Universes: A Type System for Controlling Representation Exposure

Peter Müller and Arnd Poetzsch-Heffter
Fernuniversität Hagen, 58084 Hagen, Germany
[Peter.Mueller, Arnd.Poetzsch-Heffter]@Fernuni-Hagen.de

November 23, 1999

Abstract

We present a type system that allows to express a hierarchical partitioning of the object store into so-called *universes*. Type checking enforces referencing constraints between objects in different universes. The universe type system provides support for preventing rep exposure while retaining a flexible sharing model. It is easy to apply and guarantees an invariant that is strong enough for modular verification. Our type system is related to ownership types ([CPN98]), balloon types ([Alm97]), and islands ([Hog91]). However, it is capable of specifying certain implementation patterns (e.g., binary methods, several objects using a common representation) that cannot be handled by the other approaches.

1 Introduction

Sharing mutable objects is typical for object-oriented programs. As a direct consequence of the concept of object identities, it is one of the fundamentals of the OO-programming model. Furthermore, OO-programs gain much of their efficiency through sharing and destructive updates.

However, uncontrolled sharing leads to serious problems: Usually several objects work together to represent larger components such as windows, parsers, dictionaries, etc. Current OO-languages do not prevent references to objects of such components from leaking outside the components' boundaries, a phenomenon called *rep exposure*. Thus, arbitrary objects can use these references to manipulate the internal state of components without using their explicit interface. These manipulations can effect both the abstract value of components (in the sense of [Hoa72]) and their invariants. This makes OO-programs very hard to reason about. Furthermore, in systems with uncontrolled sharing, basically every object can interact with any other object. Therefore, such systems lack a modular structure and are difficult to maintain.

In this extended abstract, we present a type system that enforces a hierarchical partitioning of the object store into so-called *universes* and controls references between universes. The universe type system provides support for preventing rep exposure while retaining a flexible sharing model. It is easy to apply and guarantees an invariant that is strong enough for modular verification. Our type system is related to ownership types ([CPN98]), balloon types ([Alm97]), and islands ([Hog91]). However, it is capable of specifying certain implementation patterns (e.g., binary methods, several objects using a common representation) which cannot be handled by the other approaches.

Overview. Section 2 describes the universe programming model and discusses related work. In Section 3, we formalize the universe type system for a Java subset and the invariant it guarantees. Furthermore, we sketch the proof of type safety. Our conclusions are contained in Section 4.

2 Alias Control with Universes

The universe type system allows one to partition the object store into several universes. References between objects of different universes are restricted in a way that prevents rep exposure. In this section, we describe the universe programming model. We informally sketch the universe type system and demonstrate its application with an example. Furthermore, we compare universes to other approaches to alias control.

2.1 Structuring the Object Store

OO-languages in general allow for arbitrary references between objects. The universe type system enables the programmer to structure the object store according to a component-oriented programming model and provides support for sharing-control between components. It is a proper refinement of usual type systems; i.e. the programmer can use the additional power of the type system, but is not forced to so.

The Universe Programming Model. Systems usually comprise several components. Components consist of one or more objects. Some of these objects are used to interact with other components. Their interfaces form the interface of the component. The other objects are the internal *representation* of the component. A component's representation should be modified only through the component's interface to control modification of the component's abstract value ([Hoa72, MPH99]) and to guarantee data consistency. Therefore, references to objects of a component's representation must not be passed to other components (*rep exposure*), i.e., references to representation objects must be kept inside the component.

Representations and Universes. We associate every component with a partition of the object store that contains the component's representation, a so-called *universe*. Since a component's representation may contain other components which are in turn associated with a universe, universes form a hierarchical structure. A designated *root universe* corresponds to the whole object store and encloses all other universes. Two universes either enclose each other or are disjoint. The hierarchy of universes introduces a partial order of universes with the root universe as greatest element. We use the term *an object X belongs to universe U* if U is the least universe containing X.

