Universes:
A Type System for Alias and Dependency Control

Peter Müller and Arnd Poetzsch-Heffter
Email: [Peter.Mueller, Arnd.Poetzsch-Heffter]@Fernuni-Hagen.de
Fachbereich Informatik
Fernuniversität Hagen
D-58084 Hagen

Abstract

We present a type system that allows one to express a hierarchical partitioning of the object store into so-called universes. The partitioning is used to control aliasing and dependencies. Alias control restricts object sharing. Dependency control is used to limit the set of objects on which class invariants or abstractions may depend. The type system guarantees an invariant on execution states that enables modular verification. It combines strong type constraints for read-write references with the flexibility of read-only references. This combination makes it capable of specifying certain implementation patterns that comparable approaches cannot handle (e.g., binary methods, iterators for containers). The report introduces the universe type system, shows its application to typical programming patterns, explains its benefits for modular verification, and provides a formal type safety proof.

1 Introduction

Sharing mutable objects is typical for object-oriented programs. As a direct consequence of the concept of object identity, 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 *representation exposure*. Thus, arbitrary objects can use these references to manipulate the internal state of components without using the component interface. On the other hand, objects inside the components can reference objects outside. If the invariants or abstract values of components (in the sense of [Hoa72]) depend on the state of outer objects, modifications of the outer objects can effect properties of the component. This makes OO-programs very hard to reason about. In particular, components cannot be verified in a modular way. 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.

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 alias and dependency control while retaining a flexible sharing model. It is easy to apply and guarantees an invariant that is strong enough for modular verification. The universe type system is a conservative extension of the Java type system. That is, each type correct Java program is also type correct w.r.t. the universe type system. Therefore, programmers can flexibly annotate their programs with refined type information according to the amount of alias and dependency control they need. 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 motivation for the work and the general approach. The universe type system, its application, and the corresponding invariant on execution states is introduced in Section 3. Section 4 formalizes the universe type system for a Java subset and contains the type safety proof. Related work is discussed in Section 5, conclusions are given in Section 6.

2 Motivation and Approach

Sharing objects through aliasing provides the flexibility of the object-oriented programming model and simplifies the effective use of computational resources. On the other hand, it often leads to programming errors (cf. e.g. [BV99] for a bug in the implementation of the security package of JDK 1.1.1). This section motivates alias and dependency control. In particular, it demonstrates why we need such control mechanisms for modular verification of object-oriented programs. It illustrates shortcomings of current techniques and introduces our approach.

2.1 Modular Verification and Representation Invariants

Whereas alias control is a nice programming feature in general, it is a *necessity* for modular verification of object-oriented programs. Modular verification means to prove the behavior of a program module without knowing the contexts in which the module will be used. As an example consider a framework to administer the data of companies. It contains a module with a class `Company`. Company objects have a field `employees` referencing the list of employees. We use the Java library class `Vector` to implement such lists:

```
public class Company {
  protected Vector  employees;
  // invariant:  i!=j  =>  employees.elementAt(i) != employees.elementAt(j)
  ... }
```

As invariant, we want to guarantee that different positions in the list refer to different employees. This invariant obviously simplifies many algorithms working on the employee vector (e.g., salary updates). The invariant has to be maintained by all methods in class `Company`. This can be proven by classical verification techniques [PH97]. Unfortunately, that does not guarantee that the invariant holds for all objects of type `Company`. Other objects outside the sketched module can get a reference to an employee vector and violate the invariant. As an example, consider the following module with the subclass `OpenCompany` and a client class `Violator`. Method `violate` adds a second reference to the first employee in `v` to the vector, thereby violating the invariant:

```
public class OpenCompany extends Company {
  public Vector getEmployees() { return employees; }
}

public class Violator {
  public void violate() {
       OpenCompany c = new OpenCompany();           // invariant holds
       Vector v = c.getEmployees();                 // invariant still holds
       if (v.size() > 0)  v.add(v.elementAt(0)); }  // invariant violated
}
```

The example illustrates that a class invariant can be broken by methods in other modules. In particular, it is not sufficient to require that subclasses are behavioral subtypes (e.g., `OpenCompany` is a behavioral subtype of `Company`; cf. [LW94] for an introduction to behavioral subtyping): The specification of `Company` is too weak to show that the invariant holds in all legal program contexts where `Company` may be used. (A program context is legal if subtypes are behavioral subtypes according to the given specifications.) To solve this problem, we have to express that objects of class `Company` or its subclasses are not allowed to give away a reference to the employee list that enables the modification of the list. That is, either no such reference is passed out or the passed out references do not enable modification. We call this property *representation encapsulation.*

To a certain extent, we can realize both alternatives by classical access and visibility restrictions. For example, instead of the library class `Vector` we could use a class private to the module of `Company` to implement the employee list. Thus, classes outside the module could not invoke methods of this class (making it impossible for a violator to call methods such as `add`). However, using private classes requires reimplementation of existing classes.

Furthermore, it does not allow subclasses declared in other modules to access the employee list, which restricts inheritance and specialization in an unacceptable way. The other alternative is to make the field `employee` private, which has similar drawbacks for subclasses and requires one to specify and prove that the methods of class `Company` do not pass out a reference to the employee list. Such a leak is not always as evident as in class `OpenCompany` since it might be indirect via other objects as illustrated by the following method of class `Company` that indirectly passes a reference to the employee vector to a `Bank` object, from where it could leak to a violator:

```
public void payStockOptions( Bank b, Vector amount ) {
  Order o = new Order(employees, amount);
  b.payOptions(o); }
```

Unfortunately, it is not possible to directly express representation encapsulation by classical specification techniques based on pre- and postconditions, invariants, modifies-clauses, and history constraints. These techniques allow one to specify functional properties of given methods and properties of the object store. In particular, invariants can be used to express aliasing patterns on the object store (so-called static aliasing [HLW+92]). However, the classical techniques are not capable of expressing properties of methods that are added in subtypes. In particular, the notion of behavioral subtyping is not strong enough to guarantee representation encapsulation (as illustrated by the `OpenCompany` example). We need a technique that allows us to specify that the employee vector belongs to the representation of `Company` and may not be passed out, neither by class `Company` nor by subtypes.

2.2 Ownership Model

To specify representation encapsulation, we build on recently developed techniques for alias control (cf. Section 5). The basic idea underlying these techniques is to distinguish between an object X and those objects that are used to implement the behavior of X. The latter objects form the *representation* of X.

There are different techniques to define which objects are considered to represent an object. We explain one of them in Section 3. Based on a binary relation on objects—the so-called *ownership relation* that is given somehow—we recursively define the representation in terms of ownership. The idea underlying the ownership relation is that an object X is the owner of all objects that are directly used to represent X. For instance, a `Company` object would be the owner of the `Vector` object referenced by field `employees`. The representation of an object X for a given ownership relation contains[1] all objects owned by X or by objects belonging to the representation.

The ownership relation can be considered as a directed graph on the allocated objects in a program state where an edge goes from X to Y if X is the owner of Y. We say a program *realizes the ownership model*, if in all states the ownership relation forms a forest, that is, a set of disjoint trees. At the roots of these trees are the objects that have no owner, that is, objects that do not belong to any representation. The fact that the ownership relation forms a forest implies the following facts: 1. Objects do not belong to their own representation. 2. Each representation has a unique owner. 3. Representations of different objects are either disjoint or one is contained in the other (*representation containment*). As abbreviation, we

[1] More precisely, it is the smallest set satisfying the recursive equation.

say object Y is *inside* X if it belongs to the representation of X; Y is *outside* X if it is not inside.

