \begin{cases} \{o\} \cup \boldsymbol{owner}^*(h, \boldsymbol{owner}(h,o)) & o \neq \mathsf{root} \\ \{\mathsf{root}\} & o = \mathsf{root} \end{cases} \\ \boldsymbol{owner}^+(h,o) & \overset{def}{=} & \boldsymbol{owner}^*(h,o) \backslash \{o\} \end{array}$$

Definition 5.8 (Well-Formed Addresses and Heaps)**.**

$$\frac{\begin{array}{l} \boldsymbol{owner}(h,a) \in (\boldsymbol{dom}(h) \cup \{\mathsf{root}\}) \\ \boldsymbol{class}(h,a) = c \\ \mathcal{F}(c,f) = t \implies h,(a,\_) \vdash h(a.f) : t \end{array}}{h \vdash a} \text{(WFAddr)}$$

$$\frac{\begin{array}{l} (a,b \in \boldsymbol{dom}(h) \ \wedge \ a \in \boldsymbol{owner}^*(h,b) \ \wedge \ b \in \boldsymbol{owner}^*(h,a)) \implies a = b \\ a \in \boldsymbol{dom}(h) \implies h \vdash a \end{array}}{\vdash h} \text{(WFHeap)}$$

We conclude the section by showing the correspondence between the source-expression type system and runtime-expression type system. The Substitution Lemma 5.9 states that, with respect to a suitable stack σ, where $\sigma(\mathsf{this})$ and $\sigma(\mathsf{x})$ match the respective type assignments in Γ, a well-formed source expression is also a well-formed runtime expression. Despite the name used, we note that no "substitution" occurs in the expression itself, which is the same in both type judgements of the Lemma. The proof is relegated to Appendix B.

Lemma 5.9 (Substitution)**.**

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

5.4 Subject Reduction

In this section we state and prove the first main result of the paper. It states that if a well-formed runtime expression e reduces with respect to a stack ,σ, and a well-formed heap ,h, then the resulting expression preserves its type, and the resulting heap preserves its well-formedness.

Theorem 5.10 (Subject Reduction)**.** *If a program is well-formed then*

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

(Proof Sketch): 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. The owner and class components of an object in a heap are immutable during execution (Lemma B.1). During execution we never remove existing addresses from the heap (Lemma B.2). Earlier in Section 4 we discussed how reduction rules make use of two operations to update a heap in the form of $h \uplus \{a \mapsto \mathbf{initO}(t, h, \sigma)\}$ and $\vdash h[(a, f) \mapsto v]$. Lemma B.3 shows that the heap extension operation creates a new object with the requested type in the heap. Lemma B.4 states that under appropriate conditions, heap update and heap extension operations preserve heap well-formedness. Lemma B.5 states that the type judgement $h, \sigma \vdash a : u\ c$ implies that the class of a in h is a sub-class of c and that a has universe u from the current viewpoint $\sigma(\mathsf{this})$. We relegate these four Lemmas to Appendix B. The proof uses also Lemma 5.5 and Lemma 5.9 from Section 5.3. The proof of main cases involved in the proof is given in Appendix B. □

6 Encapsulation

In section 5 we showed how the *topological* type system guarantees that the topology of the objects on 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 property [15], which localises the effects of execution in a heap with respect to the current active object. We then show how this result can be used to deduce preservation of object invariants.

Notation: For the subsequent discussion we find it convenient to define the notation and predicates, summarised in Definition C.1. The expression $C[e]$ denotes that it contains sub-expression e in some expression context $C[\cdot]$. The notation $h(a.f_1 \ldots f_n)$ is used as a shorthand when we want to refer to multiple dereferencing in a heap. The predicate $\mathbf{rFlds}(c,\ \mathsf{f}_1 \ldots \mathsf{f}_n)$ holds if a list of field accesses yield an object within the representation of an object of class c, where the first field access f_1 is defined in class c (not inhereted from superclasses of c); this is denoted as $\mathcal{DF}(c, f_1)$. The predicate $\mathbf{rFlds}(c,\ \mathsf{f}_1 \ldots \mathsf{f}_n)$ thus holds if the first field f_1 is a rep field defined in c, and the remaining field accesses are either rep or peer. The predicate $\mathbf{pure}(c,\ m)$ holds if m is pure in c. The formal definitions are given in Definition C.1 in the Appendix.

6.1 Modular Verification and Universe Types

Object invariants are often used in object-oriented program specification and verification. *Modularity* aims to allow subsets of the code (e.g. classes, modules, functions) to be checked in isolation, without consideration of the whole program. There have been various approaches to achieve modularity in verification[18, 4, 15].

For the case of object-oriented programs, [17] suggest:

Restrictions on Invariants: They can only be defined in terms of a restricted form of field accesses.

Restrictions on Effects: Methods can only affect the invariants of a restricted subset of objects during their execution.

These restrictions are expressed in terms of the hierarchic heap topology introduced through universes.

Ownership Admissible Invariants [17] for an object of class c may only be defined in terms of fields that fall under the object's representation(i.e. itself and the objects it transitively owns). Such invariants also have the additional restriction that the first field access **this** . f needs to be defined in class c directly, that is $\mathcal{DF}(c, f)$; this allows for subclass separation [17] whereby we can verify the methods of class c with respect to the invariant of class c without having to re-evaluate methods inherited from superclasses of c. More formally, an invariant in class c may only contain:

1. **this** . g where $\mathcal{DF}(c, \mathsf{g})$.
2. **this** . $\mathsf{f}_1 \ldots \mathsf{f}_n$.g where $\mathbf{rFlds}(c, \mathsf{f}_1 \ldots \mathsf{f}_n)$ and g is a field defined in the class of type $\mathcal{F}(\ldots \mathcal{F}(c, f_1), f_n)$.