The objects at the interface of a component are not part of the representation (and therefore not contained in the universe). We call them the *owner objects* of the corresponding universe. Owner objects of universe U belong to the universe directly enclosing U.

Consider a component for a doubly linked list of objects with iterators. The list header and the iterators are non-representation objects of the component. They are the owners of the component's universe which contains the nodes of the list.

Sharing Control. An owner object may reference objects belonging to its universe. All other references across universe boundaries are basically prohibited for the following reasons:

- Objects outside a universe must not reference objects inside. Otherwise, they could use these references to manipulate the internal state of the component.[1]
- Objects inside a universe must not reference to objects outside. If the abstract value of the component depended on the state of objects outside its representation, it could be modified without using the component's interface.

These rules guarantee that objects belonging to universe U can only be referenced by objects belonging to U and U's owner objects. However, the above rules are too strong in two situations:

[1]In this context, local variables and formal parameters behave like instance variables of the `this` object. I.e., universes control both static and dynamic aliasing.

(1) Components might want to pass parts of their representations to other components, provided that these components do not use the references for modifications. Such situations occur e.g., when a component needs to store a representation object in a container or when two components have to be tested for structural equality. (2) Objects inside a universe could contain references to objects outside if their abstract values did not depend on the states of the objects outside. To support both situations, we introduce so-called *read only references*.

Read only References. Read only references cannot be used to perform field updates or method invocations on the referenced object[2]. Reading fields via read only references in turn yields read only references (or values of primitive types). Abstract values of components must not depend on states of objects referenced read only.

Read only references can be used to pass references across universe boundaries. A read only reference to an object belonging to universe U can be turned into a normal reference by objects of U and U's owner objects. E.g., object X can pass a reference to object Y as read only reference to a container. When this reference is retrieved later, X can cast it back to a normal reference and use it for method invocations, etc.

Fig. 1 shows the object structure of a doubly linked list of objects with two iterators. (Objects are depicted by boxes; solid and dashed arrows depict normal and read only references, resp.; the universe is drawn as ellipse.) The nodes are the representation of the component and therefore inside the universe. Other components can interact with the list header and the iterators, which are the owner objects of the universe. The objects stored in the list are referenced read only. Subsection 2.3 sketches the implementation of the list/iterator example.

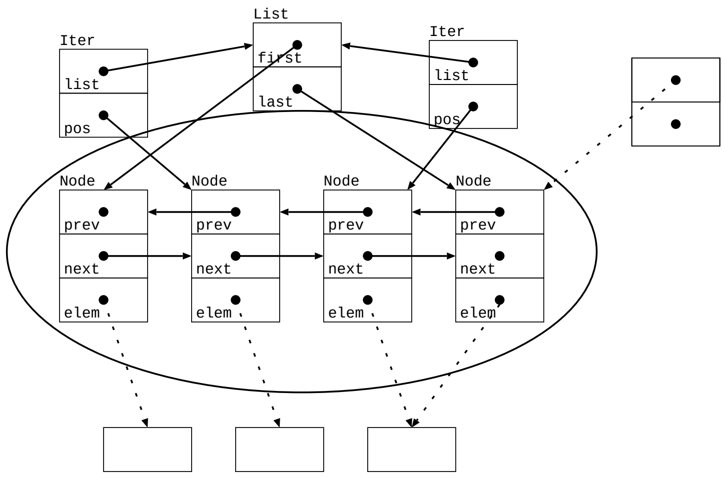

Figure 1: Object Structure for List/Iterator Example

2.2 Static Checking of Representation Containment

In the last subsection, we sketched an ideal scenario for alias control for components. However, to check reference containment statically, we have to use a slightly weaker programming model. In this subsection, we present the refined programming model and informally describe a type system to enforce it.

[2]To keep things simple, we do not consider read only methods here (i.e. methods without side-effects). For practical applications, it would be helpful.

