A Type System for Controlling Representation Exposure in Java

Peter Müller and Arnd Poetzsch-Heffter

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

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 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 for Java and similar languages 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 presents the universe programming model. The universe type system is informally described in Section 3. Section 4 demonstrates the application of universes. Our conclusions are contained in Section 5.

2 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 do 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 ([MPH00a]) 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: (a) Objects outside a universe must not reference objects inside. Otherwise, they could use these references to manipulate the internal state of the component.[1] (b) Objects inside a universe must not reference 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) Components might want to pass parts

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

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 (but can depend on their identities).

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. For example, 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. Section 4 sketches the implementation of the list/iterator example.

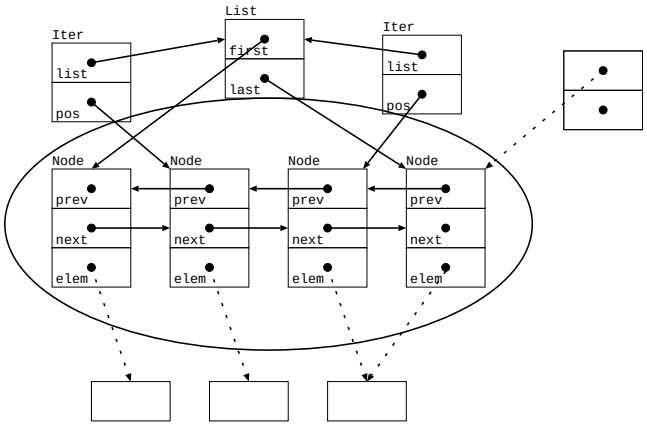

Fig. 1. Object Structure for List/Iterator Example

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

3 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.

Component Programming Model and Universes. We simplify the component programming model as follows: (1) We associate every object with its own *object universe*. That is, 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. 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.

The use of type universes reduces the amount of sharing control that can be done. For instance, 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. [MPH00a] for a discussion).

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. A formalization of the type system and a sketch of the type safety proof can be found in [MPH99,MPH00b].

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 Section 2, 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* (`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.

The subtype relation on type schemes resembles the subtype relation on types. 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.

Informal Type Rules. Three basic rules guarantee type safety of the universe type system (cf. [MPH00b] for a formalization): (1) A type scheme combinator (see appendix) is used to determine the type schemes for fields accesses and method invocations. The resulting type scheme must not be *undefined* 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 object scheme field on an object scheme 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

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.

This invariant guarantees 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.

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

4 Example

We illustrate the application of object and type universes, and of read-only types by two implementations of a doubly linked list. 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 list implementation 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`.

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

At first sight, the above example does not require the usage of universes since no `List` method returns a reference to a `Node` object. However, the universe type system guarantees that subclasses of `List` cannot introduce additional methods that violate representation containment. And, what is even more important, it prevents programmers from accidently writing classes that give away references to representation objects.

Lists with Iterators. By a variant of the above example, we demonstrate the use of type universes. The example shows how list iterators can be realized. 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()     { ... }                                 }
```

5 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.

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.

[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.

[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. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999. Technical Report 263, URL: `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[MPH00a] 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, 2000.

[MPH00b] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. *Software Practice and Experience*, 2000. (submitted).

A Appendix

Type Scheme Combinator. To determine the type scheme of method invocations and field accesses, the following table is used, where the type scheme of the target of the method invocation/field access determines the line, and the result/parameter/field type scheme determines the column (all combinations not mentioned in the table yield *undefined*). For instance, the field access expression `v.f` with `v` and `f` having type schemes `D<obj>` and `C`, resp., has type scheme `C<obj>`.

	`C`	`C<obj>`	`C<T>`	`C<ro>`	`boolean`	`int`
`D`	`C`	`C<obj>`	`C<T>`	`C<ro>`	`boolean`	`int`
`D<obj>`	`C<obj>`	*undefined*	*undefined*	`C<ro>`	`boolean`	`int`
`D<S>`	`C<S>`	*undefined*	*undefined*	`C<ro>`	`boolean`	`int`
`D<ro>`	`C<ro>`	`C<ro>`	`C<ro>`	`C<ro>`	`boolean`	`int`