The ownership model can be used to formulate restrictions on the permissible reference structures between objects. This way flexible alias control can be realized. One important step to integrate the ownership model into programming languages was the work on ownership types described in [CPN98]. It defines the ownership relation by means of a parametric type system and uses static type checking to guarantee the following property of the references between objects: Whenever there is a path in the reference structure from an object outside some object X to an object inside X, this path goes through X. That is, X can control access to its representation. We call this property *strong alias control*. (Note that this property would be violated by a call to method `getEmployees` in subclass `OpenCompany`. Such a call could create a path to the `Vector` object that does not pass through the corresponding `Company` object.)

Strong alias control cannot be used if we need or want to have multiple references from outside into a representation. Unfortunately, this situation is fairly common and necessary for several programming patterns. We consider three such patterns:

1. In container classes, iterators are very desirable to step through the elements of the container. To work efficiently, iterator objects need to have references into the representation of the container. From the perspective of modular verification, such references are not problematic, because they are usually defined in the same module as the container. However, strong alias control forbids iterators.

2. Often methods are needed to compare two objects and their representation. A typical example is a test whether all elements of two doubly linked lists are equal. Such a method needs simultaneous access to the representations of the compared objects.

3. In many situations, it is helpful to share information between different objects in a way that only one object is allowed to modify the information. As an example, we consider an extension of the company class by a field holding the address of the company:

```
public class Company {
    protected Address address;
    public Address  getAddress() { return address; }
    ... }

// Declared in some other module
public class Address {
  private String street;
  public void setStreet(String s) { street = s; }
  ... }
```

   Address objects are used to share the address information between a company and clients. Thus, the company can modify the address without the need to propagate the changes to the clients. The drawback here is that clients are able to change the company address as well. Strong alias control forbids such patterns altogether avoiding unwanted modifications. For modular verification, we do not need to be that strict. For example, to verify an invariant in class `Company` saying that all address fields are properly initialized, it is sufficient to guarantee that clients are provided with read access to the address only.

2.3 Approach

The goal of our research is to enable modular verification of object-oriented programs which requires specification techniques that allow us to express representation boundaries and semantics-based access restrictions as illustrated by the examples above. In this scenario, the distinction between inside and outside of objects is used in two ways. First, it defines a boundary for incoming references (*alias control*). In particular, it provides a mechanism to specify which references to the representation may be passed to the outside by subclasses. Second, it defines the set of objects on which invariants and abstractions may depend (cf. [Lei95] for an introduction to dependencies). For example, the invariant of class `Company` depends on the state of the referenced vector. This dependency is permissible, because we consider the vector to be part of the representation.

Similar to [CPN98], we control aliasing and dependencies by typing mechanisms. However, our situation is slightly different. We want to annotate and verify existing programs. In particular, we have to be able to handle container classes with iterators and comparison methods. On the other hand, we are not limited to typing. In situations where type information is too weak, we can complement it with information gained by verification. We developed a type system that follows the ownership model and supports read-only references. Read-only references are used

- to allow objects to hold references into representations of other objects (e.g., iterators and the pattern demonstrated by class `Address` can be implemented using read-only references);
- to mark outgoing references, that is, references that refer to objects outside; this is necessary to control dependencies;
- to provide a flexibility similar to the context parameters of ownership types.

The last item needs explanation. In [CPN98], context parameters are used to enable outgoing references. Context parameters are similar to parametric polymorphism. Instead of parameterizing over types, they parameterize over owners. As will be explained in the next section, our read-only types are supertypes of owner-specific read-write types. This way, they provide a kind of subtype polymorphism. The price is that we have to provide and use owner-specific downcasts.

The presented approach is easy to use and sufficiently simple to be integrated into our programming logic [PHM99]. It is expressive enough to handle subtyping and the mentioned programming patterns. Last but not least, downcasts are not so critical in a verification context, because they can be eliminated by proving suitable preconditions for the cast site.

3 Programming with Universe Types

In this section, we informally introduce the universe type system and describe how representation encapsulation is achieved. Read-only references enable external objects to access the internal representation of other objects, which is for example necessary to implement iterators and binary methods. Furthermore, we explain the invariant on the execution states that is guaranteed by the universe type system and describe its benefits for modular verification. The formal background of the presented concepts is given in Section 4.

3.1 Representation Encapsulation

The example in Section 2 demonstrated that objects must protect their representations from unwanted modification via aliases. In this subsection, we describe how we can use typing mechanisms to delimit the world inside the representation of an object from the world outside.

Universes. To keep things simple for the explanation, let us assume that an object-oriented program Π with classes $C_1, \ldots, C_n$ is given. The generalization to open programs in which not all classes are known is straightforward. The classes define a set of types together with a subtype relation. We call this set of types the *standard type universe* of Π. The basic idea of the universe type system is to use multiple "copies" of the standard universe (one could imagine to copy the whole program text and add a suitable postfix to the class names). Such a copy is called a *type universe*, or simply a *universe*. In each universe, there is a type C_i, but although structurally identical, the types in different universes are considered to be distinct. Each object X is created for a type of a given universe. That is, each object belongs to exactly one universe.

In addition to the standard universe, we assume a universe for each object X in a program execution (cf. Section 5 for other kinds of universes) and call X the *owner* of its universe. Notice that this implies a hierarchical structuring of the universes with the standard universe at the top. Each object in the standard universe has its own universe. Objects in these *child* universes are again owners of universes and so forth.

By type rules, we guarantee that the owner of a universe U is the only object not belonging to U that can have read-write references to objects belonging to U. Read-write references are normal object references providing read and write access to objects and their methods. They are distinguished from read-only references (see below). Consequently, read-write references either connect objects in the same universe or lead from one universe to a child universe.

Universes can easily be used to implement the ownership model. An object X puts all objects that are directly used to represent X into its universe. Consequently, the whole representation of X is contained in its universe and its descendants. The type rules guarantee that

1. objects inside the representation of X can only be read-write referenced by X or other objects of the representation; that is, universes guarantee strong alias control of read-write references;

2. objects inside the representation of X cannot have read-write references to objects outside the representation; that is, universes provide dependency control.

This invariant on the reference structure holds as well for local variables and formal parameters if they are considered as instance variables of the `this` object. Therefore, we can apply universes to control both static and dynamic aliasing [HLW+92].

Fig. 1 illustrates how the representation of a `Company` object can be encapsulated to protect it from modifications. (Objects are depicted by boxes; arrows depict read-write references; the representation of the `Company` object is encircled.) Note that neither references from objects outside the representation to the inside nor references from inside to the outside are permitted.

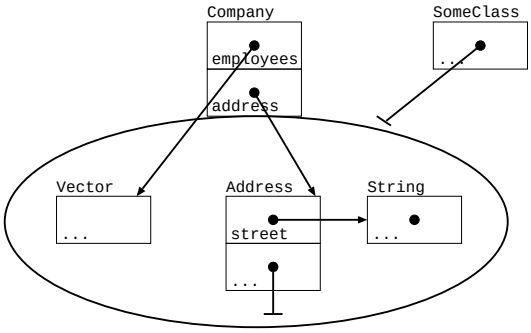

Figure 1: A `Company` Object and its Representation.

Programming with Universes. In programs, types have to be interpreted relative to the current `this` object $X_{\texttt{this}}$. A type identifier C without modifier refers to the type C in the universe to which $X_{\texttt{this}}$ belongs. A type identifier C with modifier `rep` refers to the type C in the universe owned by $X_{\texttt{this}}$.