2.2.1 Component Programming Model and Universes

We simplify the component programming model as follows: (1) We associate every object with its own *object universe*. I.e., each object X is regarded as the interface of a component with a possibly empty representation. An object is the only owner object of its object universe. (2) We associate every type with a *type universe*. If T is a type declared in module M then every object of a type declared in M is an owner object of T's type universe. Due to inheritance, objects of subtypes of types declared in M may also contain references to objects in T's type universe. However, access control guarantees that subtype methods cannot manipulate objects via such references (see below for details). Type universes allow objects of types declared in the same module to access a common representation. Thus, components with several owner objects can be realized by implementing them in one module.

Whereas the first simplification does not really affect the programming model (additional universes can't hurt), the use of type universes reduces the amount of sharing control that can be done. E.g., type universes do not provide support for keeping the nodes of two lists disjoint if the lists' representations are stored in the same type universe. However, objects in T's type universe can only be manipulated by methods implemented in T's module. Therefore, type universes provide sufficient sharing control for modular reasoning, since all "dangerous" code is located in one module (cf. [MPH99] for a discussion).

2.2.2 The Universe Type System

Reference containment for universes is statically checked by the universe type system. In this subsection, we present the basic ideas of a universe type system for the Java subset described in Subsection. 3.1 and apply it to an example. A formalization of the type system is given in Section 3.

Universes and Types. There are three kinds of universes: The root universe, type universes, and object universes. Each class C introduces one type for read only references (*read only type*) and one type for every universe in a program execution (*reference types*); C is called the *base class* of these types. All types having the same base class share a common implementation, but are regarded as different types.

The subtype relation follows the subclass relation in Java. Two reference types are subtypes if they belong to the same universe and their base classes are subclasses. Two read only types are subtypes if their base classes are subclasses. Each reference type with base class C is a subtype of the read only type for C.

Since objects of a class in different universes have different types, objects of one universe cannot be assigned to variables expecting objects of another. All reference types are subtypes of the corresponding read only type. Therefore, variables of read only types can hold objects of any universe.

Type Schemes. A class introduces one reference type for each universe (in particular, for each object universe). Thus, the set of types is not fixed at compile time. To enable static type checking, we use so-called *type schemes* to statically type variables, methods, expressions, etc.

Since the universe of a type T is not known at compile time, the implementation of the base class of T can refer to other reference types only relatively to the universe T belongs to. To support the programming model described in Subsection 2.1, the universe type system provides three kinds of type schemes for reference types: (1) *Ground type schemes* of the form `C` to refer to the type for class `C` belonging to the same universe as T, (2) *object type schemes* of the form `C<obj>` to refer to the type for class `C` in the object universe owned by `this`, and (3) *class type schemes* of the form `C<S>` to refer to the type for class `C` in the type universe associated with the type for class `S` in the universe T belongs to. Furthermore, there are type schemes for read only types (`C<ro>`), and primitive types (`int`, `boolean`, the `null` type). The subtype relation on type schemes resembles the subtype relation on types.

The Universe Invariant. In every well-typed state, each instance variable and each local variable/formal parameter holds a value of a subtype of the declared type of the variable. Thus, if object X references object Y exactly one of the following cases holds:[3] (1) X and Y belong to the same universe; (2) Y belongs to the object universe owned by X; (3) Y belongs to a type universe owned by X; (4) the reference is read only.

2.3 Example

In this subsection, we present two implementations of a doubly linked list. We illustrate the application of object and type universes, and of read only types. Our examples contain two patterns that cannot be handled in other type systems for alias control: Binary methods and cooperating objects that access a common representation.