With respect to the effect restrictions, an object x in a heap h is allowed to affect

1. invariants of objects its owner transitively owns, $\mathsf{inv}(y)$ where $\mathbf{owner}(h, x) \in \mathbf{owner}^*(h, y)$; note that this includes $\mathsf{inv}(x)$
2. invariants of objects which transitively own it, $\mathsf{inv}(y)$ where $y \in \mathbf{owner}^+(h, x)$.

In its current state, the Universe type system can only describe *direct ownership*. Hence the above effect restrictions translate to the *owner-as-modifier* property [9] whereby an object is only allowed to

1. directly assign to its own fields, the fields of rep objects (which it owns) and the fields of peer objects (which its owner owns).
2. call impure methods on itself, on objects it owns and on peer objects.

Example 6.1 (Universe, Invariants and Effects)**.**

```
class A{
   rep A fra;
    ...
}
class B extends A{
   rep B frb;   peer B fp;   any B fa;
    ...
}
```

As an example, consider the class definitions A and B above. In terms of the restriction of invariants, the invariant of class B may mention the following fields

- **this** . *frb*, **this** . *fp and* **this** . *fa - its fields*
- **this** . *frb ... fr . fp* , **this** . *frb . fp ... fp. fr and* **this** . *frb . fp ... fr . fp. fa - fields of objects in its representation where the first field access* **this** . *frb is defined in B and not inherited from A.*

The invariant of class B may not mention **this** . *fp. fr and* **this** . *fr . fa . fr because they consitute fields of objects that do not belong to its representation. It may not mention* **this** . *fra ... fr . fp either because the first field access fra is defined in the superclass A.*

In terms of the restriction on effects, let us consider the heap depicted in Figure 10. If 5 is the active object (i.e. the current **this***), then it may affect:*

- *inv(5) - itself*

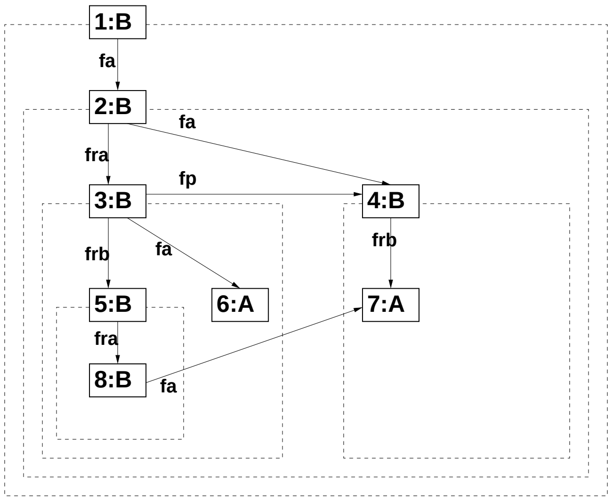

Figure 10: A heap

- *inv(6) and inv(8) - objects in its owner's representation*
- *inv(3), inv(2) and inv(1) - objects which transitively own it*

It however cannot affect the invariants of the objects 4 and 7.

6.2 Encapsulation Types

Encapuslation Types impose extra restrictions so as to support the owner-as-modifier approach and guarantee the restriction on effects. We define an encapsulation judgement for expressions, $\Gamma \vdash_{\mathbf{enc}} e$, reflecting the expression restrictions imposed in [17] to control the effects of method executions on the invariants of other objects, as discussed above. These restrictions state that for an expression e to respect encapsulation, it can only assign to and call impure methods on itself, on rep receivers or peer receivers. Recall that z is a Universe metavariable that ranges over self, rep and peer only. To separate between pure and impure methods we require a purity judgement for expressions $\Gamma \vdash_{\mathbf{pure}} e$. An expression e is pure if it never assigns to fields and *only* calls pure methods.

Definition 6.2 (Encapsulation and Purity for Source Expressions)**.**

$$\frac{\begin{array}{l}\Gamma \vdash e : t \\ e = C[e_1.f := e_2] \Longrightarrow \Gamma \vdash e_1 : z \\ e = C[e_1.m(e_2)] \Longrightarrow \Gamma \vdash e_1 : z \ \vee\ (\textbf{\textit{pure}}(c, m) \textit{ where } \Gamma \vdash e_1 : c)\end{array}}{\Gamma \vdash_{\textbf{\textit{enc}}} e} \ (\textsc{Enc})$$

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

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 neither assign to fields nor call impure methods and impure methods have bodies that only assign to fields and call impure methods on rep or peer receivers. We recall that according to Definition 5.4, pure methods are only overridden by pure methods, and similarly for impure methods. A Program P is well-formed with respect to encapsulation if all its classes are encapsulated

Definition 6.3 (Encapsulated Well-Formed Class)**.**

$$\frac{\begin{array}{l}\vdash c \\ \forall m.\ \begin{array}{rcl} \textbf{\textit{pure}}(c,\, m) & \Longrightarrow & (\textit{self}\ c, \_) \vdash_{\textbf{\textit{pure}}} \mathcal{MBody}(c,m) \\ \neg\textbf{\textit{pure}}(c,\, m) & \Longrightarrow & (\textit{self}\ c, \_) \vdash_{\textbf{\textit{enc}}} \mathcal{MBody}(c,m)\end{array}\end{array}}{\vdash_{\textbf{\textit{enc}}} c}\ (\text{WFEncClass})$$

$$\vdash_{\textbf{\textit{enc}}} P \implies \forall c \in Id_c \vdash_{\textbf{\textit{enc}}} c$$

We also define encapsulation and purity judgements for *runtime expressions* subject to a heap h and a stack σ; these judgements are denoted as $h, \sigma \vdash_{\textbf{enc}} e$ and $h, \sigma \vdash_{\textbf{pure}} e$ 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, $h, \sigma \vdash_{\textbf{enc}} \mathsf{frame}\ \sigma'\ e'$, requires that the receiver in σ', that is $\sigma'(\mathsf{this})$, is a rep,peer or self of that in σ. This condition is expressed through the predicate $h \vdash \sigma' \preceq_{\textbf{enc}} \sigma$, defined below.