Expressions of type `rep C` cannot be assigned to variables of type `C` since `C` and `rep C` correspond to different types. References of a type `rep C` always point into the universe owned by $X_{\texttt{this}}$. To prevent them from leaking into other universes, fields having rep-types and methods that have rep-types as return or parameter types can only be accessed/invoked on `this` (we will relax this restriction for read-only references, see below; the precise rules are given in Section 4.1). To illustrate this concept, we declare the vector of employees and the address to be part of the company's representation.

```
protected  rep Vector   employees;
protected  rep Address  address;
```

This prevents the owner from giving away a reference to the employee vector. For instance, `OpenCompany` cannot declare a method `getEmployees` as shown in Section 2 since the result type is not compatible with the type `rep Vector` of the field. On the other hand, if `getEmployees` had return type `rep Vector`, the method could only be invoked on `this`, which prevents the reference from leaking.

```
public     Vector  getEmployees() { return employees; }  // type error
public rep Vector  getEmployees() { return employees; }  // safe
```

3.2 References across Representation Boundaries

The universe type system as described in the previous subsection enables modular verification by enforcing representation encapsulation. However, it is too restrictive for many applications. One needs the capability of having references that leave an object's representation. For example, the representation of a list should not be forced to include the element objects of the list. In most cases, the list only stores references to its elements. The element objects are outside the list representation. In addition, several common programming patterns require that the representation of an object can be accessed by other objects, for example to implement iterators or test for structural equality. To support such idioms, we provide read-only references and functional methods. Read-only references are allowed to point into arbitrary universes. Functional methods enable one to observe aspects of the representation and are guaranteed not to make any modifications.

Read-Only References. Read-only references cannot be used to perform field updates or invocations of methods that potentially have side-effects (as opposed to functional methods, see below). Read-only references allow objects to make part of their representation accessible or at least referable without taking the risk that the representation is being modified. In addition to that, we use read-only references as markers in our specification framework for modular verification: The specification of a class (in particular invariants and abstractions) must not depend on the states of objects reachable only via read-only references.

Figure 2 shows how read-only references can be used to expose parts of a representation (here, the address) in a safe way (dashed arrows depict read-only references). Furthermore, they allow objects inside a representation to reference objects outside. For example, an address might contain data which is shared by all addresses (e.g., the domain of the email address).

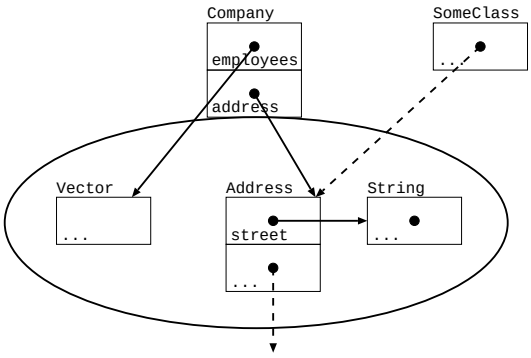

Figure 2: A `Company` Object with its Representation and Read-Only References.

In programs, we use `readonly C` for read-only references to objects of class `C` independent of the universe. Read-only types have three important properties:

1. They are supertypes of the corresponding read-write types. That is, `readonly C` is a supertype of type `C` in every universe.

2. It is not possible to use an expression of a read-only type as target for a field update or an invocation of a non-functional method. This is checked by context conditions.

3. Reading fields or invoking functional methods via read-only references yields again read-only references. Thus, it is not possible to gain a read-write reference through a read-only reference. Consequently, we can allow fields of rep-types and functional methods with rep-types as return types to be accessed/invoked on read-only references without violating representation encapsulation.

We support downcasts to convert a read-only reference into a read-write reference. If a read-only reference points into universe U, only the owner of U and objects belonging to U can downcast the read-only reference into a read-write reference. As with conventional downcasts, such casts need dynamic type checking which requires owner information to be stored for each object. However, dynamic checks can obviously be eliminated by static analysis or verification techniques.

Example. In the following, we discuss the implementation of a doubly linked list with iterators. The example demonstrates the application of universes and read-only types. In particular, it shows how multiple references into a representation can be realized, and how references to objects outside the representation are handled. The list nodes belong to the representation of the lists and are therefore protected from modifications. The elements, however, are declared of type `readonly Object` and can reside in any universe. Consequently, method `LinkedList.add` takes and `Iter.next` yields a read-only reference. Since the elements are not part of the list's representation, the list invariant may only depend on the identities of the elements, but not on their state. By using read-only references, iterators can have references to the internal node structure of the list. As illustrated by `LinkedList.remove`, the list (which is the owner of the node structure) can downcast these references to modify the node structure. The implementation of the `equals` method shows that read-only references can be used to simultaneously access two representations. In a similar way, other binary methods can be implemented as long as they do not require modification of the explicit parameter and its representation. Fig. 3 shows the object structure of a `LinkedList` with two iterators.

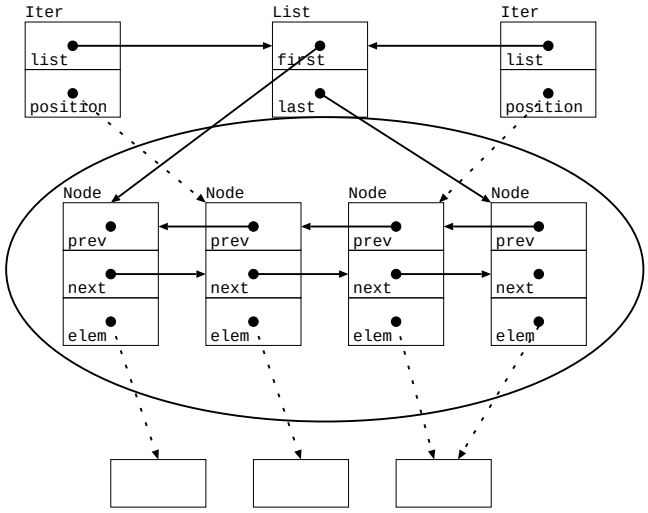

Figure 3: Object Structure for `LinkedList` Example.

```
class Node {
    public Node prev, next;
    public readonly Object  elem;  }

public class LinkedList {
    protected  rep Node  first, last;
    // invariant:  first and last are nodes of a doubly linked node structure

    public void add( readonly Object o )      { ... }
    public Iter getIter()                     { return new Iter(this); }
    protected void remove( readonly Node np ) { rep Node n = (rep Node) np; ... }
    public boolean equals( readonly LinkedList l ) {
      readonly Node f1 =   first;
      readonly Node f2 = l.first;
      while (f1 != last && f2 != l.last && f1.elem==f2.elem)
```

```
        { f1 = f1.next; f2 = f2.next; }
      return f1 == last && f2 == l.last && f1.elem==f2.elem;  }
    ... }

public class Iter {
    protected  LinkedList  list;
    protected  readonly Node  position;
    // invariant:  position belongs to the universe owned by list

    Iter( LinkedList l ) {
      list = l;  readonly LinkedList rol = l;
      position = rol.first; }
    public readonly Object next() {
      readonly Object result = position.elem;
      position = position.next;
      return result; }
    public void remove() { list.remove(position); }
    ... }
```

Functional Methods. Read-only references cannot be used to manipulate objects. Thus, methods that cause side-effects must not be invoked on read-only references. This property can be statically checked by either forbidding all method invocations on read-only references, or by introducing functional methods that can be statically checked to be side-effect free. Obviously, the former solution is not satisfactory since for example one would like to use `equals` to compare two objects referenced read-only.

Functional methods can be implemented very easily. If a method is declared to be functional, it must not contain field updates, or invocations of non-functional methods[2]. Furthermore, functional methods can only be overridden by functional methods. For example `LinkedList.equals` can be declared functional:

```
public functional boolean equals( readonly LinkedList l ) { ... }
```

3.3 The Universe Invariant and its Benefit for Verification

In this subsection, we explain the invariant on reference structures guaranteed by the universe type system and sketch its application to modular verification of classes.