Doubly Linked Lists. Our doubly linked list component consists of a class `Node` for the node structure and a class `List` for the head of the list. Since the list is supposed to contain objects of any universe, `Node`'s `elem` field is declared read only. Each node structure exclusively belongs to one list header. Therefore, the nodes are stored in the object universe of the list header (`first` and `last` use the object type scheme). The `equals` method in `List` takes a read only parameter. Thus, it can access its representation and compare it to the representation of `this` (we assume `Object` to contain a method `boolean equals(Object<ro>)`).

```
class Node  { Object<ro> elem; Node prev; Node next; }

class List {
  Node<obj> first; Node<obj> last;
  public List() {
    Node<obj> f = new Node<obj>();    Node<obj> l = new Node<obj>();
    first = f;                        last = l;
    f.next = l;                       l.prev = f;                 }
  public void appFront(Object<ro> o) { ... }
  public boolean equals(List<ro> l)  {
    Node   n1 = first;                Node<ro>   n2 = l.first;
    Node   l1 = last;                 Node<ro>   l2 = l.last;
    Object o1 = n1.elem;              Object<ro> o2 = n2.elem;
    boolean equ = o1.equals(o2);
    while (n1 != l1 && n2 != l2 && equ) {
      n1 = n1.next;                   n2 = n2.next;
      o1 = n1.elem;                   o2 = n2.elem;
      equ = o1.equals(o2);                      }
    return n1 == l1 && n2 == l2;                                  }
}
```

Lists with Iterators. We are now going to enhance our list component by iterators. The iterators allow one to remove elements from the list. Therefore, they must be able to modify the list representation and cannot be implemented via read only references. To allow lists and iterators to access a common representation, we use type universes instead of object universes to store the node structure of the list. To do that, every `Node<obj>` in the above program has to be replaced by `Node<List>`. The same type scheme is used by the implementation of `Iter`:

```
class Iter {
  List list;  Node<List> position;     public Iter(List l)      { ... }
  public boolean hasNext() { ... }     public Object<ro> next() { ... }
  public void remove()     { ... }                                  }
```

[3]Again, local variables/formal parameters behave like instance variables of `this`.

2.4 Related Work

Universes have been designed w.r.t. the following objectives: They should (1) have simple semantics, (2) be easy to apply, (3) be statically checkable (4) guarantee an invariant that is strong enough for modular reasoning, and (5) be flexible enough for many useful programming patterns. In particular, they should provide support for some of the implementation patterns that cannot be handled by related approaches (e.g., binary methods, several objects sharing one representation). In this subsection, we compare the universe type system to other approaches to alias control w.r.t. these objectives.

Ownership types ([CPN98]) provide a very flexible means for alias control. Each object is associated with a context (similar to an object universe) that contains the object's representation. Context parameters allow objects in one context to hold references to objects in another, which is useful for many implementation patterns. However, context parameters make ownership types rather difficult to apply (cf. [Bok99]). Read only types can replace context parameters in most situations and lead to programs that are easier to read and reason about. Since contexts are associated with objects, they cannot be accessed by several objects. Thus, ownership types cannot handle binary methods and objects sharing a common representation. As presented in [CPN98], ownership types do not support subtyping and inheritance.

[NVP98] proposes alias modes to control aliasing. Similar to ownership types, each object is equipped with a context. Alias modes specify constraints on references. E.g., the mode `rep` enforces representation containment (like in object universes). The mode `arg` provides references that can be freely passed around, but must not be used to manipulate the referenced object. Thus, they are similar to read only references. The so-called roles for `arg` references are similar to context parameters. Like ownership types, alias modes have been presented for a language without subtyping and inheritance.

Balloon types ([Alm97]) aim at full representation encapsulation, i.e., all objects reachable from an object are contained in its balloon (as if every reachable object would be inside an object universe). This is too restrictive for many programs (e.g., singly linked lists). Balloon types require a rather complex checking algorithm based on abstract interpretation and cannot be checked modularly.

Like balloon types, Islands ([Hog91]) also provide only full encapsulation and suffer therefore from the same lack of expressiveness. Islands are based on a destructive read operation, which has a rather unintuitive semantics. They allow for dynamic aliases but restrict them to be read only. Islands have not been formally validated.

Confined types ([BV99]) guarantee that objects of a confined type cannot be referenced in or accessed by code declared outside the confining package. Thus, confined types are similar to type universes in that they enable aliasing control on the module level. Confined types have been designed for the development of secure systems. They do not support representation containment on the object level, which makes verification of components with just one interface object difficult.