Definition 6.4 (Frame Encapsulation)**.**

$$h \vdash \sigma' \preceq_{\textbf{\textit{enc}}} \sigma \quad \overset{\textbf{\textit{def}}}{=} \quad h, \sigma \vdash \sigma'(\textit{this}) : z$$

Definition 6.5 (Purity and Encapsulation for Runtime Expressions)**.**

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

$$\frac{\begin{array}{l} h, \sigma \vdash e : t \\ e = C[\textit{frame}\ \sigma_1\ \ e_1] \Longrightarrow (h \vdash \sigma_1 \preceq_{\textbf{\textit{enc}}} \sigma \ \wedge\ h, \sigma_1 \vdash_{\textbf{\textit{enc}}} e_1) \ \vee\ (h, \sigma_1 \vdash_{\textbf{\textit{pure}}} e_1) \\ e = C[e_1.f := e_2] \Longrightarrow h, \sigma \vdash e_1 : z \\ e = C[e_1.m(e_2)] \Longrightarrow h, \sigma \vdash e_1 : z \ \vee\ (\textbf{\textit{pure}}(c,\, m)\ \textit{where}\ h, \sigma \vdash e_1 : c) \end{array}}{h, \sigma \vdash_{\textbf{\textit{enc}}} e}\ (\text{rEnc})$$

Our proof for encapsulation starts by showing a correspondence between source expression encapsulation and purity and runtime expression encapsulation and purity (with respect to a suitable stack) and that purity implies encapsulation. These properties are stated in Lemma C.2 and Lemma C.3, which we relegate to the Appendix. We also need to prove two core (but fairly standard) intermediary results. Subject reduction for purity and encapsulation, Lemma C.5, states that if a runtime expression respects purity (or encapsulation) the resulting expression after reduction will still respect purity (or encapsulation). Safety for purity and encapsulation, Lemma C.6, states that a pure expression never changes the state of existing objects and that an expression that respects encapsulation only changes objects that are transitively owned by the *owner of* the current active object. These Lemmas are also relegated to the Appendix.

We can now prove the Encapsulation Theorem, the main result of this section. It states that if an expression respects encapsulation (with respect to some h, σ), then during its execution it will only update objects that form part of the representation of the owner of the current active object.

Theorem 6.6 (Encapsulation)**.**

$$\left.\begin{array}{l} h, \sigma \vdash_{\boldsymbol{enc}} e \\ \sigma \vdash e, h \rightsquigarrow^* e', h' \\ a \in \boldsymbol{dom}(h) \\ \boldsymbol{owner}(h, \sigma(\mathsf{this})) \notin \boldsymbol{owner}^*(h, a) \end{array}\right\} \implies h(a.f) = h'(a.f)$$

6.3 Modular Invariant Preservation

We conclude by relating the Encapsulation Theorem 6.6 to modular verification. We (abstractly) define *legal invariants* as those satisfying the constraints outlined earlier in Section 6.1.

Definition 6.7 (Predicate Satisfaction)**.** *We assume a mapping from pairs of addresses and classes to predicates*

$$\mathsf{inv} \in \mathit{INV} : \mathit{Addr} \times \mathit{Id}_c \rightharpoonup \mathit{Pred}$$

We also assume a predicate satisfaction relation

$$\models \subseteq \mathit{Heap} \times \mathit{Pred}$$

In particular, $h \models \mathsf{inv}(a, c)$ represents the satisfaction of the invariant of object at address a at class c in heap h.

Definition 6.8 (Legal Invariants)**.** *An invariant $\mathsf{inv}(a, c)$ is legal if it satisfies the property:*

$$\begin{array}{l} \forall h, h' \\ \qquad \left.\begin{array}{l} \cdot\ h \models \mathsf{inv}(a, c) \\ \cdot\ h(a.f) = h'(a.f) \\ \cdot\ h(a.f_1 \ldots f_n.g) = h'(a.f_1 \ldots f_n.g) \\ \qquad \textit{where } \boldsymbol{rFlds}(c,\ f_1 \ldots f_n) \end{array}\right\} \implies h' \models \mathsf{inv}(a, c) \end{array}$$

We finally prove the Modular Invariant Preservation Theorem stating that execution preserves invariants of unrelated objects, that is objects that neither belong to the representation of the active object's owner nor transitively own the active object.