Universe Invariant. In every execution state, the following invariant holds: If object X holds a direct reference to object Y[3] then either (1) X and Y belong to the same universe, or (2) X is the owner of Y, or (3) the reference is read-only. The invariant is an immediate consequence of the type safety lemma which is presented in the next section.

Modular Verification of Classes. The universe invariant simplifies modular verification. It is not the topic of this report to explain the underlying logical details. However, since we used modular verification as a major motivation of the presented type system, we want to illustrate the benefits of the universe invariant for verification. A central property in this

[2]Instead of forbidding all modifications of the object store, more sophisticated techniques (e.g., based on data flow analysis) can be used to provide functional methods that can create and modify temporary objects.

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

respect is formulated by the following modularity lemma for class invariants. Essentially, it says that class invariants cannot be broken by reasonable program extensions. The proof sketch demonstrates where the universe invariant provides necessary information.

Lemma: Let C be a class in program Π, and X an instance of C. We assume that (1) the invariant of X only depends on nonpublic fields of X that are declared in C, and fields of objects inside the representation of X; (2) all methods in Π preserve the invariant of X if they are invoked on objects outside the representation of X; (3) all subtypes are behavioral subtypes. That is, (a) all overriding methods meet the specifications of the overridden ones, and (b) subtype methods preserve the invariants of their supertypes. If Π' is an extension of Π then every method of Π' preserves the invariant of X if it is invoked on objects outside the representation of X, that is, in calls where the `this` object is outside the representation.

Proof Sketch: For simplicity we assume here, that our program does not contain recursive methods. That is, there is an order on methods such that every method invokes only methods that are less according to this order. We can prove the lemma by induction on this order. Let m be a method of class D in Π'.
Induction Basis: m does not contain method invocations. If D is in Π or D is a subtype of C, the property holds (assumption 2 resp. 3). Otherwise, the only way m can violate the invariant of X is by field updates. Let's assume that m contains a field update `v.f=w;`. We prove the property by case distinction on the three disjuncts of the universe invariant:

(1) `this` and `v` belong to the same universe: Since `this` is outside the representation of X, `v` is also outside. Thus, the invariant of X can only depend on `v.f` if `v` holds X and `f` is a nonpublic field of `C` (assumption 1). In this case, `f` is not accessible in D, in contradiction to the existence of the update.

(2) `this` is the owner of `v`: That is, `v` is inside `this`. `this` and X are different objects since D is not in Π. Thus, `v` is outside X because `this` is outside X. Therefore, the invariant of X does only depend on `v.f` if `v` holds X (assumption 1). The rest of the proof is identical to case (1).

(3) `v` is read-only: Field updates are not allowed on read-only references.

Induction Step. Again, the property holds if D is in Π or D is a subtype of C, (assumptions 2 and 3). Otherwise, m can violate the invariant of X by field updates or method invocation. The proof for field updates is identical to the induction basis. For method invocations `w=v.n(...);` we conclude in analogy to the induction basis by case distinction on the three disjuncts of the universe invariant:

(1) `this` and `v` belong to the same universe: Since `v` is not inside X, the invocation of `v.n(...)` preserves the invariant of X (induction hypothesis).

(2) `this` is the owner of `v`: `v` is not inside X (see induction basis). Therefore, `v.n(...)` preserves the invariant of X (induction hypothesis).

(3) `v` is read-only: Functional methods do not cause side-effects and can therefore not violate the invariant of X. □

The lemma above illustrates the use of the universe invariant for modular verification. A more elaborate treatment of modular verification based on universes can be found in [MPH00a].

4 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 and an operational semantics of our language, we prove type safety and the universe invariant.

4.1 Formalization of the Universe Type System

Programming Language. To simplify the description of the formalization of the universe type system, we concentrate on a Java subset enhanced with universe-specific constructs. The resulting language provides classes and inheritance, instance methods (functional and non-functional), instance fields, and local variables as well as statements for reading and writing instance variables, simple assignments (with casts), object creation, method invocation, sequential statement composition, conditional, and loop statement. The expressions of our Java subset are literals (integer, boolean, `null`), local variables/formal parameters, and the `this` reference. For simplicity, we assume that every method has exactly one explicit formal parameter `p`. We do not provide overloading and static method binding (no static or private methods, no invocations on `super`). Field names are assumed to be unique for each program (this can be achieved by prefixing each field name with the name of the class it is declared in). The extension of the formalization to a richer language is straightforward. The abstract syntax of the Java subset is presented in the appendix.

As described in the last section, the universe type system enables types to be annotated with universe modifiers. We call the annotated types *type schemes*. They are used wherever types occur in conventional Java programs (declaration of variables and method signatures, casts). Type schemes are described in more detail in the next paragraph.

Type Schemes. The universe type system provides *ground type schemes* (of the form `C`), *rep type schemes* (`rep C`), *read-only type schemes* (`readonly C`), and type schemes for the primitive types (`int`, `boolean`, the null type). The null type scheme must not occur in programs. Type schemes are formalized by the following data type where sort *ClassId* denotes the class identifiers as given in a program.

data type

TypeScheme	=	*grndS(ClassId)*
	|	*repS(ClassId)*
	|	*roS(ClassId)*
	|	*boolS* | *intS* | *nullS*

The subtype relation on type schemes follows the subclass relation in Java: Two ground schemes/rep schemes/read-only schemes are subtypes if the corresponding classes are subclasses. In addition, every read-only scheme is a supertype of the ground and rep scheme with the same class. An axiomatization of the subtype relation can be found in the appendix.

The Type Scheme Combinator. Type schemes describe the type of a program element (expression, field, method) relatively to the universe to which `this` belongs: Ground schemes denote that the referenced object belongs to the same universe as `this`, rep schemes indicate that the referenced object belongs to the universe owned by `this`, and read-only schemes

stand for references into arbitrary universes. Consequently, when a field or method is accessed/invoked on other variables than `this`, the type scheme of the field access or method invocation expression has to be determined by combining the type schemes of the target variable and the type scheme of the field, method result, or method parameter. For example, if a local variable `v` is declared to be of type scheme `rep T` (i.e., it holds a reference into the universe owned by `this`) and class `T` contains a field `f` of type scheme `S`, the type scheme of `v.f` is `rep S`. That is, the field holds a reference to an instance of class `S` or subclasses of `S` in the universe owned by `this`. Such combinations of type schemes are described by the *type scheme combinator*

$$* : \mathit{TypeScheme} \times \mathit{TypeScheme} \to \mathit{TypeScheme} \cup \{\mathit{undef}\}$$

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

	grndS(C)	repS(C)	roS(C)	boolS	intS	nullS
grndS(D)	grndS(C)	repS(C)	roS(C)	boolS	intS	nullS
repS(D)	repS(C)	*undef*	roS(C)	boolS	intS	nullS
roS(D)	roS(C)	roS(C)	roS(C)	boolS	intS	nullS

The definition of the type scheme combinator reveals four important aspects: (1) The class of the resulting type scheme is the class of the second argument. This is as in Java where the type of `v.f` is the type of `f`. (2) The combination of two rep schemes is not defined to ensure that it is not possible to gain read-write references to objects that are neither owned by `this` nor belong to the same universe as `this`. (Combining two rep schemes would mean to go "two steps down" in the universe hierarchy[4].) (3) If one of the arguments is a read-only scheme, the result is also a read-only scheme. This guarantees that read-only references are transitive, that is, it is not possible to gain a read-write reference through a read-only reference. (4) If the first argument is a type scheme for a primitive type, the result is undefined since such situations cannot occur in Java (e.g., it is not allowed to invoke methods on integer variables).