3 The Universe Type System and its Properties

In this section, we present the universe type system in more detail: We describe our programming language and give formal definitions for types and type schemes. Based on formal type rules of our programming language, we sketch the proof of type safety. From the type safety lemma we derive that representation containment is an invariant in well-typed programs.

3.1 Formalization of the Universe Type System

Programming Language. Our programming language is a Java subset with the following features: A program consists of a set of modules. Each module contains a set of classes which may contain declarations of instance variables and instance methods. The language provides local variables and statements for reading and writing instance variables, simple assignments (with casts), object creation, method invocations, sequential statement composition, conditional and loop statement.[4] The expressions of our Java subset are literals (integer, boolean, `null`), local variables/formal parameters, the `this` reference, and the usual unary and binary operations.

Type schemes as described in Subsection 2.2.2 are used in wherever types occur in conventional Java programs (declaration of variables and method signatures, casts). In addition to the context conditions of Java, we impose the following restrictions: (1) The null type scheme (see below) must not occur in programs. (2) A class type schemes `C<T>` may only be used in classes declared in `T`'s module. The latter condition is necessary to restricts access to type universes (see Subsection 2.2.1).

We omit the abstract syntax and the operational semantics of our language for brevity. For a very similar Java subset, they can be found in [PHM99].

Type Schemes. Type schemes as described in Subsection 2.2.2 are formalized by the following data type where sort *ClassId* denotes the class identifiers as given in a program. An axiomatization of the subtype relation on type schemes can be found in App. A.

$$
\begin{array}{lcl}
\textbf{data type} & & \\
\quad \mathit{TypeScheme} & = & \mathit{grndS(ClassId)} \\
 & | & \mathit{objS(ClassId)} \\
 & | & \mathit{typeS(ClassId\ ClassId)} \\
 & | & \mathit{roS(ClassId)} \\
 & | & \mathit{boolS} \mid \mathit{intS} \mid \mathit{nullS}
\end{array}
$$

Type Rules. The type scheme of an expression or field e is denoted by $[e]$. The type schemes of method invocations and field accesses depend on both the type schemes of the target variables and the type schemes of the methods (return and parameter type schemes) and fields. E.g., if $[v]$ is a read only type scheme and $[f]$ is a reference type scheme, the type scheme of $v.f$ is a read only type scheme. These combinations of type schemes are described by a partial operation

$$* : \mathit{TypeScheme} \times \mathit{TypeScheme} \rightarrow \mathit{TypeScheme}$$

which is defined by the following table (first argument: rows, second argument: columns; all combinations not mentioned in the table yield *undef*):

	grndS(C)	objS(C)	typeS(C,T)	roS(C)	boolS	intS	nullS
grndS(D)	grndS(C)	objS(C)	typeS(C,T)	roS(C)	boolS	intS	nullS
objS(D)	objS(C)	*undef*	*undef*	roS(C)	boolS	intS	nullS
typeS(D,S)	typeS(C,S)	*undef*	*undef*	roS(C)	boolS	intS	nullS
roS(D)	roS(C)	roS(C)	roS(C)	roS(C)	boolS	intS	nullS

The universe-specific type rules are displayed below. All other rules for our Java subset are straightforward and therefore omitted. In the type rules, the judgment $\mathcal{P} \vdash$ `stmt` expresses that statement `stmt` is well-typed in program $\mathcal{P}$.

Three aspects that are important for type-safety of the universe type system are contained in the rules below: (1) The type scheme combinator $*$ is used to determine the type schemes for fields accesses and method invocations. The resulting type scheme must not be *undef* to guarantee that an expression does not evaluate to a (non-read only) reference that points "two steps down" in the universe hierarchy (e.g., by reading an *objS* field on an *objS* variable). (2) To keep object universes on the same level of the universe hierarchy disjoint (except for read only references), all local variables/formal parameters of object type schemes refer to the object universe of `this`. To check this property statically, fields of object type schemes and methods with object type schemes as result/parameter type schemes may only be accessed/invoked on `this`. (3) Neither writing field access nor method invocation is allowed on read only references.