Theorem 6.9 (Modular Invariant Preservation)**.**

$$\left.\begin{array}{l} h \models \mathsf{inv}(a, c) \\ \mathsf{inv}(a, c)\ \textit{legal} \\ h, \sigma \vdash_{\boldsymbol{enc}} e \\ a \notin \boldsymbol{owner}^*(h, \sigma(\mathsf{this})) \\ \boldsymbol{owner}(h, \sigma(\mathsf{this})) \notin \boldsymbol{owner}^*(h, a) \\ \sigma \vdash e, h \rightsquigarrow^* e', h' \end{array}\right\} \implies h' \models \mathsf{inv}(a, c)$$

Proof. Use Theorem 6.6 and the conditions defining $h \models \mathsf{inv}(a)$ in Definition 6.8. □

Example 6.10 (Modular Invariant Preservation)**.** *Recall the heap depicted in the beginning in Figure 1. If we take σ such that the active object $\sigma(\mathsf{this}) = 2$, then we can show that $h, \sigma \vdash_{\mathit{enc}} 2.\mathit{push}(7)$ We can also show that all the classes in Figure 2 are well formed with respect to encapsulation. Thus if we consider the case where $a = 10$, Theorem 6.9 guarantees that if $h \models \mathit{inv}(10, \mathit{Bag})$ and we execute expression $\sigma \vdash 2.push(7), h \rightsquigarrow^* e', h'$, then $h' \models \mathit{inv}(10, \mathit{Bag})$ for the resultant heap h'. The same argument can be applied for $a \in \{6, 7, 8, 9, 11, 12\}$.*

However, we cannot derive the same conclusion for $a \in \{3, 4, 5\}$ because these object are in the representation of the active object 2. We also cannot apply Theorem 6.9 for $a = 1$ since $\mathit{inv}(1, \mathit{Bag})$ may mention the fields $\mathbf{this}.\mathit{state}$, $\mathbf{this}.\mathit{state}.\mathit{next}$ and $\mathbf{this}.\mathit{state}.\mathit{next}.\mathit{next}$ which are objects whose states may have been affected by the execution of expression 2.push(7).

7 Conclusion

We given an alternative formalisation of the Universe type system in two steps, by first presenting a Topological Type System which preserves the ownership topology and the augmenting it to the Encapsulation Type System, which can be lead to modular reasoning about programs. This two-step formalisation primarily allows for seprations of concerns when extending this work; some extensions and applications of Universes do not require encapsulation properties such as the owner-as-modifier, introduced in the latter step. Also, the two-step formalisation permits a gentler presentation of the mathematical machinery we develop. Both these factors facilitate the adoption of the work as a starting point for further work.

Our formalisation also presented some novel techniques. We introduced a distinction between the original universes as defined in [15] and an extended version which includes the specialisation of the peer universe called self; extended universes lead to more succint definitions such as that for well-formed classes. We also introduced the notion of universe composition and decomposition, which was then used extensively for type manipulation in the typing rules. Both these aspects were already present in earlier work on Universes; we merely made them more explicit thereby facilitating their understanding.

We also demonstrated that our formalisation is ameneable to formal proofs. We proved subject reduction (for both Topological and Encapsulation Type Systems) for a small-step semantics of a strict subset of Java. We also showed how our formalisation can be used for modular verification of object invariants.

7.1 Future Work

We plan use the formalisation of the Universe type system as a basis for various extensions. First, we plan to extend the type system to handle Generics in Java[12]. We also plan to exploit the representation hierarchical structure imposed on object in the heap to help us manage object locking in a concurrent Java setting and to guarantee atomicity[11, 5]. Finally, we plan to adapt the type system to the bytecode translation of the Java code directly. This will permit the use of universes for the verification of mobile bytecode in a Proof-Carrying-Code architecture such as the one proposed by Mobius [13].

References

[1] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped Types and Aspects for Real-Time Java. In *ECOOP 06*, pages 124–147, 2006.

[2] M. Barnett, R. DeLine, M. Fahndrich, K. Rustan, M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, June 2004.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices: CASSIS04*, pages 49–69, 2005.

[4] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 280–293, New York, NY, USA, 2005. ACM Press.

[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)*, November 2002.

[6] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223, New York, NY, USA, 2003. ACM Press.

[7] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02)*, pages 292–310, Seattle, Washington, USA, November 2002. ACM Press.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18–22 1998. ACM Press.

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

[10] W. Dietl, Drossopoulou S., and P. Müller. Generic universe types. FOOL/WOOD'07, Jan 2007.

[11] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[12] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[13] Global Computing Proactive Initiative. Mobius: Mobility, ubiquity and security. `http://mobius.inria.fr/`. IST-15905.

[14] 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 *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[15] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[16] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Fernuniversität Hagen, 1999. Technical Report 263, Available from `sct.inf.ethz.ch/publications`.

[17] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[18] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.

A Proofs for Subtyping

Lemma 5.1 (Universe Composition and Decomposition preserves Subtyping)

$$t' \leq t \implies \begin{cases} u \oplus t' \leq u \oplus t \\ u \ominus t' \leq u \ominus t \end{cases}$$

Proof. By case analysis of u and the relation $u < u'$. □

Lemma 3.2 (Universe Address Judgements respect Universe Subtyping)

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

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

Lemma 3.4 (Sound Universe Composition and Decomposition)

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

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

Proof. Case analysis of u and u'. □

B Proofs for Type Soundness

Lemma 5.9 (Substitution)

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

Proof. Induction over the structure of $\Gamma \vdash e : t$. □

Lemma B.1 (Object Owner and Class Preservation)**.**

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

Proof. By case analysis of the reduction rules. □

Lemma B.2 (Heap Domain Inclusion)**.**

$$\sigma \vdash e, h \rightsquigarrow e', h' \implies \boldsymbol{dom}(h) \subseteq \boldsymbol{dom}(h)'$$

Proof. By case analysis of $\sigma \vdash e, h \rightsquigarrow e', h'$ □

Lemma B.3 (Soundness of New Object Creation)**.**

$$\left.\begin{array}{l} a \notin \boldsymbol{dom}(h) \\ u \in \{\mathsf{rep},\ \mathsf{peer}\} \end{array}\right\} \implies h \uplus \{a \mapsto \boldsymbol{initO}(u\,c, h, \sigma)\},\, \sigma \vdash a : u\,c$$