The definition of the type scheme combinator affects the context conditions for functional methods. In analogy to Java, the combination of the type scheme of the target and the actual parameter of a method invocation has to be a subtype of the type scheme of the formal parameter (see below for the type rules). In cases where the functional method is invoked on a read-only reference, this combination yields a read-only scheme. To support these cases, we require that all formal parameters of functional methods must be declared read-only to meet the requirement above. This rule is no restriction since functional methods must not modify their parameters anyway.

Type Rules. The universe-specific type rules are displayed in Fig. 4. All other rules for our Java subset are straightforward and therefore omitted. In the type rules, we use $[e]$ to denote the type scheme of an expression or field e. For literals, local variables, and fields, $[e]$ is defined by the program. In methods m of class C, [`this`] is *grndS(C)* if m is a non-functional method, and *roS(C)* if m is functional (recall that all formal parameters of functional methods a required to be read-only). We use *res(m)* and *par(m)* to refer to the result type scheme and the type scheme of the formal parameter of method m. $\preceq_S$ is the subtype relation on

[4]It would be type safe if this combination yielded a read-only scheme. However, we think that such a definition is rather unintuitive. If the read-only scheme is required, it can be achieved by an additional assignment.

TS $\preceq_S$ [v], TS $\preceq_S$ [e],	TS $\preceq_S$ [v], TS *is grndS or repS*
$\vdash$ `v=(TS)e;`	$\vdash$ `v=new TS();`
[f] *is repS* $\Rightarrow$ [w] *is roS*, [w] $*$ [f] $\preceq_S$ [v]	[this] $*$ [f] $\preceq_S$ [v]
$\vdash$ `v=w.f;`	$\vdash$ `v=this.f;`
[v] *is no roS*, [f] *is no repS*, [e] $\preceq_S$ [v] $*$ [f]	[this] *is no roS*, [e] $\preceq_S$ [this] $*$ [f]
$\vdash$ `v.f=e;`	$\vdash$ `this.f=e;`
m *is not functional*, *par*(m) *is no repS*, *res*(m) *is no repS*, [w] *is no roS*, [e] $\preceq_S$ [w] $*$ *par*(m), [w] $*$ *res*(m) $\preceq_S$ [v]	m *is not functional*, [e] $\preceq_S$ [this] $*$ *par*(m), [this] $*$ *res*(m) $\preceq_S$ [v]
$\vdash$ `v=w.m(e);`	$\vdash$ `v=this.m(e);`
m *is functional*[5], *res*(m) *is repS* $\Rightarrow$ [w] *is roS*, [e] $\preceq_S$ [w] $*$ *par*(m), [w] $*$ *res*(m) $\preceq_S$ [v]	m *is functional*, [e] $\preceq_S$ [this] $*$ *par*(m), [this] $*$ *res*(m) $\preceq_S$ [v]
$\vdash$ `v=w.m(e);`	$\vdash$ `v=this.m(e);`

Figure 4: Type Rules.

type schemes. The judgment $\vdash$ `stmt` expresses that statement `stmt` is well-typed in a given program. If the type scheme combinator occurs within a rule, the statement is only correctly types if the application of the combinator is defined.

Five aspects of the universe type rules need explanation: (1) The rule for casts/assignments is like in Java. Note that it allows read-only references to be cast to read-write references (see Section 3). (2) The rule for the new-statement forbids the creation of objects of read-only schemes since read-only schemes do not specify the universe the new object should belong to. (3) Rep schemes indicate that a reference points to an object owned by `this`. Therefore, fields of rep schemes or methods with rep schemes as parameter or result type schemes can only be accessed/invoked on `this` and read-only references. For example, if variable `v` is of a ground scheme (i.e., `v` and `this` belong to the same universe) and field `f` is of a rep scheme, `v.f` yields a reference to an object owned by `v`. That does only correspond to the universe programming model if (a) the reference is read-only or (b) `v` and `this` denote the same object. To enforce this condition statically, we require that only `this` and read-only references can be used to access/invoke fields/methods of rep schemes. (4) Neither writing field access nor invocation of non-functional methods is allowed on read-only references. (5) The type rules do not require functional methods to be side-effect free. However, this requirement is necessary for modular verification (see Section 3.3).

4.2 Type Safety

In this subsection, we present the operational semantics of our Java subset. Based on this semantics, we formalize and prove type safety.

[5]Recall that parameters of functional methods must have primitive or read-only type schemes.

Operational Semantics of the Java Subset

Capturing Statement Contexts. The semantics of a statement depends on the context of the statement occurrence. We assume that the program context of a statement is always implicitly given and that we can refer to method declarations in this context. Method declarations are denoted by T@m where m is a method name in class T. *MethDeclId* is the sort of such identifiers. The function

$$body : \ MethDeclId \to Stmt$$

maps each method declaration to the statement constituting its body. If T is a type for class C and m a method of C, the function

$$impl : \ Type \times MethodId \to MethDeclId \cup \{undef\}$$

yields the corresponding declaration; otherwise it yields *undef*. Note that C can inherit the declaration of m from a superclass.

Values. Values in the Java subset are either integers, booleans, the null reference, or references to objects. As described in Subsection 3.1, each object belongs to exactly one universe.

```
data type
  Value  =  b( Bool )
          |  i( Int )
          |  null()
          |  ref( ClassId ObjId Universe )
```

Values constructed by *ref* represent references to objects. The identity of an object is determined by its class, its object identifier, and the universe it belongs to. The sort *ObjId* denotes some suitable set of object identifiers. The sort *Universe* is defined below.

Universes and Types. A universe is either the standard universe or the universe owned by an object (identified by its class, object identifier, and universe). As explained in Section 3, a universe contains one type for every class in a program. Consequently, we formalize read-write reference *types* (not *type schemes*) as tuples of class identifiers and universes. Besides read-write reference types, we have read-only reference types and the primitive types. The subtype relation $\preceq$ on types resembles the subtype relation on type schemes. Its axiomatization is contained in the appendix.

```
data type
  Universe  =  stdU( )
            |  repU(ClassId ObjId Universe)
```

```
data type
  Type  =  refT(ClassId Universe)
        |  roT(ClassId)
        |  booleanT()
        |  intT()
        |  nullT()
```

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 an object that determines the universe of the type. For variables, this object is the `this` object. For instance variables and method incarnations, it is the target object. The interpretation of a type scheme w.r.t. an object is formalized by function τ. It maps read-only and primitive schemes to the corresponding types; for ground and rep schemes

T, $\tau(T, X)$ yields the read-write reference type for T in the universe to which X belongs or the universe owned by X, resp.

$$
\begin{array}{ll}
\tau : \mathit{TypeScheme} \times \mathit{Value} \to \mathit{Type} \cup \{\mathit{undef}\} & \\
\quad \tau(\mathit{grndS}(C), \mathit{ref}(C, O, U)) & = \mathit{refT}(C, U) \\
\quad \tau(\mathit{repS}(C), O) & = \mathit{refT}(C, \mathit{mkrepU}(O)) \\
\quad \tau(\mathit{roS}(C)), O) & = \mathit{roT}(C) \\
\quad \tau(\mathit{boolS}, O) & = \mathit{booleanT} \\
\quad \tau(\mathit{intS}, O) & = \mathit{intT} \\
\quad \tau(\mathit{nullS}, O) & = \mathit{nullT}
\end{array}
$$

Furthermore, we use the following auxiliary functions: $\mathit{mkrepU} : \mathit{Value} \to \mathit{Universe} \cup \{\mathit{undef}\}$ yields the universe owned by an object; function $\mathit{univ} : \mathit{Value} \to \mathit{Universe} \cup \{\mathit{undef}\}$ yields the universe an object belongs to; for non-reference values both functions yield *undef*. Function $\mathit{typeof} : \mathit{Value} \to \mathit{Type}$ yields the type of a value.