[4]A clone operation that performs a deep copy of an object structure and moves the result to another universe is convenient to exchange data across universe boundaries (cf. [MPH99]). However, we omit this operation here for a lack of space.

$$\frac{\mathrm{TS} \preceq_S [\mathrm{v}], \mathrm{TS} \preceq_S [\mathrm{e}],}{\mathcal{P} \vdash \texttt{v=(TS)e;}} \qquad \frac{\mathrm{TS} \preceq_S [\mathrm{v}], \mathrm{TS}\ \textit{ is grndS, objS, or typeS}}{\mathcal{P} \vdash \texttt{v=new TS();}}$$

$$\frac{[\mathrm{f}]\ \textit{is no objS}, [\mathrm{w}] * [\mathrm{f}] \preceq_S [\mathrm{v}]}{\mathcal{P} \vdash \texttt{v=w.f;}} \qquad \frac{[\mathrm{f}] \preceq_S [\mathrm{v}]}{\mathcal{P} \vdash \texttt{v=this.f;}}$$

$$\frac{[\mathrm{v}]\ \textit{is no roS}, [\mathrm{f}]\ \textit{is no objS}, [\mathrm{e}] \preceq_S [\mathrm{v}] * [\mathrm{f}]}{\mathcal{P} \vdash \texttt{v.f=e;}} \qquad \frac{[\mathrm{e}] \preceq_S [\mathrm{f}]}{\mathcal{P} \vdash \texttt{this.f=e;}}$$

$$\frac{\begin{array}{l}[\mathit{par}(\mathrm{m})]\ \textit{is no objS}, [\mathit{res}(\mathrm{m})]\ \textit{is no objS}, \\ [\mathrm{w}]\ \textit{is no roS}, \\ [\mathrm{e}] \preceq_S [\mathrm{w}] * [\mathit{par}(\mathrm{m})], [\mathrm{w}] * [\mathit{res}(\mathrm{m})] \preceq_S [\mathrm{v}]\end{array}}{\mathcal{P} \vdash \texttt{v=w.m(e);}} \qquad \frac{[\mathrm{e}] \preceq_S [\mathit{par}(\mathrm{m})], [\mathit{res}(\mathrm{m})] \preceq_S [\mathrm{v}]}{\mathcal{P} \vdash \texttt{v=this.m(e);}}$$

Since read only type schemes are supertypes of the corresponding reference type schemes, the cast operation can be used to downcast expressions of read only type schemes to reference type schemes. As for ordinary casts, a dynamic check guarantees that the dynamic type of the right-hand-side object is a subtype of the type of the left-hand-side variable and therefore refers to the same universe (see App. A).

3.2 Type Safety and Representation Containment

Executions States. We call a type system *type safe*, if it guarantees that every valid execution state is well-typed, i.e., the types of the values held by local variables/formal parameters and instance variables are subtypes of the declared types. To formalize well-typedness, we use the following definitions of the operational semantics (cf. [MPH99] for details): Objects are described by a data type $\mathit{Object} = \mathit{obj}(\mathit{Universe}, \mathit{ClassId}, \mathit{ObjId})$, where *ObjId* is a sort of object identifiers to distinguish different objects of the same type. A data type *Value* is used to formalize the values of the programming language (references, integers, booleans, `null`). An object store (sort *Store*) maps instance variables to values. The special variable \$ of sort *Store* is used to refer to the current object store. An execution state (*State*) maps local variables/formal parameters to *Value* and \$ to *Store*. Universes and types are formalized below. The function τ yields the type of a value.