Proof. By case analysis of u, the operation $\mathbf{initO}(u\,c, h, \sigma)$ and using rule (tAddr) ☐

Lemma B.4 (Heap Operations and Well-Formedness)**.** *If* $\vdash h$ *then*

(i) If $u \in \{\mathsf{rep}, \mathsf{peer}\}$ *then* $\vdash h \uplus \{a \mapsto \boldsymbol{initO}(u\,c, h, \sigma)\}$

(ii) If $h, \sigma \vdash a : u\,c$, $\mathcal{F}(c, f) = t$ *and* $h, (a, \_) \vdash v : t$ *then* $\vdash h[(a, f) \mapsto v]$

Proof. The proof for (i) follows from the assumption $\vdash h$ and the definitions of heap extension, object initialisation, and the typing rule (tNull). The proof for (ii) follows from $\vdash h$, the assumptions $h, \sigma \vdash a : u\,c$, $\mathcal{F}(c, f) = t$ and $h, (a, \_) \vdash v : t$ and from the definition of heap update. ☐

Lemma B.5 (Extracting information from Address type judgements)**.** *If* $h, \sigma \vdash a : u\ c$ *then*

(i) $\boldsymbol{class}(h, a) \leq c$

(ii) $\sigma(\mathsf{this}), \boldsymbol{owner}(h, \sigma(\mathsf{this})) \vdash a, \boldsymbol{owner}(h, a) : u$

Proof. Uses Lemma 3.2, (tADDR) and (tSUB) ☐

Lemma 5.5 (Determining the relative Universe Types of Values)

(i) If $h, \sigma \vdash a : u\ \_$ *and* $h, (a, \_) \vdash v : t$ *then* $h, \sigma \vdash v : u \oplus t$

(ii) If $h, \sigma \vdash a : u\ \_$ *and* $h, \sigma \vdash v : t$ *then* $h, (a, \_) \vdash v : u \ominus t$

Proof. Uses Lemma B.5(ii) to extract universe determination judgement for addresses. We have two cases for v

- If $v = \mathsf{null}$ then we can trivially derive any type judgement using (tNull).
- If $v = a$ we use Lemma B.5(ii) to extract universe determination judgement for address. Then we apply Lemma 3.4 as required.

☐

Lemma B.6 (Heap operations preserve value types)**.** *If* $h, \sigma \vdash v : t$ *then*

(i) $h[(a, f) \mapsto v'], \sigma \vdash v : t$

(ii) $h \uplus \{a \mapsto (o, c, \mathit{flds})\}, \sigma \vdash v : t$

Proof. There are two sub-cases:

$v =$ **null:** we can still derive the type judgement using (tNull).

$v = a$**:** Neither of the operations (i) and (ii) change the ownership and class information of an object in a heap, as we saw in Lemma B.1. Thus we can still derive $h[(a, f) \mapsto v'], \sigma \vdash a : t$ and $h \uplus \{a \mapsto (o, c, \mathit{flds})\}, \sigma \vdash a : t$ using (tAddr)

☐

Theorem 5.10 (Subject Reduction)

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

Proof. By induction over the structure of

$$h, \sigma \vdash e : t \tag{8}$$

The most interesting cases are when the last rules used to derive (8) are (tField),(tAssign) and (tCall). We leave the remaining cases for the interested reader.

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

$$e = e_1.f \tag{9}$$
$$t = u \oplus t' \tag{10}$$
$$h, \sigma \vdash e_1 : u\ c \tag{11}$$
$$\mathcal{F}(c, f) = t' \tag{12}$$

From (9) we know the reduction was derived using either (rField) or (rEvalCtxt).

(rField): we know:

$$e_1 = a \tag{13}$$
$$h' = h \tag{14}$$
$$e' = \mathbf{fields}(h, a)(f) = v \tag{15}$$

From $\vdash h$, (13), (11), (12) and (15) we deduce

$$h, (a, \_) \vdash v : t' \tag{16}$$

By (13), (11), (16) and Lemma 5.5 we obtain

$$h, \sigma \vdash v : u \oplus t'$$

The resultant heap h' is trivially well-formed from the assumption $\vdash h$ and (14).

(rEvalCtxt): we know:

$$e' = e_1'.f \tag{17}$$
$$\sigma \vdash e_1, h \rightsquigarrow e_1', h' \tag{18}$$

By $\vdash h$, (11), (18) and the inductive hypothesis we know

$$h, \sigma \vdash e_1' : u\ c \tag{19}$$
$$\vdash h' \tag{20}$$

By (19), (12), (10), (17) and (tField) we derive our first required conclusion

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

and (20) gives us the second required conclusion.

We here consider the former case, (rField), and leave the (easier) latter case for the interested reader. From (rField)

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

$$e = e_1.f := e_2 \tag{21}$$

$$h, \sigma \vdash e_1 : u\ c \tag{22}$$

$$h, \sigma \vdash e_2 : t \tag{23}$$

$$\mathcal{F}(c, f) = u \ominus t \tag{24}$$

From the structure of e, (21), 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:

$$e_1 = a \tag{25}$$

$$e_2 = e' = v \tag{26}$$

$$h' = h[(a, f) \mapsto v] \tag{27}$$

Using (25), (22), (26), (23) and Lemma 5.5(ii) we obtain

$$h, (a, \_) \vdash v : u \ominus t \tag{28}$$

By the assumption $\vdash h$, (25), (22), (24), (28) and Lemma B.4 we get

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

Also, by (26) and (23) we obtain

$$h, \sigma \vdash e' : t \tag{29}$$

and by (29) and Lemma B.6 we get

$$h[(a, f) \mapsto v], \sigma \vdash e' : t$$

as required.