Execution. A statement is essentially a partial state transformer. A state in our Java subset describes (a) the current values for the local variables and for the method parameters `p` and `this`, and (b) the current object store.

The state of an object is given by the values of its instance variables. We assume a sort *InstVar* for the instance variables of all objects and a function

$$\mathit{iv} : \ \mathit{Value} \times \mathit{FieldId} \to \mathit{InstVar} \cup \{\mathit{undef}\}$$

where $\mathit{iv}(V,\mathrm{f})$ is defined as follows: If V is an object reference and the corresponding object has an instance variable named `f`, this instance variable is returned. Otherwise *iv* yields *undef*. The state of all objects and the information whether an object is alive (i.e., allocated) in the current program state is formalized by an abstract data type Object Store with sort *Store* and the following functions:

$$
\begin{array}{lll}
\_\langle\_ := \_\rangle & : \mathit{Store} \times \mathit{InstVar} \times \mathit{Value} & \to \ \mathit{Store} \\
\_\langle\_, \_\rangle & : \mathit{Store} \times \mathit{ClassId} \times \mathit{Universe} & \to \ \mathit{Store} \\
\mathit{new} & : \mathit{Store} \times \mathit{ClassId} \times \mathit{Universe} & \to \ \mathit{Value} \\
\_(\_) & : \mathit{Store} \times \mathit{InstVar} & \to \ \mathit{Value} \\
\mathit{alive} & : \mathit{Value} \times \mathit{Store} & \to \ \mathit{Bool}
\end{array}
$$

$OS\langle IV := V\rangle$ yields the object store that is obtained from OS by updating instance variable IV with value V. Object creation is described by two functions: $OS\langle T, U\rangle$ yields the object store that is obtained from OS by allocating a new object of class T in universe U, and $\mathit{new}(OS, T, U)$ yields a reference to an object of type $\mathit{refT}(T, U)$ that is not alive in OS. $OS(IV)$ yields the value of instance variable IV in store OS. If V is an object reference, $\mathit{alive}(V, OS)$ tests whether the referenced object is alive in OS. A formalization of these functions can be found in [PHM98].

Program states are formalized as mappings from identifiers to values. To have a uniform treatment for variables and the object store, we use \$ as identifier for the current object store:

$$\mathit{State} \ \equiv \ (\mathit{VarId} \cup \{\,\mathrm{this},\, \mathrm{p}\,\} \to \mathit{Value} \cup \{\mathit{undef}\}) \times (\{\,\$\,\} \to \mathit{Store} \cup \{\mathit{undef}\})$$

For $S \in \mathit{State}$, we write $S(\mathrm{x})$ for the application to a variable or parameter identifier and $S(\$)$ for the application to the object store. By $S[\mathrm{x} := V]$ and $S[\$:= OS]$ we denote the state that is obtained from S by updating variable x and \$, resp. The canonical evaluation

of expression e in state S is denoted by $\epsilon(S, \mathrm{e})$ yielding an element of sort *Value* or *undef* (note that expressions always terminate and do not have side-effects). The state in which all variables are undefined is named *initS*. The SOS-rules for our Java subset are contained in the appendix.

Proof of Type Safety

We call a type system *type safe*, if it guarantees that every valid execution state is well-typed. An execution state S is well-typed if

1. for every local variable/formal parameter v and `this` $typeof(S(\mathrm{v})) \preceq \tau([\mathrm{v}], S(\mathrm{this}))$, and
2. for every valid instance variable $iv(S(\mathrm{x}), \mathrm{f})$ $typeof(S(\$)(iv(S(\mathrm{x}), \mathrm{f}))) \preceq \tau([\mathrm{f}], S(\mathrm{x}))$ holds.

A state S is *well-formed*—denoted by *wf(S)*—if it is well-typed and in addition

3. $S(\mathrm{this}) \neq null$ holds.

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 that have reference types are initialized to *null*. X belongs to the standard universe. Execution starts by invoking a designated method of X. Therefore, the initial state is well-formed. The proof of type safety is based on the following lemma about the type scheme combinator $*$:

Combination Lemma: In all well-formed states S with $S(\mathrm{w}) \neq null$, the equality

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

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, [w] is not a *roS*, [f] is not an *repS*, and [this] is a *grndS*).

This equality formalizes the fact that the combination $[\mathrm{w}] * [\mathrm{f}]$ interpreted w.r.t. $S(\mathrm{this})$ yields the same type as if [f] was interpreted w.r.t. $S(\mathrm{w})$ (see Subsection 4.1). The proof of the lemma runs by case distinction on the type schemes of `w` and `f`. It is straightforward and does not reveal any interesting aspects.

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

Proof Sketch: The proof of the type safety lemma runs by structural induction over the operational semantics. In the appendix, we present two of the most interesting cases of the proof: Field update (one case of the induction basis) and invocation of non-functional methods (one case of the induction step). The other cases are very similar.

The Universe Invariant Revisited

The universe invariant (cf. Subsection 3.3) is an immediate consequence of type-safety. Two read-write reference types can only be subtypes if they belong to the same universe. Therefore, the type safety statement (parts 1 and 3 for local variables and formal parameters, and part 2 for instance variables) and the definition of τ imply the universe invariant, where the three disjuncts of the invariant correspond to ground, rep, and read-only schemes, resp.

5 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 holding references into one representation). In this subsection, we compare the universe type system to other approaches to alias control w.r.t. these objectives.

Type Systems. Ownership types [CPN98] provide a very flexible means for alias control. They realize the ownership model with strong alias control by a parametric type system. So-called context parameters are used to provide references from inside a representation to the outside. Context parameters are similar to parametric polymorphism. Instead of parameterizing over types, they parameterize over owners. Ownership types are statically checkable. However, context parameters make ownership types rather difficult to apply [Bok99]. Read-only types can replace context parameters in many situations and lead to programs that are easier to read and reason about. Furthermore, they allow multiple objects to access one representation which is not supported by ownership types. 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. For example, the mode `rep` enforces representation containment (like the rep scheme). 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. The mode `free` indicates that the referenced object is not aliased. Therefore, free variables behave like unique variables (see below). Like ownership types, alias modes have been presented for a language without subtyping and inheritance.

Balloon types [Alm97] aim at full representation encapsulation. That is, all objects reachable from an object are contained in its balloon (as if every field was declared as a rep scheme). 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. Islands permit 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. Confined types have been designed

for the development of secure systems. They do not support representation encapsulation on the object level, which makes some aspects of verification difficult.

In this report, we associate universes with single objects. For certain applications, it is interesting to provide multiple owner objects sharing a common representation. In [MPH00a, MPH00b], we presented variants of the universe type system that associate universes with types or modules such that all objects of one type or all objects of the types declared in one module own a common representation. That allows several objects outside a representation to reference and modify objects inside. Therefore, these variants give more flexibility than universes on the object level, but provide weaker alias control. The type rules and formalization of universes on the type and module level are very similar to the concepts presented here.

We developed the universe type system systematically from the requirements of modular verification and formalized it similarly to ownership types [CPN98]. Since the universe type system provides one type universe for each object, it is closely related to value-dependent types [XP99]: Type schemes can be seen as types that depend on a value, namely an owner object. Read-only types correspond to existentially quantified dependent types (there exists an owner for the referenced object). For future work, we plan to formalize the universe type system as a restriction of a type system with value-dependent types.