data type

$$\begin{array}{lcl} \mathit{Universe} & = & \mathit{rootU(\ )} \\ & | & \mathit{typeU(Universe\ ClassId)} \\ & | & \mathit{objU(Object)} \\ \mathit{Type} & = & \mathit{refT(Universe\ ClassId)} \\ & | & \mathit{roT(ClassId)} \\ & | & \mathit{booleanT()} \\ & | & \mathit{intT()} \\ & | & \mathit{nullT()} \end{array} \qquad \begin{array}{ll} \multicolumn{2}{l}{\tau : \mathit{TypeScheme} \times \mathit{Object} \rightarrow \mathit{Type}} \\ \tau(\mathit{grndS}(C), \mathit{obj}(U, D, I)) & = \mathit{refT}(U, C) \\ \tau(\mathit{typeS}(C, T), \mathit{obj}(U, D, I)) & = \mathit{refT}(\mathit{typeU}(U, T), C) \\ \tau(\mathit{objS}(C), O) & = \mathit{refT}(\mathit{objU}(O), C) \\ \tau(\mathit{roS}(C)), O) & = \mathit{roT}(C) \\ \tau(\mathit{boolS}, O) & = \mathit{booleanT} \\ \tau(\mathit{intS}, O) & = \mathit{intT} \\ \tau(\mathit{nullS}, O) & = \mathit{nullT} \end{array}$$

The types of variables on the stack, the types of instance variables, and the parameter and return types of method incarnations depend on the corresponding type schemes (of the variables, fields, methods) and their universe. This universe is determined by the `this` object for variables, and the target objects for instance variables and method incarnations. The appropriate mapping of type schemes and objects to types is defined by τ.

Type Safety. An execution state S is well-typed if for every local variable/formal parameter v and `this` $\tau(S(\mathrm{v})) \preceq \tau([\mathrm{v}], S(\mathrm{this}))$, and for every valid instance variable x.f $\tau(S(\$)(\mathrm{x.f})) \preceq \tau([\mathrm{f}], S(\mathrm{x}))$ holds. A state S is *well-formed* if it is well-typed and `this` references an object ($S(\mathrm{this}) \neq \mathit{null}$). For simplicity, we assume that program execution starts in an initial state in which some predefined object X is allocated. All instance variables of X are initialized to *null*. X is considered to be the owner of the root universe. Execution starts by invoking a designated method of X. Therefore, the initial state is well-formed.

Lemma: For each program execution that starts in a well-formed state, the terminating state and all intermediate states are well-formed.

The proof of the type safety lemma runs by structural induction over the operational semantics. The central property used in the proof is that in all well-formed states S with $S(\mathrm{w}) \neq \mathit{null}$, the equality

$$\tau([\mathrm{w}] * [\mathrm{f}], S(\mathrm{this})) = \tau([\mathrm{f}], S(w))$$

holds where `w` is a variable and `f` denotes a field, or the parameter or result of a method (assuming that $[\mathrm{w}] * [\mathrm{f}]$ is defined and $[\mathrm{f}]$ is not an *objS*). Basically, this property holds (1) because `this` and `w` belong to the same universe and (2) because of the definition of the type scheme combinator $*$. The proof runs by case distinction on the type schemes of `w` and `f`. For brevity, proofs are omitted here.

Controlling Representation Exposure. The universe invariant given in Section 2.2.2 is an almost immediate consequence of the type safety lemma. It expresses the following *representation containment property*: All access paths from the root universe to a representation object X that do not contain read only references pass through owners of X's universe. It remains to be shown that an object belonging to a representation, i.e. a universe U, can only be modified by invoking a method on an owner of U. In particular, it is not possible to bypass owners via read only references. The following lemma states this fact more precisely:

Lemma: 1) An object $\mathit{obj}(U, C, Id)$ can only be modified by method incarnations in which $S(\mathrm{this})$ belongs to U or is an owner of U.

2) Method incarnations with $S(\mathrm{this}) = \mathit{obj}(U, C, Id)$ can only be invoked by method incarnations in which $S(\mathrm{this})$ belongs to U or is an owner of U.