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

$$e = e_1.m(e_2) \tag{30}$$

$$h, \sigma \vdash e_1 : u_1\ c_1 \tag{31}$$

$$h, \sigma \vdash e_2 : t_1^x \tag{32}$$

$$\mathcal{M}(c_1, m) = p : t_1^r\ (u_1 \ominus t_1^x) \tag{33}$$

$$t = u_1 \oplus t_1^r \tag{34}$$

From the structure of e derived from (30) we know that

$$\sigma \vdash e, h \rightsquigarrow e', h'$$

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

$$e_1 = a \text{ and } e_2 = v \tag{35}$$

$$h' = h \tag{36}$$

$$e' = \mathsf{frame}\ \sigma'\ e_b \tag{37}$$

$$\sigma' = (a, v) \tag{38}$$

$$c_a = \mathbf{class}(h, a) \tag{39}$$

$$e_b = \mathcal{MBody}(c_a, m) \tag{40}$$

From (36) and the assumption $\vdash h$ we know that the resulting heap is well-formed. Thus from (37), (34) we only need to show that

$$h, \sigma \vdash \mathsf{frame}\ \sigma\ 'e_b : u_1 \oplus t_1^r \tag{41}$$

The rest of the proof is dedicated to solving this.

From (31), (35), (39) and Lemma B.5 we derive

$$c_a \leq c_1 \tag{42}$$

From (42), (33), the premises of the definition of (WFClass) and the assumption of well-formed programs, giving $\vdash c_a$, we derive

$$\mathcal{M}(c_a, m) = p : t^r\ (t^x) \tag{43}$$
$$t^r \leq t_1^r \tag{44}$$
$$u_1 \ominus t_1^x \leq t^x \tag{45}$$

Also, by (40), (43), $\vdash c_a$ and the premises of (WFClass) we derive

$$\Gamma \vdash e_b : t_r \tag{46}$$
$$\text{where } \Gamma = (\mathsf{self}\ c_a,\ t^x) \tag{47}$$

Using (38), (39) and (tAddr) we derive

$$h, \sigma' \vdash a : \mathsf{self}\ c_a \tag{48}$$

Also, from (35), (38), (31), (32) and Lemma 5.5(ii) we get

$$h, \sigma' \vdash v : u_1 \ominus t_1^x \tag{49}$$

and by (49), (45) and (tSub) we derive

$$h, \sigma' \vdash v : t^x \tag{50}$$

By (38), (46), (47), (48), (50) and the Substitution Lemma 5.9 we derive

$$h, \sigma' \vdash e_b : t^r \tag{51}$$

and by (31), (35), (51) and (tFrame) we get

$$h, \sigma \vdash \mathsf{frame}\ \sigma'\ e_b : u_1 \oplus t^r \tag{52}$$

From (44) and Lemma 5.1 we derive

$$u_1 \oplus t^r \leq u_1 \oplus t_1^r \tag{53}$$

and thus by (52), (53) and (tSub) we get

$$h, \sigma \vdash \mathsf{frame}\ \sigma'\ e_b : u_1 \oplus t_1^r$$

as required by (41).

□

C Definitions and Intermediary Results for Encapsulation

Definition C.1 (Encapsulation Predicates and Notation).

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

$$h(a.f_1 \dots f_n) \overset{\textit{def}}{=} h(\dots h(a.f_1) \dots f_n)$$

$$\mathcal{F}_c(f) \overset{\textit{def}}{=} \mathcal{F}(c,f) = t \ \wedge\ (\forall c' \geq c.\ \nexists t'.\mathcal{F}(c',f) = t')$$

$$\boldsymbol{rFlds}(c,\ f_1 \dots f_n) \overset{\textit{def}}{=} \begin{cases} \mathcal{DF}(c,f_1) \wedge \mathcal{F}(c,f_1) = \mathit{rep}\ c_1 \\ \wedge\ \boldsymbol{rpFlds}(c_1,\ f_2 \dots f_n) \end{cases}$$

$$\boldsymbol{rpFlds}(c,\ f_1 \dots f_n) \overset{\textit{def}}{=} \begin{cases} (\mathcal{F}(c,f_1) = \mathit{rep}\ c_1 \ \vee\ \mathcal{F}(c,f) = \mathit{peer}\ c_1) \\ \wedge\ \boldsymbol{rpFlds}(c_1,\ f_2 \dots f_n) \end{cases}$$

$$\boldsymbol{pure}(c,\ m) \overset{\textit{def}}{=} \mathcal{M}(c,m) = \mathit{pure} : \_\ (\_)$$

Lemma C.2 (Substitution for Encapsulation).

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

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

Proof. Using substitution lemma 5.9. □

Lemma C.3 (Purity implies Encapsulation).

1. $\Gamma \vdash_{\boldsymbol{pure}} e \implies \Gamma \vdash_{\boldsymbol{enc}} e$
2. $h,\sigma \vdash_{\boldsymbol{pure}} e \implies h,\sigma \vdash_{\boldsymbol{enc}} e$

Proof. Immediate from the definitions of $\vdash_{\mathbf{pure}}$ and $\vdash_{\mathbf{enc}}$. □

Lemma C.4 (Properties of Transitive Ownership).