Unique Variables. The reference held by a unique variable is the only reference to the referenced object [Wad90, Hog91, Min96, Boy00]. Unique variables are usually realized by a destructive read operation. They provide very strict alias control since they completely forbid sharing of objects referenced by a unique variable. However, the techniques developed for uniqueness support programming patterns that cannot be handled by the universe type system, such as objects that exchange their representations, or capturing, which often occurs when the representation for a newly created object is passed to the constructor method by a read-write reference (see [DLN98] for an example). We plan to investigate combinations of uniqueness and universes to support such idioms in the future.

Read-only Types and Functional Methods. [KT99] realizes read-only types in Java by implicitly generating an interface for every type declaration. This interface contains only the signatures of functional methods. To achieve transitive protection, read-only types are used as result types for these methods. This is a common technique, often proposed as a design pattern for write-protecting objects. Like in our approach, read-only types are supertypes of the user-defined types. Read-only types as described in [KT99] have three major drawbacks: (1) Since they are not directly supported by the type checker or runtime checks, inspection, reflection, or casts can be used to break the write-protection. (2) Java interfaces provide only public methods. Thus, they cannot be used to provide read-only access to the protected interface. (3) Functional methods do not modify the state of `this`, but are not guaranteed to be side-effect free. Thus, they are not *functional* is our sense. This is also true for `const` member functions in C++: They only forbid to *directly* manipulate the state of the *implicit* parameter. However, explicit parameters can be modified and thus, via aliasing, also the object referenced by `this` [Str91].

6 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 allow objects inside a representation to be referenced by arbitrary objects outside. The universe type system is easy to apply and does not impose much effort on programmers. We proved type safety of the universe type system and derived an invariant on the execution states.

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

Acknowledgment

We thank John Boyland for his valuable comments on an earlier version of this report.

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`.

[Boy00] J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 2000. (to appear).

[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.

[DLN98] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research Report 156, Digital Systems Research Center, 1998.

[HLW+92] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. Report on ECOOP'91 workshop W3: The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[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).

[KT99] G. Kniesel and D. Theissen. JAC — Java with transitive readonly access control. 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`.

[Lei95] K. R. M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.

[LW94] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[Min96] N. Minsky. Towards alias-free pointers. In P. Cointe, editor, *ECOOP '96 European Conference on Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer-Verlag, 1996.

[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. A type system for controlling representation exposure in Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[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.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, Jan. 1997. URL: `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[PHM98] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 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.

[Str91] B. Stroustrup, editor. *The C++ Programming Language, 2nd Edition.* Addison-Wesley, 1991.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods (PROCOMET)*, 1990.

[XP99] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. Principles of Programming Languages*, pages 214–227. ACM Press, New York, 1999.

APPENDIX

Abstract Syntax of the Java Subset.

data type

Program	=	list of *ClassDecl*
ClassDecl	=	*ClassDecl(ClassId ClassId ClassBody)*
ClassBody	=	list of *MemberDecl*
MemberDecl	=	*FieldDecl(Mode TypeScheme FieldId)*
	|	*MethodDecl (Mode TypeScheme MethodId TypeScheme VarList Stmt)*
	|	*ROMethodDecl (Mode TypeScheme MethodId TypeScheme VarList Stmt)*
Mode	=	*Private() | Default() | Protected() | Public()*
VarList	=	list of *VarDecl*
VarDecl	=	*Vardcl(TypeScheme VarId)*
TypeScheme	=	*grndS(ClassId) | repS(ClassId) | roS(ClassId) | boolS | intS | nullS*
Stmt	=	*Seq(Stmt Stmt)*
	|	*While(Exp Stmt)*
	|	*If(Expr Stmt Stmt)*
	|	*Invoc(VarId VarId MethodId Expr)*
	|	*New(VarId ClassId)*
	|	*GetAttr(VarId VarId FieldId)*
	|	*SetAttr(VarId FieldId Expr)*
	|	*CastAssign(VarId ClassId Expr)*
Expr	=	*Var(VarId) | IntLiteral(Int) | BoolLiteral(Bool) | Null() | This()*

Rules of Operational Semantics. To have a compact notation, we treat `this` like a formal parameter in the SOS rules.

$$\frac{S(\mathrm{v}) \neq \mathit{null},\ \mathit{initS}[\mathrm{this} := S(\mathrm{v}), \mathrm{p} := \epsilon(S,\mathrm{e}), \$:= S(\$)] : \mathit{body}(\mathit{impl}(\mathit{typeof}\,(S(\mathrm{v})), \mathrm{m})) \ \to S'}{S :\ \mathrm{w{=}v.m(e);}\ \to S[\mathrm{w} := S'(\mathrm{result}), \$:= S'(\$)]}$$

$$\frac{\mathit{true}}{S :\ \mathrm{v{=}new\ C();}\ \to S[\mathrm{v} := \mathrm{new}(S(\$),\mathrm{C},\mathit{univ}(S(\mathrm{this}))), \$:= S(\$)\langle \mathrm{C}, \mathit{univ}(S(\mathrm{this}))\rangle]}$$

$$\frac{\mathit{true}}{S :\ \mathrm{v{=}new\ rep\ C();}\ \to S[\mathrm{v} := \mathrm{new}(S(\$),\mathrm{C},\mathit{mkrepU}(S(\mathrm{this}))), \$:= S(\$)\langle \mathrm{C}, \mathit{mkrepU}(S(\mathrm{this}))\rangle]}$$

$$\frac{S :\ \mathrm{stm1}\ \to S',\ S' :\ \mathrm{stm2}\ \to S''}{S :\ \mathrm{stm1\ stm2}\ \to S''} \qquad \frac{\mathit{typeof}\,(\epsilon(S,\mathrm{e})) \preceq \tau(\mathrm{T}, S(\mathrm{this}))}{S :\ \mathrm{v{=}(T)e;}\ \to S[\mathrm{v} := \epsilon(S,\mathrm{e})]}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{true}),\ S :\ \mathrm{stm}\ \to S',\ S' :\ \mathrm{while(e)\{stm\}}\ \to S''}{S :\ \mathrm{while(e)\{stm\}}\ \to S''} \qquad \frac{\epsilon(S,\mathrm{e}) = b(\mathit{false})}{S :\ \mathrm{while(e)\{stm\}}\ \to S}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{true}),\ S:\ \mathrm{stm1}\ \rightarrow S'}{S:\ \mathrm{if(e)\{stm1\}\ else\{stm2\}}\ \rightarrow S'} \qquad \frac{\epsilon(S,\mathrm{e}) = b(\mathit{false}),\ S:\ \mathrm{stm2}\ \rightarrow S'}{S:\ \mathrm{if(e)\{stm1\}\ else\{\ stm2\ \}}\ \rightarrow S'}$$

$$\frac{S(\mathrm{v}) \neq \mathit{null}}{S:\ \mathrm{w = v.f;}\ \rightarrow S[\mathrm{w} := S(\$)(\mathit{iv}(S(\mathrm{v}),\mathrm{f}))]} \qquad \frac{S(\mathrm{v}) \neq \mathit{null}}{S:\ \mathrm{v.f{=}e;}\ \rightarrow S[\$:= S(\$)\langle \mathit{iv}(S(\mathrm{v}),\mathrm{f}) := \epsilon(S,\mathrm{e})\rangle]}$$

Subtype Relation on Type Schemes. The subtype relation $\preceq_S$ on type schemes is the smallest reflexive, transitive relation satisfying the following axioms ($\preceq_J$ denotes the subclass relation as defined by the Java program):

$$\begin{array}{rcl} \mathit{nullS} & \preceq_S & \mathit{grndS(C)} \\ \mathit{nullS} & \preceq_S & \mathit{repS(C)} \\ \mathit{grndS(T)} & \preceq_S & \mathit{roS(T)} \\ \mathit{repS(T)} & \preceq_S & \mathit{roS(T)} \end{array} \qquad \begin{array}{rcl} S \preceq_J T & \Leftrightarrow & \mathit{grndS(S)} \preceq_S \mathit{grndS(T)} \\ S \preceq_J T & \Leftrightarrow & \mathit{repS(S)} \preceq_S \mathit{repS(T)} \\ S \preceq_J T & \Leftrightarrow & \mathit{roS(S)} \preceq_S \mathit{roS(T)} \end{array}$$

Subtype Relation on Types. The subtype relation $\preceq$ on types is the smallest reflexive, transitive relation satisfying the following axioms ($\preceq_J$ denotes the subclass relation as defined by the Java program):

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

Two Cases of the Type Safety Proof. Each statement transforms a state S into a state S' as defined by the SOS rules. We have to prove $\mathit{wf}(S) \Rightarrow \mathit{wf}(S')$, that is, that S' meets the three well-formedness conditions presented in Section 4.