The second part of the lemma states in particular that the call tree of methods corresponds to the universe hierarchy: An invocation either stays in the same universe or goes into a directly contained universe. The proof of the lemma uses context condition (2) given in Subsection 3.1 and the fact that type schemes of target objects must not be read only.

4 Conclusion

We presented a flexible model for object-oriented programming that supports a hierarchical structure of the object store. It is a proper extension of the classical model in which all objects belong to one universe. It supports read only references to express restricted access to objects. Read only references increase the flexibility of the programming model and simplify the implementation of methods that need access to two representations. The programming model is realized by a type system that enforces a special representation containment property.

The representation containment property guarantees that modification of a representation is only possible by calling a method on a corresponding owner object. It can be considered as a further step towards "semantic encapsulation" simplifying program verification and optimization. In addition to this, the underlying programming model might be helpful for a better understanding of component-based programming approaches and distributed programming.

The presented techniques form the kernel of work in progress. Currently, we investigate the relation to modular verification (see [MPH99]) and the integration of the approach with other encapsulation techniques like module concepts and encapsulation by access modes (e.g. `private`, `protected`, ...).

References

[Alm97] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, 1997.

[Bok99] B. Bokowski. Implementing "object ownership to order". Presented at the Intercontinental Workshop on Aliasing in Object-Oriented Systems at ECOOP'99), 1999. Available from `http://cuiwww.unige.ch/~ecoopws/iwaoos/papers/index.html`.

[BV99] B. Bokowski and J. Vitek. Confined types. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, 1999.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[Hog91] J. Hogg. Islands: Aliasing protection in object-oriented languages. In A. Paepcke, editor, *OOPSLA '91 Conference Proceedings*, pages 271–285, October 1991. SIGPLAN Notices, 26 (11).

[MPH99] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 1999. (to appear).

[NVP98] J. Noble, J. Vitek, and J. M. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98: Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

A Appendix

Subtype Relation on Type Schemes. The subtype relation $\preceq_S$ on type schemes is the smallest reflexive, transitive relation satisfying the following axioms:

$$\begin{array}{rcl@{\qquad}rcrcl}
\mathit{nullS} & \preceq_S & \mathit{grndS(C)} & S \preceq_J T & \Leftrightarrow & \mathit{grndS(S)} & \preceq_S & \mathit{grndS(T)} \\
\mathit{nullS} & \preceq_S & \mathit{objS(C)} & S \preceq_J T & \Leftrightarrow & \mathit{objS(S)} & \preceq_S & \mathit{objS(T)} \\
\mathit{nullS} & \preceq_S & \mathit{typeS(C,R)} & S \preceq_J T & \Leftrightarrow & \mathit{typeS(S,R)} & \preceq_S & \mathit{typeS(T,R)} \\
\mathit{grndS(T)} & \preceq_S & \mathit{roS(T)} & S \preceq_J T & \Leftrightarrow & \mathit{roS(S)} & \preceq_S & \mathit{roS(T)} \\
\mathit{objS(T)} & \preceq_S & \mathit{roS(T)} & & & & & \\
\mathit{classS(T)} & \preceq_S & \mathit{roS(T)} & & & & &
\end{array}$$

Subtype Relation on Types. The subtype relation $\preceq$ an types is the smallest reflexive, transitive relation $\mathit{Type} \times \mathit{Type} \to \mathit{bool}$ satisfying the following axioms ($\preceq_J$ denotes the subclass relation in Java):

$$\begin{array}{lcl@{\qquad}rclcl}
\mathit{nullT} & \preceq & \mathit{refT(U,C)} & S \preceq_J T & \Leftrightarrow & \mathit{refT(U,S)} & \preceq & \mathit{refT(U,T)} \\
\mathit{refT(U,T)} & \preceq & \mathit{roT(T)} & S \preceq_J T & \Leftrightarrow & \mathit{roT(S)} & \preceq & \mathit{roT(T)}
\end{array}$$