1. $a \in \boldsymbol{owner}^*(h,a)$
2. $b \in \boldsymbol{owner}^*(h,a) \implies \boldsymbol{owner}(h,b) \in \boldsymbol{owner}^*(h,a)$

Proof. Immediate from Definition 5.7. □

Lemma C.5 (Subject Reduction for Encapsulation and Purity).
Assuming $\vdash_{\boldsymbol{enc}} P$*:*

1. $\sigma \vdash e,h \rightsquigarrow e',h'$ *and* $h,\sigma \vdash_{\boldsymbol{pure}} e \implies h',\sigma \vdash_{\boldsymbol{pure}} e'$

2. $\sigma \vdash e, h \rightsquigarrow e', h'$ *and* $h, \sigma \vdash_{\textbf{\textit{enc}}} e \implies h', \sigma \vdash_{\textbf{\textit{enc}}} e'$

Lemma C.6 (Encapsulation and Purity Safety)**.**
Assuming $\vdash_{\textbf{\textit{enc}}} P$*:*

$$1. \quad \left.\begin{array}{l} \sigma \vdash e, h \rightsquigarrow e', h' \\ a \in \textbf{\textit{dom}}(h) \\ h, \sigma \vdash_{\textbf{\textit{pure}}} e \end{array}\right\} \implies h(a.f) = h'(a.f)$$

$$2. \quad \left.\begin{array}{l} \sigma \vdash e, h \rightsquigarrow e', h' \\ a \in \textbf{\textit{dom}}(h) \\ h, \sigma \vdash_{\textbf{\textit{enc}}} e \\ \textbf{\textit{owner}}(h, \sigma(\textsf{\textit{this}})) \notin \textbf{\textit{owner}}^*(h, a) \end{array}\right\} \implies h(a.f) = h'(a.f)$$

Proof. (For both Theorem C.5 and C.6).
The proof proceeds by induction on the derivation of

$$\sigma \vdash e, h \rightsquigarrow e', h' \tag{54}$$

assuming $h, \sigma \vdash_{\textbf{pure}} e$ in case (1) and $h, \sigma \vdash_{\textbf{enc}} e$ in case (2). We here consider the interesting cases and leave the simpler cases for the interested reader. Case the last rule used in deriving (54) is

(rAssign): Then we know

$$e = b.f := v \tag{55}$$
$$e' = v \tag{56}$$
$$h' = h[(b, f) \mapsto v] \tag{57}$$

From (55) we know $h, \sigma \not\vdash_{\textbf{pure}} e$. So we do not need to consider case (1). For case (2) however, we also know

$$h, \sigma \vdash_{\textbf{enc}} b.f := v \tag{58}$$
$$\textbf{owner}(h, \sigma(\textsf{this})) \notin \textbf{owner}^*(h, a) \text{ for } a \in \textbf{dom}(h) \tag{59}$$

Subject reduction for encapsulation is easy to show because from (56), Definition 6.5 and an application of the topological subject reduction Lemma 5.10 we know

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

The more involving part is showing encapsulation safety, that is $h(a.f) = h'(a.f)$. From (58) and Definition 6.5 we know

$$h, \sigma \vdash b : z$$

where z has three sub-cases.

$z =$ **self:** Then we know

$$b = \sigma(\textsf{this}) \tag{60}$$

Substituting (60) in (59) we get

$$\textbf{owner}(h, b) \notin \textbf{owner}^*(h, a) \tag{61}$$

By applying LemmaC.4.2 we obtain $b \notin \textbf{owner}^*(h, a)$ and subsequently we apply Lemma C.4.1 to derive

$$b \neq a \tag{62}$$

Hence by (57) and (62) we can safely conclude $h(a.f) = h'(a.f)$

$z = $ **peer:** Then we know

$$\mathbf{owner}(h, b) = \mathbf{owner}(h, \sigma(\mathsf{this})) \tag{63}$$

Substituting (63) in (59) we get

$$\mathbf{owner}(h, b) \notin \mathbf{owner}^*(h, a) \tag{64}$$

By applying LemmaC.4.2 and then Lemma C.4.1 we derive $b \neq a$ Hence by (57) we can conclude $h(a.f) = h'(a.f)$

$z = $ **rep:** Then we know

$$\sigma(\mathsf{this}) = \mathbf{owner}(h, b) \tag{65}$$

and Substituting (65) in (59) we get

$$\mathbf{owner}(h, \mathbf{owner}(h, b)) \notin \mathbf{owner}^*(h, a) \tag{66}$$

By Applying LemmaC.4.2 twice and then Lemma C.4.1 we derive $b \neq a$ Hence by (57) we can conclude $h(a.f) = h'(a.f)$

(rCall): Then we know

$$e = b.m(v) \tag{67}$$
$$e' = \mathsf{frame}\ (b, v)\ e_b \tag{68}$$
$$e_b = \mathcal{MBody}(\mathbf{class}(h, b), m) \tag{69}$$
$$h' = h \tag{70}$$

We now have two cases to consider:

$(h, \sigma \vdash_{\mathbf{pure}} e)$**:** From Definition 6.5 we know

$$h, \sigma \vdash b.m(v) : t \tag{71}$$
$$h, \sigma \vdash b : c \tag{72}$$
$$\mathbf{pure}(c,\ m) \tag{73}$$

From the assumption $\vdash_{\mathbf{enc}} P$ we know $\vdash_{\mathbf{enc}} c$ and thus by (73), (69) and Definition 6.3 we know

$$(\mathsf{self}\ c, \_) \vdash_{\mathbf{pure}} e_b \tag{74}$$

From (72) we know b and v respect the environment $(\mathsf{self}\ c, \_)$ that is

$$h, (b, v) \vdash \mathsf{this} : \mathsf{self}\ c \tag{75}$$
$$h, (b, v) \vdash \mathsf{x} : \_ \tag{76}$$
$$\tag{77}$$

and as a result of Lemma C.2 (74), (75) and (76) we get:

$$h, (b, v) \vdash_{\mathbf{pure}} e_b \tag{78}$$

By (71), (54), (68) and the Subject Reduction Lemma 5.10 we get

$$h, \sigma \vdash \mathsf{frame}\ (b, v)\ e_b : t \tag{79}$$

and hence by (78), (79), (68) and Definition 6.5 we obtain

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

as required by Lemma C.5. Also, by (70) we trivially conclude

$$h(a.f) = h'(a.f)$$

as required by Lemma C.6.