Field Update: We consider a field update statement `v.f = e;`. From the context conditions, we know that all requirements are met to apply the combination lemma for [v], [f], and S.
ad 1: Let `x` be a local variable, a formal parameter, or `this`.

$\mathit{wf}(S) \Rightarrow \mathit{typeof}(S(\mathrm{x})) \preceq \tau([\mathrm{x}], S(\mathrm{this})) \Rightarrow$
$\mathit{typeof}(S[\$:= S(\$)\langle \mathit{iv}(S(\mathrm{v}),\mathrm{f}) := \epsilon(S,\mathrm{e})\rangle](\mathrm{x})) \preceq \tau([\mathrm{x}], S[\$:= S(\$)\langle \mathit{iv}(S(\mathrm{v}),\mathrm{f}) := \epsilon(S,\mathrm{e})\rangle](\mathrm{this}))$

ad 2: Let O and F be a *Value* and a *FieldId* such that *iv(O,F)≠undef*. For *iv(O,F)≠iv(S(*v*),*f*)*, the proof obligation is a direct consequence of the well-typedness of S. Otherwise, we conclude:

$[\mathrm{e}] \preceq_S [\mathrm{v}] * [\mathrm{f}]$
$\Rightarrow$ $[S(\mathrm{this}) \neq \mathit{null},\ \mathit{type\ rule}]$
$\tau([\mathrm{e}], S(\mathrm{this})) \preceq \tau([\mathrm{v}] * [\mathrm{f}], S(\mathrm{this}))$
$\Rightarrow$ $[\mathit{wf}(S), \mathrm{e}$ is either a constant or a variable, parameter, or `this`$]$
$\mathit{typeof}(\epsilon(S,\mathrm{e})) \preceq \tau([\mathrm{v}] * [\mathrm{f}], S(\mathrm{this}))$
$\Rightarrow$ [combination lemma]
$\mathit{typeof}(\epsilon(S,\mathrm{e})) \preceq \tau([\mathrm{f}], S(\mathrm{v}))$
$\Rightarrow$
$\mathit{typeof}(S[\$:= S(\$)\langle \mathit{iv}(S(\mathrm{v}),\mathrm{f}) := \epsilon(S,\mathrm{e})\rangle](\$)(\mathit{iv}(S(\mathrm{v}),\mathrm{f}))) \preceq \tau([\mathrm{f}], S[\$:= S(\$)\langle \mathit{iv}(S(\mathrm{v}),\mathrm{f}) := \epsilon(S,\mathrm{e})\rangle](\mathrm{v}))$

ad 3: $\mathit{wf}(S) \Rightarrow S(\mathrm{this}) \neq \mathit{null} \Rightarrow S[\$:= S(\$)\langle \mathit{iv}(S(\mathrm{v}),\mathrm{f}) := \epsilon(S,\mathrm{e})\rangle](\mathrm{this}) \neq \mathit{null}$

Method Invocation: We consider an invocation of a non-functional method `w=v.m(e);` . From the context conditions, we know that all requirements are met to apply the combination lemma for [v], [p]/*res(m)*, and S. To be able to apply the induction hypothesis, we have to show $wf(S) \Rightarrow wf(initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)])$:
ad 1: Let `x` be a local variable or the formal parameter of m, or `this`. For $\text{x} \neq \text{p} \wedge \text{x} \neq \text{this}$, we conclude $S(\text{x}) = initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)](\text{x})$. For the other cases, we derive (*class* yields the *ClassId* of a *Type*):

x=this:
$typeof(initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)](\text{this})) = typeof(S(\text{v})) =$
$refT(class(typeof(S(\text{v}))), univ(S(\text{v})) = \tau(grndS(class(typeof(S(\text{v})))), S(\text{v}))$
$\preceq$ [Since `v` is the target of the invocation, the scheme of `this` is determined by the class of $S(\text{v})$]
$\tau([\text{this}], S(\text{v})) = \tau([\text{this}], initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)](\text{this}))$
x=p: We consider only the interesting case that `e` is not a constant.
$[\text{e}] \preceq_S [\text{v}] * [\text{p}]$
$\Rightarrow$ $[S(\text{this}) \neq null]$
$\tau([\text{e}], S(\text{this})) \preceq \tau([\text{v}] * [\text{p}], S(\text{this}))$
$\Rightarrow$ $[wf(S), \text{combination lemma}]$
$typeof(S(\text{e})) \preceq \tau([\text{p}], S(\text{v}))$
$\Rightarrow$ [`e` is not a constant $\Rightarrow S(\text{e}) = \epsilon(S,\text{e})$]
$typeof(\epsilon(S,\text{e})) \preceq \tau([\text{p}], initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)](\text{this}))$
$\Rightarrow$
$typeof(initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)](\text{p})) \preceq \tau([\text{p}], initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)](\text{this}))$

ad 2: The well-typedness of instance variables follows directly from $wf(S)$.
ad 3: $initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)](\text{this}) = S(\text{v}) \neq null$

Since $initS[\text{this} := S(\text{v}), \text{p} := \epsilon(S,\text{e}), \$:= S(\$)]$ is well-formed, we can assume the induction hypothesis for S'. Now we have to show $wf(S[\text{w} := S'(\text{result}), \$:= S'(\$)])$
ad 1: Let `x` be a local variable or the formal parameter of m, or `this`. For $\text{x} \neq \text{w}$, we conclude $S(\text{x}) = S[\text{w} := S'(\text{result}), \$:= S'(\$)](\text{x})$. For $\text{x} = \text{w}$, we derive:

$[\text{v}] * res(m) \preceq_S [\text{w}]$
$\Rightarrow$ $[S(\text{this}) \neq null]$
$\tau([\text{v}] * res(m), S(\text{this})) \preceq \tau([\text{w}], S(\text{this}))$
$\Rightarrow$ [combination lemma]
$\tau(res(m), S(\text{v})) \preceq \tau([\text{w}], S(\text{this}))$
$\Rightarrow$ $[wf(S')]$
$typeof(S'(\text{result})) \preceq \tau([\text{w}], S(\text{this}))$
$\Rightarrow$
$typeof(S[\text{w} := S'(\text{result}), \$:= S'(\$)](\text{w})) \preceq \tau([\text{w}], S[\text{w} := S'(\text{result}), \$:= S'(\$)](\text{this}))$

ad 2: The well-typedness of instance variables follows directly from $wf(S')$.
ad 3: $S[\text{w} := S'(\text{result}), \$:= S'(\$)](\text{this}) = S(\text{this}) \neq null$ □