($h, \sigma \vdash_{\mathbf{enc}} e$): From Definition 6.5 we know we have two sub-cases. The first sub-case states that the method called is pure and the proof progresses as the previous case when $h, \sigma \vdash_{\mathbf{pure}} e$. The sub-second case states that we have $h, \sigma \vdash_{\mathbf{enc}} e$ because

$$h, \sigma \vdash b : z\ c \tag{80}$$

$$h, \sigma \vdash b.m(v) : t \tag{81}$$

If the method m is impure in c, then by the assumption $\vdash_{\mathbf{enc}} P$, thus $\vdash_{\mathbf{enc}} c$ and (69) we know

$$(\mathsf{self}\ c, \_) \vdash_{\mathbf{enc}} e_b \tag{82}$$

By (80) we know b and v correspond to $(\mathsf{self}\ c, \_)$ that is

$$h, (b, v) \vdash \mathsf{this} : \mathsf{self}\ c \tag{83}$$

$$h, (b, v) \vdash \mathsf{x} : \_ \tag{84}$$

and thus by (83), (84), (82) and Lemma C.2 we get

$$h, (b, v) \vdash_{\mathbf{enc}} e_b \tag{85}$$

From (80) we derive

$$h \vdash (b, v) \preceq_{\mathbf{enc}} \sigma \tag{86}$$

Also, by (81), (54), (68) and the topological subject reduction we know

$$h, \sigma \vdash \mathsf{frame}\ (\ b, v) e_b : t \tag{87}$$

Thus by (87), (86), (85), (68) and Definition 6.5 we conclude

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

as required by Lemma C.5. Also, by (70) we trivially conclude

$$h(a.f) = h'(a.f)$$

as required by Lemma C.6.

(rFrame2): From the rule we know

$$e = \mathsf{frame}\ \sigma\ ' v \tag{88}$$

$$e' = v \tag{89}$$

$$h' = h \tag{90}$$

From (90) we trivially conclude

$$h(a.f) = h'(a.f)$$

as required by Lemma C.6. For Lemma C.5 we have two cases two consider:

$(h,\sigma \vdash_{\textbf{enc}} e)$: From Definition 6.5 we know that either $h,\sigma' \vdash_{\textbf{pure}} v$ or

$$h,\sigma \vdash \mathsf{frame}\ \sigma\ 'v : t \tag{91}$$
$$h \vdash \sigma' \preceq_{\textbf{enc}} \sigma \tag{92}$$
$$h,\sigma \vdash_{\textbf{enc}} v \tag{93}$$

By (91), (54), topological subject reduction Lemma 5.10 and (89) we get

$$h,\sigma \vdash v : t \tag{94}$$

and by (94) and Definition 6.5 we obtain

$$h,\sigma \vdash_{\textbf{enc}} v$$

as required by Lemma C.5.

$(h,\sigma \vdash_{\textbf{pure}} e)$: Similar to the above case.

□

Corollary C.7. *$h,\sigma \vdash_{enc} e$ and $\sigma \vdash e,h \rightsquigarrow^* e',h'$ implies $h',\sigma \vdash_{enc} e'$*

Proof. By induction on the length of $\rightsquigarrow^*$ using Theorem C.5. □

Theorem 6.6 (Encapsulation)

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

Proof. By induction on the length of $\rightsquigarrow^*$.

Case $n = 0$: Immediate.

Case $n = k+1$: We have $\sigma \vdash e,h \rightsquigarrow^k e_k,h_k \rightsquigarrow e',h'$. By inductive hypothesis ($n = k$) we know that

$$a \in \mathbf{dom}(h) \quad \text{and} \quad \begin{array}{l}\mathbf{owner}(h,\sigma(\mathsf{this})) \notin \mathbf{owner}^*(h,a) \\ \text{implies } h(a.f) = h_k(a.f)\end{array} \tag{95}$$

Also by $h,\sigma \vdash_{\textbf{enc}} e$ and Corollary C.7 we know

$$h_k,\sigma \vdash_{\textbf{enc}} e_k \tag{96}$$

By (96) and Theorem C.6.2 we obtain

$$a \in \mathbf{dom}(h_k) \quad \text{and} \quad \begin{array}{l}\mathbf{owner}(h_k,\sigma(\mathsf{this})) \notin \mathbf{owner}^*(h_k,a) \\ \text{implies } h_k(a.f) = h'(a.f)\end{array} \tag{97}$$

Finally by (95), (97), Lemma B.1 and Lemma B.2 we obtain

$$a \in \mathbf{dom}(h) \quad \text{and} \quad \begin{array}{l}\mathbf{owner}(h,\sigma(\mathsf{this})) \notin \mathbf{owner}^*(h,a) \\ \text{implies } h(a.f) = h'(a.f)\end{array}$$

□